

基于指令关系图的二进制代码与源代码相似性分析方法

李庚, 金大海, 宫云战

(北京邮电大学 网络与交换技术全国重点实验室, 北京市 100876)

摘要: 可执行二进制程序的分析和逆向工程在计算机安全等各个领域具有广泛的应用。以二进制形式进行程序的逆向工程通常被视为一种主要手动和耗时的过程, 难以高效地应用于大规模样本集。然而, 安全公司通常需要每天分析数千个未知的二进制文件, 急需快速和自动化的二进制分析和逆向工程方法。故二进制代码与源代码的相似性分析技术在二进制程序的分析和逆向工程中扮演着重要的角色。目前的方法在提取代码语义方面存在不足, 有些方法忽略了部分语义信息, 有些方法则包含了冗余的信息。因此, 本文提出了一种基于指令关系图的二进制代码和源代码相似性分析方法。该方法从二进制代码和源代码转换为的 LLVM-IR 中生成了包含了指令之间的控制流、数据流和调用流关系的指令关系图。此外, 本文还提出了一种基于图神经网络的代码相似性分析模型, 通过指令关系图和其粗化后生成全局控制流图提取代码的语义特征, 并计算相似度。本文方法在开源数据集 CodeNet 上进行了评估, 并针对不同编译选项下的二进制代码进行了性能评估。实验结果表明, 本文方法在取得了先进的性能。

关键词: 逆向工程、克隆检测、中间表示、二进制代码、代码匹配、深度学习

中图分类号: TP311

Binary-Source Code Similarity Comparison

Li Geng, JIN Dahai, GONG Yunzhan

(State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunication, Beijing 100876, China)

Abstract: The analysis and reverse engineering of executable binary programs have extensive applications in various fields, including computer security. Reverse engineering programs in binary form are often considered a manual and time-consuming process, making it challenging to efficiently apply them to large sets of samples. However, security companies often need to analyze thousands of unknown binary files daily, necessitating fast and automated methods for binary analysis and reverse engineering. Therefore, the similarity analysis of binary code to source code plays a crucial role in the analysis and reverse engineering of binary programs. Existing methods have limitations in extracting code semantics, as some methods overlook certain semantic information while others include redundant information. Hence, this paper proposes a binary-to-source code similarity analysis method based on LLVM. This method generates an instruction relationship graph from LLVM-IR, obtained from both binary and source code, capturing the control flow, data flow, and call flow relationships between instructions. Additionally, a code similarity analysis model based on graph neural networks is introduced, which extracts semantic features from the code using the instruction relationship graph and a globally pooled control flow graph. The proposed method is evaluated on the open-source dataset CodeNet and its performance is assessed for different compilation options of binary code. Experimental results demonstrate that the proposed method outperforms existing state-of-the-art methods. **Keywords:** reverse engineering, clone detection, intermediate representation, binary code, code matching, deep learning.

Key words: reverse engineering, clone detection, intermediate representation, binary code, code matching, deep learning.

作者简介: 李庚 (出生年 1997), 男, 硕士研究生, 主要研究方向: 图神经网络、缺陷检测

通信联系人: 金大海, 男, 1974 年生, 博士, 副教授, CCF 会员, 主要研究方向: 软件测试、缺陷检测. E-mail: jindh@bupt.edu.cn

0 引言

可执行二进制程序的分析和逆向工程在计算机安全等各个领域具有广泛的应用。逆向工程的常见安全应用包括分析潜在的恶意软件或检查商业软件。以二进制形式进行程序的逆向工程通常被视为一种主要手动和耗时的过程，难以高效地应用于大规模样本集。然而，安全公司通常需要每天分析数千个未知的二进制文件，急需快速和自动化的二进制分析和逆向工程方法。同时代码重用是各种计算机程序编码中非常常见的做法，包括自由软件、商业软件以及恶意软件，这些都是逆向工程的典型目标。因此在二进制程序逆向工程中，通常希望能够快速识别重用的代码片段。可靠地检测到克隆代码可以让逆向工程师节省时间，跳过已知功能的片段，专注于驱动程序的主要功能部分。故二进制代码与源代码的相似性分析技术在二进制程序的分析和逆向工程中扮演着重要的角色。通过给定二进制代码，找到与之匹配的源代码可以提高逆向工程的效率。同样，给定源代码找到相匹配的二进制代码，可以检测二进制代码中可能存在的漏洞。

1 相关工作

1.1 二进制与源代码匹配

OSSPolice^[1]将源代码中的字符和函数作为特征，并使用软件相似性比较算法来检测二进制代码的重用。B2SFinder^[2]推断出大约 7 种在二进制代码和源代码中都可追溯的代码特性，并结合三种加权特征匹配算法，匹配分数和代码结构来识别不同类型的代码重用。RESource^[3]灵感来自 Re-Google，利用在源文件级别和汇编文件级别上都存在的一些字符特性，基于这些特性触发匹配查询。

以上研究主要进行工程级的相似性分析，分析粒度过大，并基于代码文本信息进行提取代码特征，但代码具有复杂的结构，因此分析效果不尽如人意。

近年来由于深度学习在自然语言处理和图像处理等领域取得的优秀成果，研究者们也将深度学习技术应用在程序语言的相关任务中，随着技术的不断发展，检测的效果在不断提升，基于机械学习的二进制代码与源代码相似性分析研究也不断涌现。

BinPro^[4]第一个使用机器学习和静态分析的技术组合来匹配程序源代码和二进制文件，BinPro 利用机器学习来计算最优代码特征，并利用一个静态分析方法来提取和计算二进制和源代码的相似度。Bin2Source^[5]提出了一种源码和二进制码指令对指令匹配的新方法，通过提取二进制代码的体系结构，并将源代码转换为目标体系结构的二进制文件，将问题转换为两个二进制文件的比较。CodeCMR^[6]研究了函数级二进制代码和源代码的跨模态匹配，使用两种方法提取了源代码和二进制代码的语义特征，并提出了两个编码器网络将代码表示为两个隐藏向量，然后设计了一个相似性损失函数来共同学习代码特征。

BugGraph^[7]通过两个步骤计算源二进制代码相似度，即通过规范化消除二进制代码的架构、编译器、编译器版本、优化级别带来的差异，再从代码中提取属性控制流图

(ACFG)，将问题转化为图相似度计算，最后利用一个图神经网络计算图的相似度。

XLIR^[8]将来自不同编程语言的二进制代码和源代码解析为中间语言 LLVM-IR。再利用基于 BERT 的模型将 LLVM-IRs 映射到一个共同的特征空间中，并共同学习它们之间的相关性。

