GraphBinMatch：基于图的跨语言二进制代码与源代码匹配相似性学习

Ali TehraniJamsaz、Hanze Chen、Ali Jannesari
艾奥瓦州立大学，艾姆斯，艾奥瓦州，美国{
tehrani, hanzech,jannesar}@iastate.edu

arXiv:2304.04658v1 [cs.SE] 10 Apr 2023

摘要——**二进制代码与源代码之间的相互匹配在计算机安全、软件工程和逆向工程等不同领域有着广泛应用。尽管已有一些方法尝试将源代码与二进制代码进行匹配，以加快逆向工程进程，但其中大多数都专注于单一编程语言。然而，在现实中，程序会根据需求使用不同的编程语言开发。因此，跨语言二进制代码与源代码匹配近年来受到越来越多的关注。尽管如此，由于跨编程语言匹配二进制代码与源代码这一问题本身存在困难，现有方法在做出准确预测方面仍面临挑战。**

本文研究跨语言二进制源代码匹配问题。我们提出GraphBinMatch，这是一种基于图神经网络的方法，能够学习二进制代码与源代码之间的相似性。我们在多项任务上评估GraphBinMatch，例如跨语言二进制代码与源代码匹配，以及跨语言源代码匹配。我们还评估了该方法在单语言二进制代码与源代码匹配任务上的性能。实验结果表明，GraphBinMatch 显著优于最先进的方法，F1 分数最高提升了 15%。

索引词—**跨语言、代码相似性、二进制源代码匹配**

I. 引言

二进制代码是一组可由计算系统直接执行的指令，而程序员编写的源代码则可读且易于理解。二进制代码与源代码匹配是一种评估二者相似性的技术。这是许多安全软件工程任务中的重要环节，例如漏洞 [1] 、恶意软件检测 [2] 和逆向工程 [3], [4]。

通常，在将二进制代码与源代码匹配时，我们要么希望找到与二进制文件匹配的源代码，要么希望找到与源代码文件匹配的二进制代码。例如，当我们拥有一个二进制代码片段时，若能检索到与之相似的源代码片段，将有助于开展逆向工程任务。检索到的源代码片段使研究人员能够了解二进制代码片段的功能。另一方面，如果我们有一段存在漏洞的源代码片段，将其与二进制代码形式进行匹配，有助于确定该漏洞是否存在于二进制文件中。

现有方法尝试衡量二元与源代码之间的语义相似度。然而，大多数研究侧重于二元与二元之间或源代码与源代码之间的匹配[5]–[7]。二元到源代码的匹配并非易事，因为其中涉及两种模态：二元和源代码。近期的研究一直尝试衡量二元与源代码之间的相似度；然而，它们在跨不同编程语言进行二元到源代码匹配方面仍有不足。最近，Gui 等人 [8]提出了一种基于 Transformer 的神经网络，用于学习跨编程语言的二元与源代码相似度；他们将中间表示（IR）作为模型的输入数据。然而，他们将 IR 视为词元序列，因此模型难以学习跨编程语言的相似度。

在本文中，我们提出了GraphBinMatch。一种基于图神经网络的方法，用于学习二元与源代码之间的语义相似性。不同于以往的方法，二元和源代码借助三种类型的流被视为图：控制流、数据流和调用流。将二元和源代码表示为图，有助于神经网络模型更好地学习代码的结构和语义，因为源代码包含特定的结构，不同于自然语言文本。

实验结果表明，GraphBinMatch优于最先进的方法，将F1从0.65提升至0.79，将召回率从0.59提升至0.82，并将精确率从0.73提升至0.76。

总的来说，本文的主要贡献如下：1）将该问题表述为学习图之间的相似性。2）开发了一种专用图神经网络，作为 GraphBinMatch 的骨干网络，用于学习图的相似性。3）在一套全面的任务上评估 GraphBinMatch。4）该方法不仅适用于跨语言场景，也适用于单语言场景。5）与最先进的方法相比，性能最高提升 15%。

本文其余部分结构如下：我们首先在第二节对二进制代码与源代码匹配问题进行形式化定义。然后，在第三节概述并阐释我们提出的方法。在第四节讨论并说明实验设置。第五节展示评估结果，随后在第六节讨论一些见解。接下来，我们在第七节概述相关工作，并

最后，第 VIII 节总结了本文并阐述了未来的研究工作。

II. 问题形式化

我们将跨语言二进制代码匹配检测问题表述如下：给定两个分别以二元形式和源代码形式表示的程序 P_a 和 P_b，它们使用两种不同的编程语言编写。我们的目标是训练一个深度学习模型来学习函数 γ，以预测这两个输入程序是否构成二进制代码匹配对，还是不匹配对。

训练集由三元组 (P_a，P_b，y_{ab}) 组成，其中 y_{ab} 是标签。我们将数据集中从同一编码任务收集的所有二元程序与源程序配对视为正样本，并标记为 1，表示它们是二进制源代码匹配对。相反，我们将数据集中由不同任务生成的所有二元程序与源程序配对视为负样本，并标记为 0，表示它们不是二进制源代码匹配对。

$$\gamma(P_a, P_b) = \begin{cases} 1, & \text{if } P_a \text{ and } P_b \text{ are matching} \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

III. 方法

本节将介绍我们提出的方法，用于识别不同编程语言中的二进制源代码匹配对。整体工作流程如图 1 所示。GraphBinMatch 的输入由两个文件组成：一个源文件和一个以不同编程语言编写的二进制文件。首先，使用相应的特定语言前端和工具编译源文件并反编译二进制文件，生成与语言无关的中间表示（IR）。接下来，我们创建图来刻画程序的控制流、数据流和调用流。为了更好地建模不同类型的节点和边，我们将这些图视为异构图。图中的边表示两个节点之间的关系。这些异构图是我们的图二进制匹配相似度神经网络（GraphBinMatch）的输入。

A. 中间表示

如前所述，我们提出的方法的第一步是将源语言和二元文件转换为中间表示。中间表示（中间表示）是现代编译器用于表示程序的一种中间形式。中间表示提供了更直接的抽象。它允许编译器中的多个转换和分析阶段更高效地生成目标代码。将程序降低为中间表示是简化和优化代码的常用编译器技术。在 GraphBinMatch 中，我们首先将输入文件转换为LLVM IR。借助 LLVM IR，我们的方法可以使用与语言无关的格式表示输入文件，从而更容易比较使用不同编程语言编写的代码。

我们使用两个前端 JLang [1] 和 Clang [2]，分别将 Java和 C++ 程序转换为 LLVM IR。JLang 支持最高至第 7版的 Java，而 Clang-5.0 将 C++程序转换为LLVMIR。值得一提的是，LLVM 的不同版本可能会生成略有不同的中间表示，因此使用与 JLang内置使用的相同版本的Clang，有助于让 Java 和 C++ 程序之间拥有更多相似之处。