1.2 代码表示学习

代码表示学习，也称为代码嵌入，旨在将程序的语义保留为分布式向量。本文的任务分析二进制代码和源代码的语义相似性，故学习代码语义的表示方法至关重要。

基于特征的方法需要专家从程序中手动提取特征来表示应用于不同任务的程序语义。例如，FLUCSS^[9]面向代码故障定位，结合了基于频谱的故障定位度量和源代码相关的度量来识别软件中的故障。

基于序列的表示将代码视为 Token，并将它们转换为向量，并进一步将这些向量用于不同的任务。例如 CodeBERT^[10]是一种基于 BERT 的针对编程语言 (PL) 和自然语言 (NL) 的预训练模型，它将代码分解为 Token 输入到模型中学习代码语义。IR2Vec^[11]提出一种新颖的基于中间语言 LLVM-IR 的框架，并基于 TransE^[12]模型提出了基于语法和流信息的编码方式，可以以任务无关的方式捕获输入程序的隐含特征。GraphCodeBERT^[13]基于 CodeBERT 模型，除了捕获语法特征之外，还在预训练阶段增加了代码数据流信息来解释语义。基于 LLVM-IR 的 OSCAR^[14]受操作语义的启发，在学习代码语义时考虑了程序抽象环境信息，并引入了位置编码 PCE 将控制流信息合并到模型中，同时在预训练期间生成具有不同优化等级的代码语法提升模型效果。

与顺序数据相比，程序是一个高度结构化的数据，因此许多与代码相关的工作试图提取代码背后的结构语义，一般通过抽象语法树(AST)、控制流图、数据流图等结构化表示来捕获语义。例如 ASTNN^[15]将代码的 AST 分割成许多小的语法树并编码为向量，并使用双向 RNN 模型来学习代码表示。Code2Vec^[16]是一个基于 AST 的代码表示学习模型，它将代码片段表示为单个固定长度的代码向量，使用 AST 将程序分解为一组路径，并学习每个路径的表示。inst2vec^[17]使用从 LLVM-IR 语句构建的上下文流图(XFG)作为神经网络的输入。XFG 结合部分数据和控制流来表示单个语句的上下文，然后使用图中的邻居将这些语句映射到潜在空间表示。ProGraML^[18]是一种基于 LLVM-IR 的全程序语义表示，它提取的 LLVM-IR 代码中的各种指令之间的流信息，包含数据流、控制流和调用流，并采用消息传递神经网络来学习图的特征来表示代码特征。

1.3 本文工作

针对上述现有二进制代码和源代码相似性分析方法的不足，本文提出基于 LLVM-IR 的二进制代码和源代码相似性分析框架。本文首先提出一种用于表征代码的指令间控制流、数据流和调用关系的指令关系图及其生成算法。与现有的研究不同，本文根据 LLVM-IR 指令之间的控制流关系、数据流关系和彼此之间的调用关系构建了包含多种类型边的指令关系图 IRG。其中图中节点包含了代码指令信息。节点之间的控制流信息、数据流信息

和调用信息将被视作不同类型边被显式地保留在图中。其次，针对 LLVM 指令结构提出了指令表示学习的方法。该方法根据指令结构分别学习指令各个部分，再将各个部分组合表示整条指令。

最后，本文面向指令关系图建立代码语义相似性分析模型。首先通过图神经网络学习指令关系图中指令之间不同类型的关系，同时本文根据不同的关系提出了不同的聚合算法以学习指令表示。再通过图粗化算法将代码关系图中的节点合并转化为跨函数的控制流图，从而学习代码特征，计算图之间相似度。

本文通过对开源代码数据集中相同语义的不同代码进行相似度分析和对比试验，验证了本文方法的有效性。

2 基于指令关系图的代码相似性分析模型框架

本文的目标是建立基于指令关系图的二进制代码和源代码相似性分析模型。本文方法的整体设计包括代码编译和反编译、代码指令嵌入、代码表示图生成、图相似度分析神经网络设计。本文方案的整体设计如图 1 所示。

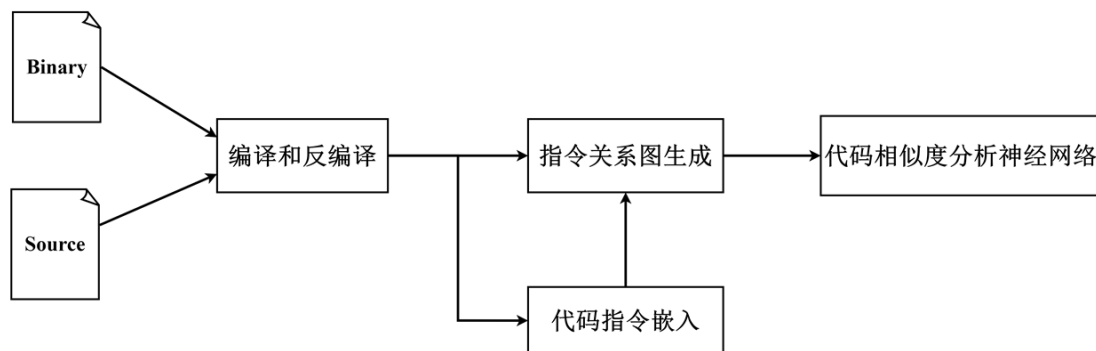


图 1 方案的整体设计

Fig.1 Overall design drawing

对于输入的二进制文件和源代码，首先通过编译工具和反编译工具将输入转换为中间语言表示，根据中间语言生成指令关系图，并通过代码指令嵌入技术将代码指令图中的节点转化为嵌入表示，随后将包含指令嵌入表示的指令关系图输入到代码相似度分析神经网络中，计算出代码相似度。

3 代码编译和反编译

中间表示(IR)是一种语义表达明确且具有良好格式的程序表示，语法规则通常简单。现代编译器首先解析源代码，将其转换为中间表示，然后由中间表示生成目标代码。因此中间表示具有跨平台和目标无关的特性，即中间表示是与特定的源代码语言和目标架构无关的。

本文采用中间表示 LLVM-IR^[19]进行代码语义的中间表示，通过将二进制代码和源代码反编译和编译为 LLVM-IR，并学习其中的代码语义。LLVM-IR 有两种表示形式：一种是

可读的汇编语言形式，以.ll 文件存储；另一种是序列化之后的 bitcode 形式，以.bc 文件存储。本文主要对具有可读性的.ll 文件进行分析。编译后的 LLVM-IR 代码具有静态单赋值形式（SSA，static single assignment form），即每一个变量仅能被赋值一次。相较于高级编程语言，一方面，LLVM-IR 代码的 SSA 形式和变量标识符替换有利于后续代码语义特征提取；另一方面，使用 LLVM-IR 可以消除不同类型高级编程语言的语法差异，并且能够兼容不同类型的高级编程语言。LLVM-IR 代码示例如图 2 所示。

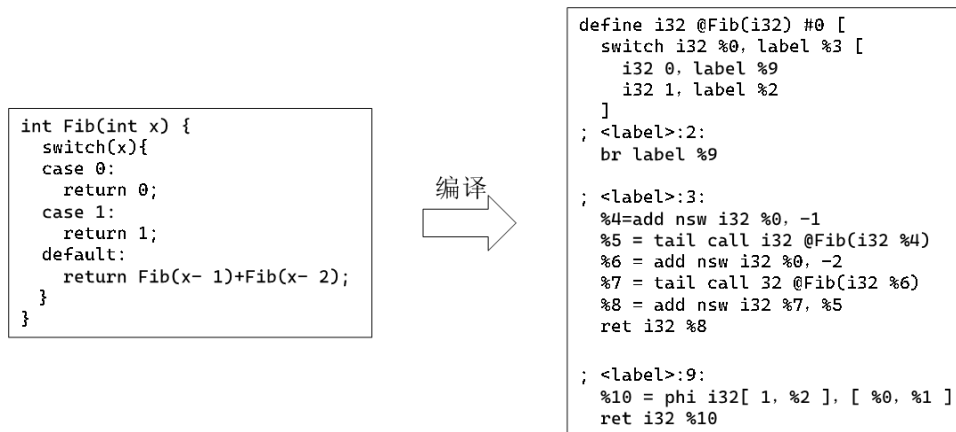


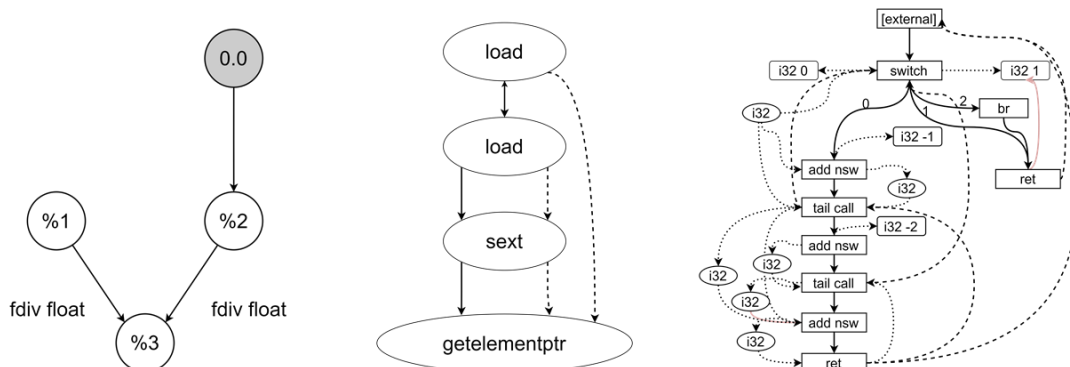
图 2 LLVM-IR 代码示例

Fig.2 LLVM-IR Example

本文通过使用 LLVM 官方的工具 Clang 将 C/C++源代码编译成 LLVM-IR，同时使用 Retdec^[20]将二进制文件反编译为 LLVM-IR。Clang 是一个支持 C、Objective-C、C++和 Objective-C++语言的开源编译器。Retdec 是一个基于 LLVM 的机器代码反编译器，反编译器不限于任何特定的目标体系结构、操作系统或可执行文件格式。本文通过 Clang 和 Retdec 可以将源代码和二进制代码统一转换为 LLVM-IR。

4 指令关系图

基于图结构的代码学习已被用于许多任务中，例如代码摘要、语义代码检索和代码克隆检测。同时 Brauckmann^[21]等人 and Cummins^[22]等人的工作清楚地表明基于图的代码表示方法优于基于顺序的代码表示方法。目前许多图结构与程序代码相关，传统的基于 LLVM 的代码表示图主要有控制流图(CFG)、控制依赖图(CDG)、数据流图(DFG)等。



(a)XFG

(b)CDFG

(c)ProGraML

图 3 现有基于 LLVM 的代码表示图

Fig.3 Existing LLVM-based code representation graph

为了更好的学习代码表示, 目前有许多研究提出了新的代码表示图。如上下文流图 (XFG) 结合部分指令数据流和控制流来表示单个语句的上下文, 如图 3 (a) 所示。然而, 部分组合 DFG 和 CFG 的 XFG 表示省略了重要信息, 例如指令操作的顺序, 因此 XFG 无法捕获代码的执行顺序, 这对于许多优化任务至关重要。最近的 LLVM-IR 表示使用控制流图(CFG)进行代码表示, 如图 3 (b) 所示。这种表示使指令之间的控制流和数据关系明确, 但仅使用指令操作码来计算潜在表示, 但省略了至关重要的程序的信息, 例如数据类型、变量和常量的存在以及操作数的顺序。ProGraML 则是指令调用图、控制流图和数据流图的并集, 如图 3 (c) 所示。ProGraML 将程序表示为有向多图, 其中指令语句、标识符和常量是顶点, 顶点之间的关系是边。节点之间的边分为控制流、数据流和调用流。此外, 使用位置标签来增强边来编码语句操作数的顺序, 并区分控制流中的不同分支。ProGraML 中包含了大量的信息, 并将数据节点拆分为几种类型的, 直觉上应该有很好的效果。然而如[23]中的研究所示, 一个好的图结构数据应是紧凑的, 并且多种的数据节点类型对于代码的语义学习并不能带来良好的效果。

为了解决以上这些问题, 本文提出了指令关系图 IRG, 旨在构建包含了指令之间控制流关系、数据流关系和调用关系的代码表示图充分高效的表示代码指令间的复杂关系, 在充分表达代码信息的同时尽量紧凑, 从而学习其中的代码语义。

4.1 指令关系图定义

对于指令关系图 $G = (V, E_C, E_D, E_{Ca})$, V 为指令节点的集合, E_C 为控制流边的集合, E_D 为数据流边的集合, E_{Ca} 为调用流边的集合。对于每个节点 $i \in V$, 均包含一条完整指令信息。指令关系图中包含的三种类型的边: 控制流、数据流和调用流。图 4 展示了图 3 中 LLVM-IR 示例代码生成的指令关系图。