我们提出的方法使用 RetDec[3] 从二进制可执行文件生成 LLVM IR。RetDec 是一款开源机器码反编译器，可从二进制可执行文件生成 LLVM IR。它支持多种架构，包括 x86、ARM、MIPS 和 PowerPC。RetDec 使用指令解析、数据流分析和控制流重建技术，对二进制代码进行逆向工程并将其转换为 LLVM IR。

B. 图生成

尽管已有多种方法 [9]–[11] 可将 LLVM IR 表示为深度学习模型的输入，但近期研究表明，将程序表示为图，有助于深度学习模型更有效地学习程序的特征 [12]–[14]。鉴于近期将程序表示为图的方法取得了成功，我们也使用 LLVM IR 文件构建图。具体而言，我们使用 ProGraML [14] 为数据集中的每个 LLVM IR 文件生成一张图。ProGraML 从 LLVM IR 中提取信息，并构建由不同类型的节点（即指令、变量和常量）及边（即控制流、数据流和调用流）组成的图。这些图捕捉程序的结构和语义信息，可作为机器学习模型的输入，用于执行代码相似性检测等各种程序分析任务。这些图必须经过编码，并输入图神经网络，以学习匹配对之间的相似性以及非匹配对之间的差异。

C. 节点特征嵌入

为生成节点嵌入，我们将这一过程分为两个部分：源代码预处理和模型内部处理。在源文件预处理过程中，我们使用 ProGraML 生成的图中每个节点的 full_text 或 text 属性内容，为该节点创建一个特征向量。full_text 属性表示节点对应的完整 LLVM 中间表示指令，而 text 属性仅包含相应的指令类型。例如，指令 `%16 = load i32 i32 %15 align 8` 表示加载整数指针 %15，并将其存储到临时变量 %16 中。我们只能从 text 属性中获取指令类型。然而，使用 full_text 属性，我们可以确定 load 指令负责加载整数指针。这些额外信息可用于更好地训练嵌入层并提高模型的准确率。

[1] https://polyglot-compiler.github.io/JLang [2] https://clang.llvm.org [3] https://github.com/avast/retdec

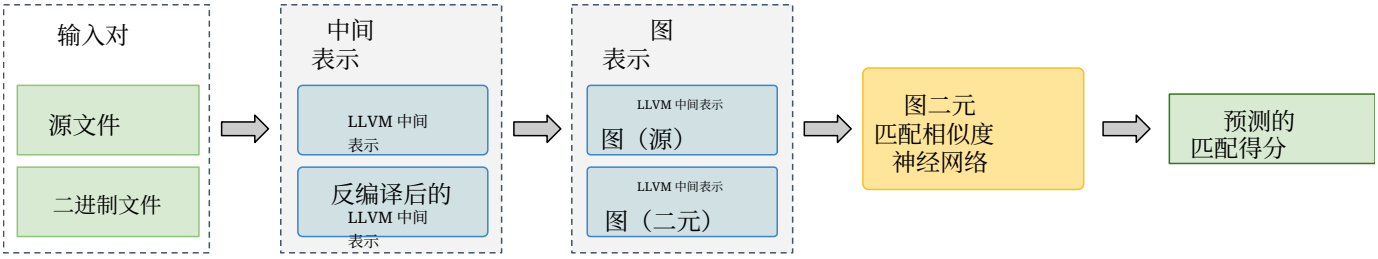


图 1：我们提出的方法概览。

我们发现，在预处理过程中，并非所有节点都具有完整文本_属性，因为有些节点仅表示只有文本段的编译器配置。为了解决这一问题，当完整_文本属性不可用时，我们使用文本属性作为备用选项。

ProGraML 论文本身使用节点的文本属性作为节点特征；然而，在后面的实验结果部分，我们将看到，使用节点的完整_文本属性可以提升 GraphBinMatch 的预测效果。此外，我们使用 ProGraML 提供的位置属性作为边特征。ProGraML 中的位置属性包含边的位置信息。例如，对于一条有效的 LLVM 中间表示指令，如 % result= add i32 %1, %2，ProGraML 会分别为 %1 和 %2 生成一条对应的边。在此示例中，第一条边的位置为 0，第二条边的位置为 1。

最后，我们使用分词器对这些 LLVM 中间表示指令进行分词，以创建最终的节点特征，然后将其添加到图中。分词器还能够将每个词元映射为相应的整数。因此，我们可以得到一个由整数构成的序列，用于表示节点的一条 LLVM 中间表示指令。该整数序列被视为节点的特征。在转换过程中，我们会将所有 LLVM 中间表示变量（例如 $12）转换为名为 [VAR] 的特殊词元。通过使用分词器并仔细选择截断和填充长度，我们确保特征向量包含丰富的信息，并能在后续建模步骤中得到有效利用。

LLVM 中间表示指令的长度各不相同，因此，我们图中每个节点的特征集长度也各不相同。为解决这个问题并确保所有节点的特征长度相同，我们将所有节点特征向量的平均长度向上取整至最近的 2 的幂，作为最终的截断长度。例如，如果数据集中所有 LLVM 中间表示指令在分词后的平均长度为 50，则最终截断长度为 64。长度为 64 的所有特征向量都将被截断，而长度小于 64 的所有特征向量都将使用 [PAD]token 填充。

D. 图二进制匹配相似度神经网络

预处理完成后，我们将把图传递给我们的图二进制匹配相似度神经网络（GraphBinMatch）进行进一步处理。GraphBinMatch 是基于 SimGNN [15], ，但我们进行了多项修改，使其适用于异构图相似度任务，相关内容将在后续章节中讨论。

GraphBinMatch 的架构如图 2 所示。为了训练 GraphBinMatch，我们创建由三个项目组成的数据点：**源文件 A、二进制文件 B**和**标签**。我们使用嵌入层作为 GraphBinMatch的第一层。该层处理每个节点的特征向量，并尝试通过学习嵌入空间来嵌入这些节点，使相似节点具有相似的嵌入。嵌入层将整个特征向量的维度增加到二维。我们利用最大值操作将二维特征向量降为一维。这个一维特征向量作为图卷积层的输入。

在以下小节中，我们将详细说明 GraphBinMatch 的各个组成部分。

1）异构卷积：

GraphBinMatch 旨在接收两个文件的图表示，并预测它们是否匹配。卷积层通过使用 pytorch-geometric 库中的HeteroConv 包装器来支持异构图。如前所述，我们的异构图表示具有三种类型的关系。该层包含三个彼此分离的GATv2Conv[16] 层，用于对每种关系进行建模。GATv2Conv 层的输出会堆叠在一起，然后计算逐元素最大值，以获得节点的潜在表示。在每个 GATv2Conv 之后，我们都会加入额外的层归一化，以稳定训练并防止过拟合。