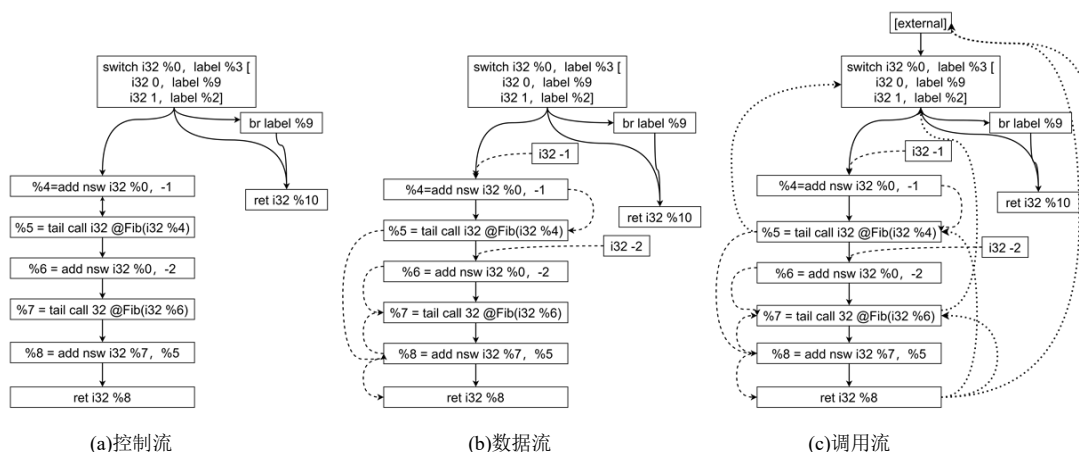


图 4 代码关系图

Fig.4 Instruction Relation Graph

4.1.1 控制流边

190 本文通过遍历为每条指令创建节点和节点之间的控制流边来构建一个指令间的全局控制流图，如图 2（a）所示。控制流边捕获了程序中指令的执行顺序信息。对于具有单个控制后继指令的指令，有一条控制流出边。对于具有 n 个后继指令的分支指令，具有 n 条控制流出边。

4.1.2 数据流边

195 本文同样通过遍历指令的方式在图中添加数据流边以捕获指令之间的变量流向的关系。控制流边捕获了整个程序的数据流向信息。与ProGraML对代码指令中具有的常量值单独创建节点不同，本文并不对其创建节点而是在获取指令嵌入时引入其中。而对于全局变量则单独创建节点。如图 2（b）所示。

4.1.3 调用流边

200 调用流边捕获被调用函数和调用函数的指令之间的关系，旨在捕获从调用指令和被调用函数节点之间的调用关系，如图 3（c）所示。图中将返回调用边从函数的所有终端语句添加到调用语句中。如果指令调用当前LLVM-IR中未定义的函数，则创建一个虚拟函数节点，并且与调用指令节点通过一对调用流边连接。与控制流边不跨越函数不同，数据流边和调用流边可以跨函数，如在程序中的多个函数中使用的全局常数的情况，或是调用指令调用其他函数。
205 数。

4.2 指令关系图构建算法

本文在ProGraML图的基础上，提出一种构建指令关系图流程算法。算法主要分为生成ProGraML图和根据ProGraML图进行节点和边的处理，最终生成指令关系图IRG。

4.2.1 ProGraML 图生成

210 ProGraML是一种图结构程序表示法，可应用机器学习模型的学习代码语义。ProGraML生成工具只需pip安装，无需编译。该工具支持多种编程语言，如C、C++、LLVM-IR、XLA等，同时也支持多种图形格式，如NetworkX、DGL、Graphviz、JSON等。如前文介绍，ProGraML可捕捉整个程序中的所有控制、数据和调用关系。表现形式与源语言无关。可按任何粒度添加特征和标签，以支持整个程序、每条指令或每种关系的推理任务。由于ProGraML包含的
215 全面的信息，故本文在NetworkX格式的ProGraML基础上进行修改生成指令关系图。

4.2.2 指令关系图生成

本文在ProGraML图的基础上生成指令关系图，缩减了图的规模的，有效捕获了代码指令间的控制流、数据流和调用流，从而更好的学习其中的代码语义信息。指令关系图生成算法如算法 1 所示。

算法 1: 指令关系图生成算法

输入: ProGraML G

输出: 指令关系图 IRG

```
1.  begin:
2.  nodes_to_remove = [] //需要删除的节点
3.  for node in G.nodes: //遍历ProGraML的节点
4.      if node.type is 1://如果是变量节点
5.          In_neighbors = node.predecessors
6.          out_neighbors = node.successors
7.          nodes_to_remove.append(node)
8.          for in_node in in_neighbors:
9.              for out_node in out_neighbors:
10.                  G.add_edge(in_node, out_node, flow=1)
11. edges_to_remove = []//需要删除的边
12. for edge in G.edges://遍历ProGraML的边
13.     if edge.flow is 2://如果是调用流边
14.         edges_to_remove.append(edge)
15.         G.add_edge(edge.out, edge.in, flow=2)
16.     if (node.in.text is "[external]" and node.out.text is "; undefined function") or (node.in.text is ";
undefined function" and node.out.text is "[external]"):
17.         edges_to_remove.append(edge)
18. G.remove_nodes_from(nodes_to_remove)//删除变量节点
19. G.remove_edges_from(edges_to_remove)//删除部分调用流边
20. return G
21. end
```

220 总的来说, 该算法遍历ProGraML中的所有节点和边, 首先将ProGraML中每条指令分成的输出变量节点、输入变量节点删除, 并保留操作码节点, 其中包含一条包含完整指令信息, 同时把经过变量节点相连接的指令节点通过数据流边相连。同时对于调用流边, 该算法将ProGraML中的调用流边进行处理, 保留从被调用指令或函数的节点到调用指令的调用流边。在指令关系图保留ProGraML中用以连接图中的所用ret指令的额外节点, 同时删去了与虚拟函数节点的调用关系, 以更好的保留代码文本中本身的代码语义。

225

5 代码指令嵌入

在现有的代码指令嵌入方法中, 大多将代码指令看作文本, 使用处理自然语言的方法 word2vec^[24]进行嵌入表示, 实际上代码指令本身的一定的结构信息, LLVM 在指令选择算