2）注意力:

在 GraphBinMatch 中，我们采用了类似于 [15]中介绍的注意力层。该注意力层通过获取前一层的节点嵌入，并将其传递 through 注意力机制来运行。我们使用一个全局图嵌入向量 $c \in \mathbb{R}^D$ ，其中 D 与节点的嵌入维度相同。 c 通过对经过非线性变换的节点嵌入求平均来创建。该全局嵌入向量 c 包含图的整体结构和信息。

为了计算给定节点 n_i 的注意力，我们计算 c 与 n_i 的潜在表示之间的内积。这种方法背后的直觉是，与图的整体上下文更相似的节点应获得

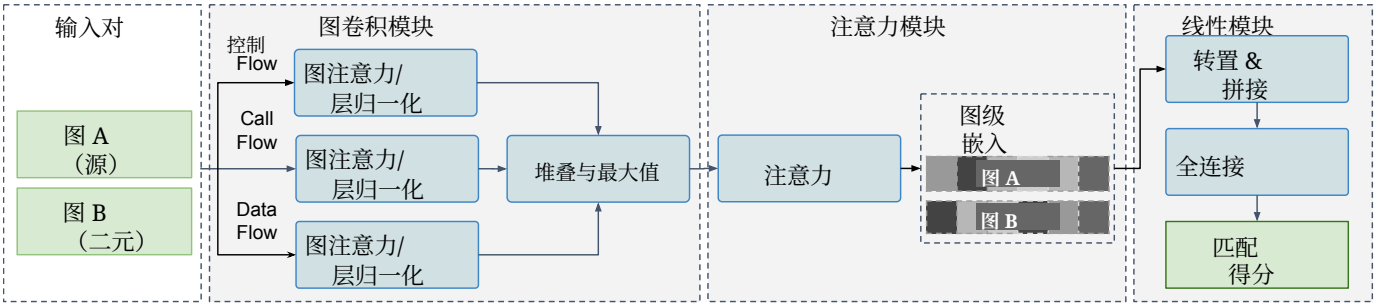


图 2：图二进制匹配相似度神经网络（GraphBinMatch）的结构 .

更高的注意力权重。最后，通过计算所有节点的加权和来生成图级嵌入。这里，节点的权重指的是节点的注意力值。

3) 全连接层*:*

计算出两个文件的图级嵌入后，将这两个向量拼接并转置，再输入全连接层以得到最终预测。在本研究中，我们使用两个全连接层。第一个全连接层之后还会应用一个归一化层，以稳定学习过程。此外，我们在最后一个线性层之前引入了一个额外的 dropout 层，以增强模型的非线性和正则化能力。

在下一节中，我们评估 GraphBinMatch，并将其结果与当前最先进的方法进行比较。

IV. 实验设置

在本节中，我们评估 GraphBinMatch，展示结果，并回答四个研究问题。

A. 任务与研究问题

我们在跨语言和单语言任务上评估GraphBinMatch。对于这些任务中的每一项，我们使用特定的数据集。我们将研究问题定义如下

- **RQ1:** 在跨语言二进制源代码匹配问题中，基于图的表示和图神经网络能否优于基于 Transformer 的模型?
- **RQ2:** 当应用于具有不同优化级别的单语言场景时，GraphBinMatch 能否提供一致的结果?
- **研究问题3：** GraphBinMatch 在不同编译器下的性能如何?
- **RQ4：** GraphBinMatch 在源代码匹配方面表现良好吗?

B. 数据集统计信息

跨语言代码匹配。为了评估 GraphBin- Match 在跨语言二进制源代码匹配任务中的表现，我们使用CLCDSA 数据集 [17], ，该数据集由从两个编程竞赛网站收集的源代码文件组成：AtCoder 和Google CodeJam。这些编程竞赛网站包含多个任务，而该数据集包含使用各种编程语言编写的、作为这些任务解决方案的源代码。我们选取了 c、c++ 和 Java 作为我们研究中的编程语言，用于评估我们的模型学习不同编程语言之间二元与源代码相似性的有效性。

为确保我们的数据集具有均衡的正负样本分布，我们将同一任务的有效解视为匹配对，将不同任务的有效解视为非匹配对。我们会剔除所有无法编译的文件。遵循基线论文，我们采用相同的训练集、验证集和测试集划分比例，即按 6:2:2 的比例划分数据集。

如前所述，在本研究中，我们使用 JLang[4] 和 clang-5.0 作为所有 Java 源代码文件的编译器。JLang 是一个开源编译器前端，可将 Java 源代码编译为相应的 LLVM 中间表示。该编译器用于为 Java 文件生成所有相应的 LLVM 中间表示表达式。为了将 LLVM 中间表示转换为二进制可执行文件，我们使用 Clang-5.0。在整个实验过程中，除非明确提及优化级别，否则编译器的默认优化级别设为 0z。为了将所有二进制可执行文件转换为对应的 LLVM 中间表示，我们使用开源反编译器 RetDec。

同语言代码匹配 对于检测同一编程语言中的匹配关系这一任务，我们使用 POJ-104 数据集 [18] [5]。该数据集包含来自在线评测系统（OJ）的 500 名学生针对 104 道不同编程题提交的 C++ 语言解决方案，该系统用于教育目的。我们使用不同优化级别的 Clang 和 GCC，将 C++ 源代码文件编译为 LLVM 中间表示和二进制可执行文件。为了将二进制可执行文件反编译为其对应的 LLVM 中间表示，我们对所有二进制可执行文件采用了 RetDec。

表 I 展示了两个数据集的统计信息。

C. 基线

如前所述，我们针对两种不同的匹配任务评估 GraphBinMatch 的性能：二进制源代码匹配和源代码-源代码匹配。为了评估 GraphBinMatch 的有效性，我们将其与用于二进制源代码匹配的 BinPro [3], B2SFinder[7], 和 XLIR 进行了比较。

BinPro 是一款旨在解决识别源代码与二元代码之间相似性这一挑战的工具，甚至还可用于.....

[4]https://github.com/polyglot-compiler/JLang [5] https://drive.google.com/uc?id=0B2i-vWnOu7MxVlJwQXN6eVNONUU

表 I：数据集统计信息

	语言	# 源	# LLVM 中间表示	# 二进制文件	# 反编译的 LLVM—中间表示
CLCDSA	C	15605	13929	14370	13929
	C++	16676	15375	15766	15589
	Java	19836	15124	17072	15124
POJ-104	C++	52000	38598	38598	37909

当所使用的编译器或优化级别未知时。为此，BinPro采用机器学习技术，计算用于确定二元到源代码相似性的最佳代码属性。随后，使用静态分析工具提取并计算这些代码属性，并通过二分匹配算法匹配二元代码和源代码。

B2SFinder通过推断二进制代码和源代码中的七种可追踪特征来检测二进制代码克隆。它采用一种加权特征匹配算法，能够处理不同的特征，并根据代码特征实例的特异性及其出现频率计算权重。

XLIR 是一种基于 Transformer 的神经网络模型，目前在二进制与源代码匹配方面达到了最先进的水平。顾名思义，XLIR 也使用 LLVM IR。为了嵌入 LLVM IR 中的词元，XLIR 利用一个预训练的 BERT 模型。模型首先使用大型外部 LLVM 中间表示语料库，通过掩码语言建模（MLM） [19] 作为预处理步骤，对神经网络进行预训练。此步骤旨在学习有意义的 LLVM 中间表示词元表征。词元嵌入后，XLIR 将它们映射到一个公共空间，并使用三元损失函数联合学习 LLVM 中间表示表征。这种方法使 XLIR 能够跨不同编程语言匹配二进制源代码。

对于源-源匹配，除了XLIR 之外，我们还将结果与 LICAA [20] 进行比较。**LICCA** 基于源代码运行，并提取程序的语义和句法特征，以识别匹配对。

D. 实验设置

GraphBinMatch使用 Pytorch-Geometric[6],这一用于开发图深度学习模型的强大库。为了优化 GraphBin-Match的学习参数，使用 Adam [21]优化器，学习率为 $6.6e^{-5}$，并使用二元交叉熵作为损失函数。我们使用 GATv2 [16] 作为图卷积层，正如前文所述。GraphBinMatch 的超参数使用 RayTune[7]进行调优。使用 HuggingFace tokenizer[8] 对 LLVM 中间表示指令进行分词，这些指令在我们的图表示中充当节点的特征。

GraphBinMatch 由一个维度为 128 的 PyTorch 嵌入层（用于嵌入词元）、五个维度为 256 的 GATv2 图卷积层，以及一个最大词汇量为 2048 的 Hugging-Face GPT 分词器组成。在 GraphBinMatch 模型中，我们使用 LeakyReLU 作为激活函数，但最后一个线性层后接 Sigmoid 函数。我们使用四块配备 80GB 显存的 NVIDIA A100 GPU，以及一颗配备 128GB 内存的 Intel Xeon 6140 CPU 来训练 GraphBinMatch。

[6]https://pytorch-geometric.readthedocs.io [7] https://docs.ray.io/en/latest/tune/index.html [8] https://huggingface.co/docs/transformers/index

E. 评估指标

参数	预测结果	实际结果
真正例（**TP**）	匹配	匹配
真阴性**(TN)**	非匹配	不匹配
假阳性**(FP)**	匹配	不匹配
假阴性（**FN**）	不匹配	匹配

表 II

在代码匹配检测领域，精确率（P）、召回率（R）和F1 分数（F1）通常用于评估模型的性能和准确性。这三个标准源自四项指标：真正例（TP）、真负例（TN）、假正例（FP）和假负例（FN）。这些指标的定义见表 II。

精确率通常用于描述模型正向预测的准确性，即被识别为匹配的配对的准确性。其定义为模型预测的所有匹配配对中真实匹配配对所占的比例，如公式 2 所示。

$$P = \frac{TP}{TP + FP} \tag{2}$$

召回率 被定义为数据集中匹配对中模型正确预测为匹配对的百分比，这正是公式 3 所描述的情况。

$$R = \frac{TP}{TP + FN} \tag{3}$$

F1分数通常用于评估模型的性能。F1 分数定义为精确率和召回率的调和均值，即公式 3 所描述的情况

$$F1 = \frac{2PR}{P + R} \tag{4}$$

V. 实验结果

在本节中，我们将回答研究问题。

1) ***RQ1***：在跨语言二进制源代码匹配问题中，基于图的表示和图神经网络能否优于基于 ***Transformer*** 的模型？*:*

为回答 RQ1 的问题，我们使用 CLCDSA 数据集训练了模型。我们通过将 GraphBinMatch 在二进制源代码匹配任务中的表现与上述基线进行比较来评估其性能，如表 III 所示。

表 III：跨语言二进制配对任务的性能（阈值为 0.5）。

	C/C++二进制代码与 Java 源代码			Java 二进制代码与 C/C++ 源代码		
	精确率	召回率	F1	精确率	召回率	F1
BinPro	-	-	-	0.36	0.37	0.36
B2SFinder	-	-	-	0.35	0.41	0.38
XLIR（LSTM）	0.62	0.53	0.57	0.55	0.51	0.53
XLIR（Transformer）	0.73	0.59	0.65	0.68	0.55	0.61
GraphBinMatch	0.75	0.73	0.74	0.75	**0.78**	0.77
GraphBinMatch（Tokenizer）	**0.76**	**0.82**	**0.79**	**0.76**	0.77	**0.77**

*由于 BinPro 和 B2Sfinder 在处理 Java 源代码方面存在局限，我们无法提供 Java 源代码的测试结果。*本表及后续表格中其他工具的结果均引自基准论文。

在实验中，我们将二元 C/C++ 程序的 LLVM 中间表示和 Java 源代码作为 GraphBinMatch 的输入。结果表明，GraphBinMatch 效果显著，其精确率、召回率和 F1 分数分别达到 0.76、0.82 和 0.79。与基线论文相比，这一结果提升了 20% 以上。为进一步增强结果的稳健性，我们还使用 Java 二元程序的 LLVM 中间表示和 C/C++ 源代码进行了补充实验。结果令人满意，精确率、召回率和 F1 分数分别为 0.76 、 0.77 和 0.77，比基线高出 25%。

表 III 显示，对于相同的任务和数据集，使用二进制 Java 源代码与二进制 C/C++代码的性能存在差距。这种差距可能归因于反编译获得的 LLVM 中间表示与从源代码获得的 LLVM 中间表示之间存在某些差异，而 GraphBinMatch 难以理解这些差异。反编译得到的 LLVM 中间表示并不总是与源代码完全一致，我们认为这种差异主要由两个因素造成。首先，反编译后的代码可能并不总是具有完全准确的数据类型，或者数组类型的形状正确。其次，反编译过程通常涉及推测和假设，导致反编译二进制文件时生成的控制流有所不同。上述因素共同作用于二进制文件，造成了 LLVM 中间表示的差异。