法的中间表示 SelectionDAG^[25]就提取了代码指令的结构表示，如图 5 所示。SelectionDAG 是 LLVM 编译器框架中的一个中间表示。SelectionDAG 的主要作用是进行低级别的优化和代码生成。它是在 LLVM-IR 和目标指令集之间的一个中间表示形式，用于描述高级源代码在目标机器上的执行。一个 SelectionDAG 表示一个代码块，其中每个节点代表一个操作数或操作符。节点中包含了操作码、参数类型、参数数量和输出类型等信息。节点间的边代表了节点的依赖关系，共有三类关系类型，包括数据依赖关系,执行依赖关系和强执行依赖关系。利用 SelectionDAG 可以将 LLVM-IR 转换为目标机器的汇编指令。由此可见，SelectionDAG 中的结构可以准确表达指令的语义。以此为依据，本文将指令代码分解，指令嵌入时考虑了代码指令中各个部分及关系，在更好的表达代码指令语义的同时，也解决了 OOV 问题。

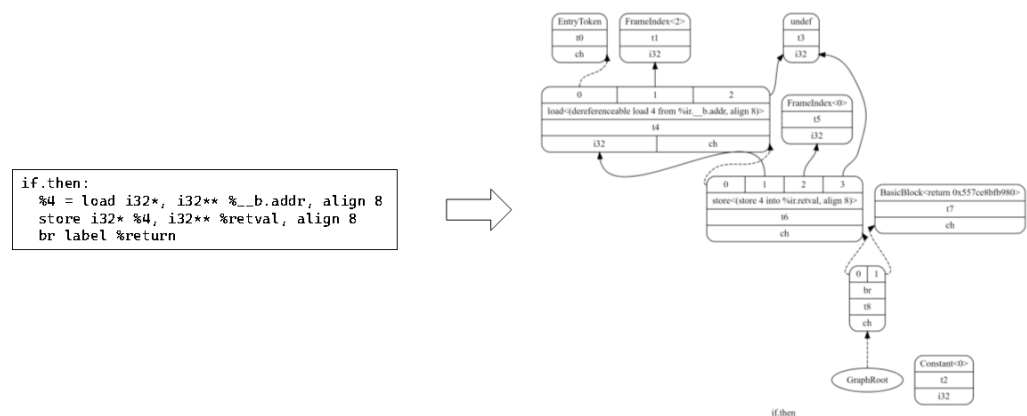


图 5 SelectionDAG 示例
Fig.5 Example of SelectionDAG

同时由 IR2Vec 启发，本文利用 TransE 模型学习 LLVM-IR 的指令嵌入。将一条指令拆分为操作码、操作类型和参数，再利用 TransE 模型学习各种操作码、操作类型和不同类型参数嵌入表示的同时，也学习它们之间关系，最后再将三者组合，共同表示一条指令的特征。具体方式为从 LLVM-IR 代码中获取每条指令的操作码、操作类型和操作数类型，其中操作类型与指令输出类型相同，并将操作码和参数类型表示为相应的实体名。表 1 展示了参数类型与构造实体的对应关系。然后构造两个关系：（1）*type*：操作码与操作类型之间的关系 （2）*parameter*：操作码与参数类型之间的关系。最后将 LLVM-IR 指令进行到关系 $\langle h, r, t \rangle$ 三元组转换，其中 h 为操作码、 r 为关系、 t 为指令类型或指令参数类型，作为 TransE 模型的输入。三元组示例如表 2 所示。

表 1 指令参数类型和对应实体
Tab.1 Instruction parameter types and corresponding entities

参数类型	实体
变量	var
指针	ptr
结构体	struct
常量	const

函数

function

学习到指令各部分的嵌入表示之后, 就可以将各部分合成表示一整条指令的特征。设指令 I 的各部分实体表示为 $O(I)$ 、 $T(I)$ 、 $P(I)$, 对应表示操作码、操作类型和参数类型, 一整条指令的嵌入 $\|I\|$ 可表示为三者嵌入表示 $\|O(I)\|$ 、 $\|T(I)\|$ 、 $\|P(I)\|$ 的组合。具体为公式(1)所示:

$$\|I\| = w_0\|O(I)\| + w_1\|T(I)\| + w_2(\|P_0(I)\| + \dots + \|P_n(I)\|) \quad (1)$$

其中, w_0 、 w_1 、 w_2 分别代表指令各部分的权重, 本文赋予操作码更多的权重, 分别设置为 1, 0.8, 0.4。至此可以表示出一条指令的嵌入表示。例如 `store i32 %0, i32* %3, align 4` 的嵌入可表示为 $w_0(\|store\|) + w_1(\|Integer\|) + w_2(\|var\| + \|ptr\|)$ 。

表 2 指令到三元组转换

Tab.2 Instruction to triple conversion

store i32 %0, i32* %3, align 4
<store,type,Integer>
<store,parameter,var>
<store,parameter,ptr>

6 代码相似度分析神经网络模型

6.1 模型总体结构

图相似度分析是最重要的基于图的应用程序之一, 例如, 找到与查询化合物最相似的化合物、图编辑距离和最大公共子图等。SimGNN^[26]提出了一种新的基于神经网络的方法来解决这一经典但具有挑战性的图问题, 旨在在保持良好性能的同时减轻计算负担。

基于 SimGNN, 本文提出了一种基于图的代码相似度分析模型, 如图 6 所示。该模型由三个主要组成部分组成: GNN 卷积层、图池化层和图相似度计算层, 用于计算两个代码表示图之间的相似度。

具体而言, 该论文首先获取二进制代码和源代码生成的代码关系图, 并利用先前的指令嵌入算法为每个节点生成特征表示。同时, 该模型标注了图中的控制流、数据流和调用依赖流边, 并为每个节点指定所属的函数和代码块信息。随后, 通过基于消息传递机制的图神经网络, 在整个代码结构中学习各个指令节点之间的关系。然后, 通过图池化层将指令关系图粗化为全局控制流图, 并使用消息传递机制的图神经网络来学习代码的语义信息。在粗化后的图中, 每个节点表示整个代码块。最后, 通过图相似度计算层获取整个图的特征表示, 并计算出两个图之间的相似度。