2) ***RQ2***：***GraphBinMatch*** 在单语言环境下应用于不同优化级别时，能否提供一致的结果？*:*

表 IV：单语言二元匹配任务的性能（阈值为 0.5）。

	精确率	召回率	F1
BinPro	0.38	0.42	0.40
B2SFinder	0.43	0.46	0.44
XLIR(LSTM)	0.67	0.72	0.44
XLIR（Transformer）	0.85	0.86	0.85
GraphBinMatch	0.88	0.86	0.87

我们可以自信地对这一研究问题回答“是”。在本研究中，由于相同语言的数据不足，我们使用了不同于 CLCDSA 的数据集。如表 I 所示，CLCDSA 最多仅提供 15589 个 C++源代码，而 POJ-105 数据集提供 37909 个源代码。替换数据集后，代码数量增加了 1.5 倍。POJ-105 也是基线论文中使用的数据集。表 IV 显示，GraphBinMatch 在以下方面优于基线论文：

表 V：来自不同优化级别的同语言二元匹配结果

	clang-10.0			gcc-9.4		
	精确率	召回率	F1	精确率	召回率	F1
O0	0.88	0.86	0.87	0.87	0.86	0.87
O1	0.87	0.88	0.88	0.89	0.85	0.85
O2	0.86	0.82	0.84	0.87	0.83	0.85
O3	0.86	0.83	0.85	0.84	0.81	0.83
Oz	0.90	0.85	0.87	0.87	0.87	0.87

精确率、召回率和 F1 分数分别为 0.88、0.86和 0.87，这表明 GraphBinMatch 更擅长检测相同语言的二进制源代码匹配。该表显示，基于 Transformer 的模型（XLIR）能够表现得相当出色，与相同语言的 LLVM中间表示一样，因为这些中间表示展现出更多相似性。尽管XLIR(Transformer)在单一语言上的得分表现良好，GraphBinMatch 仍然优于 XLIR(Transformer)。

为了验证 GraphBinMatch 在不同优化场景下的鲁棒性，我们使用 Clang 在同一数据集上评估了其性能，分别采用四种不同的优化级别（O0、O1、O2、O3、Oz）。表 V 显示，GraphBinMatch 在不同优化级别下均能保持稳定的性能。这表明模型的性能不依赖于特定的编译器或优化级别，能够在不同条件下提供可靠的结果。

通过考察 GraphBinMatch 在同一编译器的不同优化级别下的性能，我们可以对 RQ2 得出一些见解。我们认为，较高的优化级别会促使编译器进行更激进的优化，例如控制流调优。这些优化会为反编译器提供额外的假设和推测，从而增大源代码与二元可执行文件反编译得到的 LLVM 中间表示之间的差异。如表 V 所示，随着优化级别提高，用于衡量模型性能的精确率、召回率和 F1 分数逐渐下降。这表明，更激进的优化会对模型的代码匹配检测性能产生轻微影响。

*3)*研究问题***3***：***GraphBinMatch*** 在不同编译器下的性能如何？：

GraphBinMatch 在同一编译器的不同优化级别下表现稳定；在本小节中，我们将进一步探究它能否在不同编译器下也保持类似的性能。这对于实际应用而言是一个重要的考量，

因为生成二进制可执行文件所用的编译器可能未知。为了验证这一点，我们使用 Clang 从源代码生成 LLVM 中间表示，但与 RQ2 相比，使用 gcc 生成 POJ-104s 数据集的二进制可执行文件。我们采用与 RQ2 相同的设置，使用 RetDec 将二进制可执行文件转换为相应的 LLVM 中间表示。表 V 展示了使用 gcc 生成二进制可执行文件时 GraphBinMatch 的性能。结果表明，使用不同编译器时，GraphBinMatch 的表现基本相同。

我们推测，gcc 的性能指标优于 clang，是因为所使用的反编译器在编译由 gcc 生成的 C++ 代码时，往往会提供更多信息。我们的分析支持这一推测：由 clang 编译的二进制可执行文件所生成的 LLVM 中间表示平均大小约为 10,769.9 字节，而由 gcc 编译的二进制可执行文件所生成的 LLVM 中间表示平均大小约为 18,525.2 字节。这意味着在反编译阶段，gcc 编译的二进制文件大小约比 clang 编译的二进制文件大 70%。如此显著的大小差异可能会影响 GraphBinMatch 准确检测二元匹配的能力。

4) ***RQ4***：***GraphBinMatch*** 在源代码匹配方面表现如何？：

在源代码匹配任务中，我们使用 CLCDSA 数据集评估 GraphBinMatch 的性能，并将其与基线论文进行比较。实验设置与 RQ1 相同，不同之处在于我们使用 JLang 生成的 LLVM 中间表示作为 GraphBinMatch 的输入。我们测试了三种语言组合：C/C++ 与 Java、C 与 Java，以及 C++ 与 Java。表 VI 显示，在精确率、召回率和 F1 分数方面，GraphBinMatch 的表现比基线论文高出约 20%。GraphBinMatch 还能有效检测不同编程语言之间的源代码匹配。

A. 变化的阈值

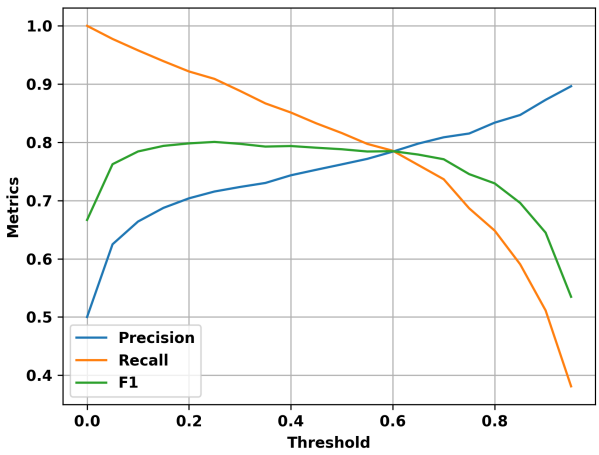


图3： [越高越好]改变阈值会导致精确率、召回率和 F1 得分不同。

在上一小节中，我们提到GraphBinMatch的阈值设为0.5。我们通过改变阈值进行实验，以研究不同阈值对精确率、召回率和F1 分数的影响。图3展示了通过改变阈值可以达到的不同得分。根据我们的研究结果，阈值为0.2时可以获得略高的 F1分数。然而，我们还观察到，使用该阈值会使准确率显著下降，最低仅为 7%，因此为了获得最佳 F1分数而寻找最佳阈值并不实际。最终，我们选择使用 0.5作为阈值，因为我们认为对于本研究而言，这是一个更合理的默认阈值。较小的阈值会带来更高的召回率，因为GraphBinMatch 会将所有对预测为匹配对。另一方面，较大的阈值会带来更高的精确率，因为 GraphBinMatch 会将所有对预测为不匹配对。根据具体使用场景，GraphBinMatch 的用户可以手动设置阈值，也可以让GraphBinMatch 根据给定的指标（即精确率、召回率、F1）决定最佳阈值。