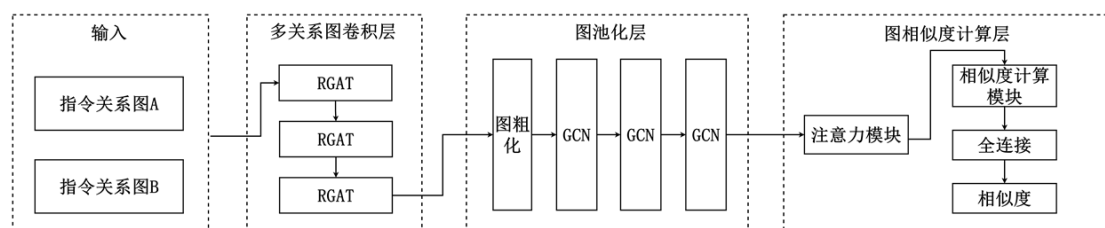


图6 代码相似度分析神经网络设计架构

Fig.6 Graph similarity analysis neural network design architecture

6.2 多关系卷积层

指令关系图是异构图，具有多种类型的边，如控制流边、数据流边和调用流边，因此本文使用能够处理包含多种类型边的图结构的关系型图注意力神经网络 RGAT^[27]。RGAT 可视为可以处理多种关系的 GAT^[28]。GAT 注意力机制可以为每个邻居分配不同的注意力权重，从而识别出更重要的邻居节点。GAT 在消息传播过程引入自注意力机制，每个节点的隐藏状态通过注意其邻居节点来计算。RGAT 和 GAT 类似，但在节点间关系上下功夫，RGAT 利用边的关系特征计算一个注意力系数，最后通过传播算法更新节点特征。设 RGAT 的输入是一个具有 $R = |\mathcal{R}|$

个关系类型和 N 个节点的图，第 i 个节点的特征由 $h_i \in \mathbb{R}^F$ 表示，所有节点的特征矩阵为 $H = [h_1 h_2 \dots h_N] \in \mathbb{R}^{N \times F}$ ，转换后的特征矩阵为 $H' = [h'_1 h'_2 \dots h'_N] \in \mathbb{R}^{N \times F'}$ ，每种关系类型的注意力系数为 $\alpha_{i,j}^{(r)}$ ， $G^{(r)} = [g_1^{(r)} g_2^{(r)} \dots g_N^{(r)}]$ 是节点间关系 r 下的特征矩阵，节点特征更新算法如公式(2)所示。

$$h'_i = \sigma \left(\sum_{r \in \mathcal{R}} \sum_{j \in n_i^{(r)}} \alpha_{i,j}^{(r)} g_j^{(r)} \right) \in \mathbb{R}^{N \times F'} \quad (2)$$

本文利用 IRG 显式保存指令间的控制流关系、数据流关系和调用流关系。使用关系型图注意力神经网络分别对三类关系进行消息传递和聚合操作。

6.3 图池化层

指令关系图中的节点都表示一条指令或是一个变量，图中的边也表征了指令之间的控制流、数据流和调用流的结构信息。本文通过将指令关系图进行图粗化，即节点聚合，从而将图中的节点表示整个代码块，图中的边表示基本块之间的控制流，从代码块的结构中学习代码语义。指令关系图粗化后如图 7 所示。

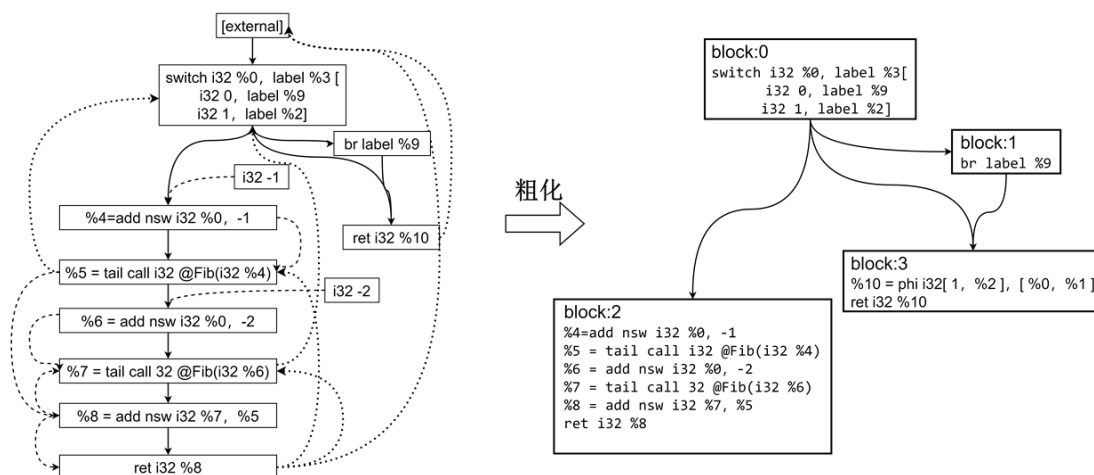


图7 指令关系图粗化

Fig.7 Coarsening of Instruction Relation Graph

粗化算法主要将具有相同函数表示和基本块标识的节点进行特征聚合，聚合方式使用

非加权平均算法来表示代码块特征，然后经过 3 个 GCNConv^[29]学习粗化后的图的结构语义信息。另外本文将图中的全局变量节点删除。粗化算法如算法 2 所示。

算法 2:指令关系图粗化算法

输入: 指令关系图IRG

输出: 全局控制流图GCFG

```

1. begin:
2. cluster=[]//节点聚合列表初始化
3. idx = 0
4. for function in IRG.functions://生成节点聚合列表
5.   nodes = nodes belong to function
6.   dict = { }
7.   for node in nodes:
8.     if node.block not in dict:
9.       dict[node.block] = []
10.      dict[nodes.block].append(node)
11. for value in dict:
12.   for i in value:
13.     cluster[i] = idx
14.   idx += 1
15. mask = (edge_type == 0) | (edge_type == 2)
16. selected_edge_index = IRG.edge[:, mask]//筛选控制流和调用流
17. gcfg = avg_pool(cluster, graph.x, selected_edge_index)//节点特征平均聚合，生成全局控制流图
18. gcfg remove isolated nodes//删除孤立节点
19. return gcfg
20. end

```

6.4 图相似度计算

经过 T 轮传播和聚合之后，各个节点特征得到了更新，需要从各个节点特征中获取整图特征。要使用一组节点嵌入为每个图生成一个嵌入，可以执行节点嵌入的非加权平均值，或者执行一个加权和，其中与节点相关联的权重由其程度决定。然而，哪些节点更重要，应该得到更多的权值取决于特定的相似性度量。本文使用 SimGNN 中的算法从一组节点的特征中计算出整图的特征。如公式(3)所示：

$$h = \sum_{n=1}^N f_2(h_n^T \tanh((\frac{1}{N} \sum_{m=1}^N h_m) W_2)) h_n \quad (3)$$