VI. 讨论

本节讨论一些可能影响 GraphBinMatch 性能的挑战。

A. 调查 *GraphBinMatch* 可能失败的原因

我们的实验发现，模型偶尔会错误识别匹配的代码片段对。逐一检查预测错误的样本后，我们发现，大多数假阳性都是由于 LLVM-IR 片段的大小差异过大造成的。我们计算并分析了测试集图中的节点数量，结果如表 VII 所示。表中显示，对于假阳性集，模型所预测节点数量的均值和中位数之差远大于真阳性集。两组之间节点数量的中位数差异接近 50%。

经过深入分析，我们发现这些假阳性的两个主要原因。第一个原因是 LLVM 中间表示之间的差异大于GraphBinMatch的容忍范围。例如，一些代码片段可能使用标准库提供的排序方法，而另一些代码片段可能在内部实现自己的排序方法。GraphBinMatch 可能无法有效识别，标准库的方法调用与作者在代码中自行实现的排序方法是等价的。此外，C++ 中的模板机制也会影响 GraphBinMatch，因为许多 C++ 标准库以模板形式发布。这意味着模板也会作为 LLVM 中间表示的一部分进行编译，从而导致 GraphBinMatch难以识别，编译后的模板代码与 Java 中的标准库调用是等价的。

第二种情况涉及 Java 和 C++ 之间的语言差异，以及各自用户群体不同的使用习惯。即使两者都经过转换并编译为 LLVM 中间表示，LLVM 中间表示也可能存在显著差异。这些差异源于不同的.....

表 VI：跨语言源代码匹配结果

	GraphBinMatch			XLIR（LSTM）			XLIR（Transformer）		
	精确率	召回率	F1	精确率	召回率	F1	精确率	召回率	F1
C 与 Java	**0.77**	**0.80**	**0.78**	0.62	0.51	0.56	0.75	0.55	0.63
C++ 对比 Java	0.76	**0.82**	**0.79**	0.65	0.53	0.58	**0.77**	0.57	0.66
C/C++ 对比 Java	0.81	0.73	0.78	-	-	-	-	-	-

表 VII：测试集中的节点数量统计

Type	Mean	中位数
真正例	1506	864
误报	2133	1303
真负例	2573	1680
假阴性	2293	1289

不同编程语言用户的习惯。例如，图 4 展示了一对匹配项，尽管它们具有相似性，但生成的中间表示图规模却大相径庭：Java 生成的中间表示图包含 330 个节点和 660 条边，而 C++ 生成的中间表示图仅包含 65 个节点和 115 条边。这种差异可能会给 GraphBinMatch 识别匹配项带来挑战，因为该模型可能未考虑语言使用习惯对中间表示生成过程的影响，从而导致错误结果。

B. 将 *GraphBinMatch* 扩展到其他编程语言

本研究提出了一种利用图神经网络（GNNs）检测跨语言二进制代码匹配的新方法。我们方法的一个关键组成部分是使用 LLVM IR 衡量以不同编程语言编写的两个程序之间的相似性。借助 LLVM IR，我们能够捕获代码的高层信息，而不受底层编程语言的影响。

尽管我们的方法旨在实现跨语言支持，但它依赖于编译器前端的可用性。要将我们的方法扩展为支持其他编程语言并生成 LLVM IR，必须使用相应的编译器前端。一旦生成中间表示，我们的方法就保持不变。我们的方法足够灵活，可以支持任何编译为 LLVM IR 的编程语言。

C. 分词器和不同的词元嵌入如何影响结果

表VIII：用于同语言二元匹配和跨语言二元匹配的不同 LLVM 中间表示嵌入技术的性能

	Cpp 与 Cpp			Cpp/C 与 Java		
	精确率	召回率	F1	精确率	召回率	F1
text	0.86	0.83	0.85	0.75	0.73	0.74
全文	0.89	0.87	0.88	0.84	0.75	0.79

在本节中，我们使用 CLCDSA 数据集开展实验研究，采用与 RQ1 相同的设置和预处理方法。我们的目标是考察各种词元嵌入技术和分词器对 GraphBinMatch 性能的影响。具体而言，我们旨在探究不同嵌入方法和分词方案对 GraphBinMatch 跨多种编程语言或同种语言匹配二元与源代码对的能力有何影响。通过考察这些因素，我们希望更好地了解如何优化 GraphBinMatch 的性能，并提升其在不同语言对中检测代码克隆的能力。

首先，我们旨在评估使用完整_文本属性与完整_文本相比，对源-二元匹配任务的影响。在论文的方法部分，我们提到，使用ProGraML提供的完整_文本属性，相较于仅使用完整_文本，可能带来更好的性能。如表 VIII所示，我们观察到，使用完整_文本属性能够提升跨语言二进制源代码匹配任务和同语言二进制源代码匹配任务的性能。值得注意的是，对于跨语言二进制源代码任务，使用完整_文本属性所带来的提升，比使用以相同语言编写的二进制源代码匹配任务更为显著。我们认为，这一结果是因为向模型暴露更多信息后，我们能够更好地弥合模型在跨语言任务上的理解差距。

七、相关工作

随着研究工具和机器学习模型的出现，代码相似性检测近年来受到广泛关注。这些进展为该领域的研究带来了新的机遇。在代码相似性检测中，源代码的相似性通常分为四个级别，如 [13] 所述：

- **I型**：这也称为精确克隆。源代码可以完全相同，仅修改缩进、注释和代码布局。
- **II型**：这也称为参数化克隆。除TypeI中提到的修改外，源代码的结构和语法也基本相似，仅对部分变量名、方法名和数据类型进行了修改。
- **III型**：与II型相比，III型代码克隆涉及语句修改，但仍保持功能相似性。
- **IV型**：两个代码片段在语法和结构上存在差异，但对于相同输入，两个代码片段执行相似的行为。

代码相似性检测研究大致分为两个方向：基于算法的方法和基于机器学习的方法。下面将详细介绍这两个不同的方向：

```java
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        String s = sc.next();
        int ans = 0;
        for (char c : s.toCharArray()) {
            ans += Integer.parseInt(""+c);
        }

        System.out.println(ans);
    }
}
```

(a) Java 示例

```java
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        String s = sc.next();
        int ans = 0;
        for (char c : s.toCharArray()) {
            ans += Integer.parseInt(""+c);
        }

        System.out.println(ans);
    }
}
```

(b) 来自 c++ 的样本

图4：假阴性案例示例

A. 基于算法的方法

基于算法的代码克隆检测依赖词法或语义分析，因此大多数方法无法有效检测IV型代码克隆；当III型代码克隆的差异过大或过于频繁时，检测效果也不理想。大多数算法仅适用于I型、II型以及部分III型代码克隆。此外，这些算法在跨语言代码克隆检测方面的效果往往不尽如人意。

CCFinderSW [22] 和 SourcerCC[23] 等研究项目采用基于词元的方法检测代码克隆。这些工具各自使用自己的词法分析器实现，将源代码转换为词元流，并以此作为分析基础，再通过不同算法测量相似度分数。然而，由于缺乏词法信息抽象，这些算法难以获取和理解其内在语义。

基于语义分析的代码克隆检测工具试图解决这一问题。DECKARD [24] 提出了一种基于树的代码克隆检测方案：它创新性地使用特征向量来描述语法树的结构，并根据特征向量之间的欧几里得距离对克隆进行聚类。

传统的基于算法的代码克隆检测方法通常难以迁移到跨语言克隆检测中，因此针对这一问题的研究非常少。即使相关研究提出了相应的解决方案，也受到各种限制，无法满足现实世界的需求。其中，最具代表性的两项研究是 LICCA [25]和CLCMiner [26]。LICCA通过尝试将不同语言转换为统一的表达形式来检测代码克隆，但只能涵盖两段代码具有相似结构和语法元素的情况。CLCMiner 采用 NLP 方法查看源代码，但依赖于源代码的修改记录。

B. 机器学习方法

随着机器学习和图神经网络的发展，出现了许多解决代码克隆检测问题的新思路。随着各种代码克隆数据集的推出，使用机器学习解决此类问题已成为可能。例如，CLCDSA [17] 和 BigCloneBench [27]，都包含具有 IV 型相似性的代码克隆。近年来，许多研究尝试用机器学习模型取代传统算法，以提高准确率，并获得跨不同语言进行比较的能力。

CLDH [28]首先提出了一种哈希方法，用于检测同一语言中的两段代码是否为克隆代码。其模型接受原始代码片段作为输入，使用特定规则对抽象语法树进行编码，并利用 LSTM 网络将其转换为源代码的向量表示。然而，他们的编码方法更加关注 AST 的结构，而没有考虑节点类型等其他语义信息。

除了基于树的方法外，CCLearner[29] 和 C4 [30] 等方法也使用基于词元的代码克隆检测：CCLearner 是最早将词元作为代码克隆检测模型输入的研究之一。它将词元分为八类，分别计算每一类的相似度分数，然后利用向量的相似度计算代码片段之间的相似度。该研究得出的结论是，特征向量越相似，代码对属于代码克隆的概率就越高。该研究只捕获了词元和部分语法层面的信息，却没有捕获代码的结构。论文指出，正因如此，CCLearner 对 IV型/III型克隆的检测效果不佳。

另一项基于词元的研究是 C4 [30]。他们的研究使用一个名为 codebert [31] 的预训练 BERT 模型对模型的输入进行嵌入。由于 CodeBERT 支持对多种语言进行编码，C4 也支持跨语言代码克隆检测。他们的研究以原始源代码作为输入，首先使用 CodeBERT 对其进行编码以获得特征向量，然后由模型学习并分类该特征向量，得到最终的相似度分数。他们还采用了对比学习，以提高数据集的利用率。然而，由于 CodeBERT 的限制，C4 仅支持编码前 512

个词元，这影响了 C4 的最终适用性。

当前学术研究主要聚焦于源代码克隆检测，但 Gui *et al.* 提出的 XLIR 重新将关注点转向跨语言二进制源代码匹配问题。软件应用通常使用不同的编程语言编写，以满足各种需求并适配不同的计算平台。因此，在多种编程语言中检测二进制代码与源代码的克隆，在实际场景中大有裨益。例如，当检测到存在漏洞的二进制代码时，就有必要检索所有可能使用的编程语言编写的相关源代码片段，以便更好地评估漏洞。我们以 XLIR 作为基线论文，其方法和性能的更多细节将在第五节中介绍。

VIII. 结论与未来工作

本文介绍了GraphBinMatch，这是一种新颖的跨语言二进制源代码匹配模型，利用 LLVM 中间表示和异构图来学习相似性。该模型以LLVM IR作为输入，可以处理三类代码匹配检测任务：跨语言二进制匹配检测、单语言二进制匹配检测以及跨语言源代码匹配检测。需要注意的是，GraphBinMatch 并不局限于这些特定语言。它可以支持任何具有编译器前端、能够生成 LLVM IR 的编程语言。

实验结果表明，GraphBinMatch 有效，且优于所有三种最先进的工具。具体而言，我们的方法在精确率、召回率和 F1 分数方面均优于这些工具，这表明它在识别跨语言代码匹配方面表现出色。

未来，我们计划将 GraphBinMatch 扩展到更多编程语言，并集成更多编译器前端，为这些语言生成 LLVM IR。此外，由于我们的方法由数据驱动，我们正在收集更多源代码文件，以扩大数据集规模；我们相信这将提升 GraphBinMatch 的预测能力。通过实现这些改进，我们相信 GraphBinMatch 将成为更强大的代码匹配检测工具，并帮助软件开发人员维护和改进其代码库。

参考文献

[1] R. T. Yarlagadda，“通过二元分析实现计算机安全的方法，”工程研究与技术创新国际期刊 *[IJIERT]*，2020。[2] H. Yang、M. Fritzsche、C. Bartz 和 C. Meinel，“Bmxnet：一种基于 mxnet 的开源二元神经网络实现，”载于第 *25* 届 *ACM* 多媒体国际会议论文集，2017 年，第 1209–1212 页。[3] D. Miyani、Z. Huang 和 D. Lie，“Binpro：一种用于二元源代码溯源的工具，”*arXiv* 预印本 *arXiv:1711.00830*，2017 年。[4] A. Shahkar，“关于二元与源代码的匹配，”博士论文，康考迪亚大学，2016 年。[5] G. Zhao 和 J. Huang，“Deepsim：代码功能相似性的深度学习，”载于*2018* 年第 *26* 届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集，2018 年，第 141–151 页。