$f_2(\cdot)$ 其中是 *sigmoid* 函数。

计算两个图的图级特征之间的相似度的一个简单方法是取两个图级特征 $h_i \in \mathbb{R}^D, h_j \in \mathbb{R}^D$ 的内积。然而，这种简单的数据表示的使用往往会导致两者之间相似度的计算不足或较弱。本文使用 TenorNetworkModule^[30]计算两个图级特征之间的关系，如公式(4)所示：

$$g(h_i, h_j) = f_3\left(h_i^T W_3^{[1:K]} h_j + V \begin{bmatrix} h_i \\ h_j \end{bmatrix} + d\right) \quad (4)$$

315 其中 $W^{[1:K]} \in \mathbb{R}^{D \times D \times K}$ 是一个权重张量, $[]$ 表示连接操作, $V \in \mathbb{R}^{K \times 2D}$ 是一个权重向量, $d \in \mathbb{R}^K$ 是一个偏差向量, $f_3(\cdot)$ 是一个激活函数。 K 是一个超参数, 表示模型为每个图嵌入对产生的交互 (相似性) 分数的数量。在获得一个相似度得分向量后, 应用一个标准的多层全连通神经网络来逐步降低相似度得分向量的维数。最后, 获得一个的相似度分数。

6.5 模型学习方法

320 以上通过将二进制代码和源代码的嵌入映射到一个同一约束的公共空间来学习代码特征, 并计算相似度。直觉是, 如果二进制代码和源代码具有相似的语义, 那么它们的特征应该彼此接近。本文构建三元组 $\langle b, s^+, s^- \rangle$, 其中 $b, s^+, s^- \in D$ 表示一个训练实例, 其中 b 表示二进制代码, s^+ 表示具有相同语义的源代码 (也称为正样本), s^- 表示具有不同语义的源代码。当训练时, 使用以下损失函数计算 \mathcal{L} :

$$\mathcal{L} = \sum_{(b, s^+, s^-) \in D} \max(0, \beta - \text{sim}(b, s^+) + \text{sim}(b, s^-)) \quad (5)$$

325 其中 D 表示训练数据集, sim 表示二进制代码和源代码之间的相似度, β 是一个常数。 b 、 s^+ 和 s^- 分别是样本的特征。本文默认将 β 设置为 0.6。

7 实验结果与分析

7.1 数据集

330 本文在二进制代码与源代码之间的语义相似性方面的实验主要基于开源数据集 CodeNet^[31]。CodeNet 项目是一个大型数据集, 包含约 1,400 万个代码样本, 每个样本都是对 4000 个编码问题之一的解决方案。代码样本来自两个 OJ 网站: AIZU Online Judge 和 AtCoder。这些代码样本是用 50 多种编程语言编写的, 主要语言是 C++、C、Python 和 Java。本文截取了其中部分的 C++ 和 C 代码进行了实验。在二进制-源代码匹配中, 本文通过将 C/C++ 源代码编译成二进制代码, 同时保持另一个具有相同语义的的源代码不变, 进行匹配。为了研究编译优化等级对本文方法效果的影响, 因此本文编译的二进制代码包含了具有多个优化等级 (即 -O0、-O1、-O2、-O3)。本文按照 6: 2: 2 的比例将数据集划分为训练集、验证集和测试集。表 3 显示了本文使用的可编译的数据集的统计数据。

表 3 二进制源代码匹配的 CodeNet 数据集

Tab.3 CodeNet dataset of binary-source code matching

	源代码	LLVM-IR	二进制文件	反编译的 LLVM-IR
C	16000	13929	13929	12015
C++	16000	14375	14375	12793
Total	32000	28304	28304	24808

7.2 实验环境与实验指标

7.2.1 实验环境

340 本文的模型测试环境为 Ubuntu20.04 操作系统, 14 vCPU Intel(R) Xeon(R) Gold 6330

CPU @ 2.00GHz 处理器, 45 GB 内存, RTX 3090 显卡, 24GB 显存。模型使用 GPU 显卡加速模型训练。在图结构数据生成部分, TransE 算法生成的指令嵌入向量的

num_node_features 大小为 300。在实验中, 本文设置 0.6 为默认阈值, 即当相似度大于或等于 0.6 时, 认为代码对是相似的。为了验证方法的有效性, 本文还选取了现有的其他二进制和源代码匹配的方法进行了对比试验。如 BinPro、B2Sfinder 和 XLIR。

7.2.2 实验指标

本文采用召回率、精确度和 F1 分数进行模型评价。召回率, 表示匹配正确的样本占实际所有样本的比例, 召回率越高越好。精确率, 表示在模型检测出匹配的所有样本中真正匹配的样本所占的比例, Precision 越高越好。F1 分数是 Precision 和 Recall 的调和平均, 综合了两者的结果, F1 值越高, 代表模型性能越好。这些指标已广泛应用于文本匹配和信息检索。对于分析的代码片段(源代码或二进制代码), 本文使用将相应的具有相同语义的不同代码作为正代码, 将具有不同语义代码作为负代码。设 T_p 和 F_p 为检测到的真和假阳性相似数, F_n 为检测到的假阴性克隆数, 精确度 P 、召回率 R 和 $F1$ 分数计算公式如下:

$$P = \frac{T_p}{T_p + F_p}, R = \frac{T_p}{T_p + F_n}, F1 = \frac{2PR}{P + R}. \quad (6)$$

7.3 实验结果

本文的实验在 CodeNet 数据集上进行, 其中表二展示了 C/C++ 的二进制文件和源文件之间的二进制源代码匹配的性能。从该表 4 中可以观察到, 本文提出的方法在检测二进制文件与源代码的语义克隆方面表现良好。在 C/C++ 二进制到 C++ 源代码匹配任务中, 精确度、召回率和 F1 分别为 0.87、0.92 和 0.87。这些结果表明, 通过将二进制文件和源代码转换为 LLVM-IR 表示后, 可以学习到它们的代码语义信息。这表明 LLVM-IR 的编译和反编译过程有效地保留了源代码和二进制代码的语义, 并验证了本文方法的有效性。

表 4 本文方法与其他方法对比实验

Tab.4 Comparative experiments between this method and other methods

	精确率	召回率	F1 分数
BinPro	0.40	0.47	0.43
B2SFinder	0.42	0.51	0.46
XLIR(LSTM)	0.65	0.75	0.69
XLIR(Transformer)	0.83	0.88	0.85
本文方法	0.87	0.92	0.87

为了证明指令关系图的有效性, 本文进行了 ProGraML 和指令关系图的对比实验, 结果如表 5 所示, 可以看到在其他条件相同的情况下, 使用指令关系图的方法相比 ProGraML 在性能上有所提升, 即精确度, F1 平均上升了 0.07 和 0.05。