[6] Z. Yu、R. Cao、Q. Tang、S. Nie、J. Huang 和 S. Wu，“顺序很重要：用于二进制代码相似性检测的语义感知神经网络”，载于《AAAI 人工智能会议论文集》，第 34 卷，第 01 期，2020 年，第 1145–1152 页。[7] Z. Yuan、M. Feng、F. Li、G. Ban、Y. Xiao、S. Wang、Q. Tang、H. Su、C. Yu、J. Xu 等，“B2sfinder：检测商业现货软件中的开源软件复用”，载于《2019 年第 34 届 IEEE/ACM 自动化软件工程国际会议（ASE）》。IEEE，2019 年，第 1038–1049 页。[8] Y. Gui、Y. Wan、H. Zhang、H. Huang、Y. Sui、G. Xu、Z. Shao 和 H. Jin，“利用中间表示进行跨语言二进制代码与源代码匹配”，载于《2022 年 IEEE 软件分析、演化与再工程国际会议（SANER）》。美国加利福尼亚州洛斯阿拉米托斯：IEEE 计算机学会，2022 年 3 月，第 601–612 页。 [在线]。网址：https://doi.ieeecomputersociety.org/10.1109/ SANER53432.2022.00077 [9] T. Ben-Nun、A. S. Jakobovits 和 T. Hoefler，“神经代码理解：代码语义的可学习表示”，《神经信息处理系统进展》，第 31 卷，2018 年。[10] S. VenkataKeerthy、R. Aggarwal、S. Jain、M. S. Desarkar、R. Upadrasta 和 Y. Srikant，“Ir2vec：基于 LLVM IR 的可扩展程序嵌入”，《ACM 架构与代码优化汇刊（TACO）》，第 17 卷，第 4 期，第 1–27 页，2020 年。[11] Y. Sui、X. Cheng、G. Zhang 和 H. Wang，“Flow2vec：基于值流的精确代码嵌入”，《ACM 编程语言论文集》，第 4 卷，OOPSLA 期，第 1–27 页，2020 年。[12] M. Allamanis，“程序分析中的图神经网络”，载于《图神经网络：基础、前沿与应用》。Springer，2022 年，第 483–497 页。[13] M. Allamanis、M. Brockschmidt 和 M. Khademi，“学习用图表示程序”，arXiv 预印本 arXiv:1711.00740，2017 年。[14] C. Cummins、Z. V. Fisches、T. Ben-Nun、T. Hoefler 和 H. Leather，“Programl：用于程序优化与分析的基于图的深度学习”，arXiv 预印本 arXiv:2003.10536，2020 年。[15] Y. Bai、H. Ding、S. Bian、T. Chen、Y. Sun 和 W. Wang，“Simgnn：一种快速计算图相似度的神经网络方法”，载于《第十二届 ACM 网络搜索与数据挖掘国际会议论文集》，2019 年，第 384–392 页。[16] S. Brody、U. Alon 和 E. Yahav，“图注意力网络的注意力程度如何？”，载于《国际学习表征会议》，2022 年。

[ ] = 72 7 1 在线。[17] 网址：https://openreview.net/forum?id F ximsx C K. W. Nafi、T. S. Kar、B. Roy、C. K. Roy 和 K. A. Schneider，“Clcdsa：利用句法特征和 API 文档进行跨语言代码克隆检测”，载于《2019 年第 34 届 IEEE/ACM 自动化软件工程国际会议（ASE）》。IEEE，2019 年，第 1026–1037 页。[18] L. Mou、G. Li、L. Zhang、T. Wang 和 Z. Jin，“面向编程语言处理的树结构卷积神经网络”，载于《AAAI 人工智能会议论文集》，第 30 卷，第 1 期，2016 年。[19] K. Sinha、R. Jia、D. Hupkes、J. Pineau、A. Williams 和 D. Kiela，“掩码语言建模与分布假设：预训练时词序的重要性”，arXiv 预印本 arXiv:2104 06644，2021 年。[20] T. Vislavski、G. Raki´c、N. Cardozo 和 Z. Budimac，“Licca：一种跨语言克隆检测工具”，载于《2018 年 IEEE 第 25 届软件分析、演化与再工程国际会议（SANER）》。IEEE，2018 年，第 512–516 页。[21] D. P. Kingma 和 J. Ba，“Adam：一种随机优化方法”，arXiv 预印本 arXiv:1412.6980，2014 年。[22] Y. Semura、N. Yoshida、E. Choi 和 K. Inoue，“Ccfindersw：一种具有灵活多语言分词功能的克隆检测工具”，载于《2017 年第 24 届亚太软件工程会议（APSEC）》，2017 年，第 654–659 页。[23] H. Sajnani、V. Saini、J. Svajlenko、C. K. Roy 和 C. V. Lopes，“Sourcerercc：将代码克隆检测扩展至大型代码库”，载于《第 38 届软件工程国际会议论文集》，ICSE ’16 系列。美国纽约：计算机协会，2016 年，第 1157–1168 页。 [在线]。网址：https://doi.org/ 10 1145 2884781 2884877 。/。L. Jiang、[24] G. Misherghi、Z. Su 和 S. Glondu，“Deckard：可扩展且准确的基于树的代码克隆检测”，载于《第 29 届软件工程国际会议（ICSE’07）》，2007 年，第 96–105 页。[25] T. Vislavski、G. Raki´c、N. Cardozo 和 Z. Budimac，“Licca：一种跨语言克隆检测工具”，载于《2018 年 IEEE 第 25 届软件分析、演化与再工程国际会议（SANER）》，2018 年，第 512–516 页。

[26] X. Cheng、Z. Peng、L. Jiang、H. Zhong、H. Yu 和 J. Zhao，“Clcminer：无需中间表示即可检测跨语言克隆”，《*IEICE* 信息与系统汇刊》，第 100 卷，284，2017 2，273 第 – 页。[27] J. Svajlenko 和 C. K. Roy，“Bigclonebench”，载于《代码克隆分析》。Springer，2021 年，第 93–105 页。[28] H. Wei 和 M. Li，“通过利用源代码中的词汇和句法信息，采用监督式深层特征进行软件功能克隆检测”，载于《第二十六届国际人工智能联合会议论文集（*IJCAI-17*）》，2017 年，第 3034–3040 页。[在线]。网址：https://doi.org/10.24963/ijcai.2017/423

[29] L. Li、H. Feng、W. Zhuang、N. Meng 和 B. Ryder，“Cclearner：一种基于深度学习的克隆检测方法”，载于 *2017* 年 *IEEE* 软件维护与演化国际会议（*ICSME*）。IEEE，2017 年，第 249–260 页。[30] C. Tao、Q. Zhan、X. Hu 和 X. Xia，“C4：对比式跨语言代码克隆检测”，载于 *2022* 年 *IEEE/ACM* 第 *30* 届程序理解国际会议（*ICPC*），2022 年，第 413–424 页。[31] Z. Feng、D. Guo、D. Tang、N. Duan、X. Feng、M. Gong、L. Shou、B. Qin、T. Liu、D. Jiang 等，“CodeBERT：一种面向编程语言和自然语言的预训练模型”，*arXiv* 预印本 *arXiv:2002.08155*，2020 年。