表 5 指令关系图和其他代码表示图对比实验

Tab.5 Comparison experiment between Instruction Relation Graph and other code representation graph

	精确率	召回率	F1 分数
--	-----	-----	-------

本文方法+ProGraML	0.80	0.93	0.82
本文方法+IRG	0.87	0.92	0.87

370 为了研究不同编译选项对相似性分析的影响，本文使用了由不同编译选项编译生成的二进制代码，并进行了源文件和二进制文件的相似性检测。结果如表 6 所示。观察结果可以发现，该论文的方法在使用 Clang 编译的二进制文件上保持了良好的性能，但在优化等级 3 和 4 时略有下降。而对于使用 GCC 编译的二进制文件，性能下降明显。使用 Clang 编译的结果显示，精确度、召回率和 F1-score 的平均值分别约为 0.80、0.87 和 0.82，方差分别为 0.00177、0.00186 和 0.00157。而使用 GCC 编译的结果显示，精确度、召回率和 F1-score 的平均值约为 0.73、0.74 和 0.73，方差分别为 0.00008、0.00021 和 0.00008。这些结果表明，该方法能够有效地匹配源代码和不同编译选项生成的二进制文件。

表 6 本文方法在不同编译选项下的性能

Tab.6 Performance of this method under different compilation options

	Clang			GCC		
	精确率	召回率	F1 分数	精确率	召回率	F1 分数
O0	0.87	0.92	0.87	0.74	0.73	0.74
O1	0.82	0.90	0.83	0.74	0.76	0.75
O2	0.77	0.83	0.79	0.72	0.74	0.73
O3	0.76	0.82	0.77	0.73	0.73	0.73

380 为了研究阈值对最终结果的影响，该论文进行了一系列的分析，并对阈值进行了调整。图 7 展示了这些调整的结果。从图中可以观察到，在一定范围内，随着阈值的增加，精确度会增加而召回率会降低。当阈值从 0.2 增加到 0.6 时，精确度和召回率显著提高。然而，当阈值超过 0.6 时，精确度和召回率开始下降。因此，该论文得出结论：相似度阈值越高，并不一定导致精确度和召回率越高。在阈值为 0.6 时，可以达到一个平衡点。

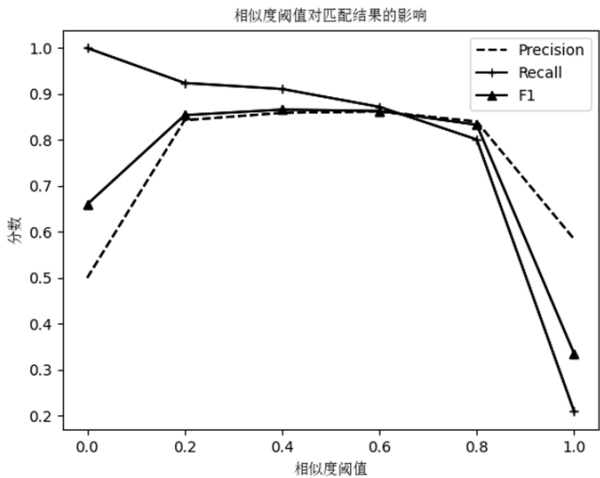


图 8 相似度阈值对匹配结果的影响

Fig.8 The impact of similarity threshold on matching results

8 总结与展望

本文提出了一种基于指令关系图的二进制代码和源代码相似性分析方法。该方法通过解析 LLVM-IR 的指令结构来学习指令的嵌入表示,并引入了指令关系图 IRG。然后,构建了一个代码相似性分析神经网络,利用图神经网络在图上进行信息传播和聚合,通过图池化学习方法来捕捉代码的语义信息。最后,使用图相似度算法计算二进制代码和源代码之间的语义相似度,从而判断它们是否相似。在实验中,该方法在开源数据集 CodeNet 上进行了验证,并取得了良好的效果,相较于现有的方法具有更高的有效性。然而,该方法在优化等级高的二进制文件下的性能有一定下降。此外,该方法仅在 C 和 C++ 的源代码和二进制代码上进行了实验研究,而 LLVM 还支持其他多种语言,如 Rust、Swift、Fortran 等语言。未来的研究将扩展到其他高级语言的实验研究。另外,该方法采用的是全局指令关系图的分析方法,因此分析粒度是文件级别的。实际上,在其他代码相似性研究中还存在其他粒度的分析方法,例如函数级别和代码基本块级别。未来的研究将探索更细粒度的代码相似性分析。

[参考文献] (References)

- [1] Duan R, Bijlani A, Xu M, et al. Identifying open-source license violation and 1-day security risk at large scale[C]//Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security. 2017: 2169-2185.
- [2] Yuan Z, Feng M, Li F, et al. B2SFinder: Detecting open-source software reuse in COTS software[C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019: 1038-1049.
- [3] Rahimian A, Charland P, Preda S, et al. RESource: a framework for online matching of assembly with open source code[C]//Foundations and Practice of Security: 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers 5. Springer Berlin Heidelberg, 2013: 211-226.
- [4] Miyani D, Huang Z, Lie D. Binpro: A tool for binary source code provenance[J]. arXiv preprint arXiv:1711.00830, 2017.
- [5] Aslanyan H, Movsisyan H, Arutunian M, et al. Bin2Source: Matching Binary to Source Code[C]//2021 Ivannikov Ispras Open Conference (ISPRAS). IEEE, 2021: 3-7
- [6] Yu Z, Zheng W, Wang J, et al. Codemr: Cross-modal retrieval for function-level binary source code matching[J]. Advances in Neural Information Processing Systems, 2020, 33: 3872-3883.
- [7] Ji Y, Cui L, Huang H H. Buggraph: Differentiating source-binary code similarity with graph triplet-loss network[C]//Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. 2021: 702-715.
- [8] Gui Y, Wan Y, Zhang H, et al. Cross-language binary-source code matching with intermediate representations[C]//2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2022: 601-612.
- [9] Sohn J, Yoo S. Flucss: Using code and change metrics to improve fault localization[C]//Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2017: 273-283.
- [10] 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.
- [11] VenkataKeerthy S, Aggarwal R, Jain S, et al. Ir2vec: Llvm ir based scalable program embeddings[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2020, 17(4): 1-27.
- [12] Bordes A, Usunier N, Garcia-Duran A, et al. Translating embeddings for modeling multi-relational data[J]. Advances in neural information processing systems, 2013, 26.
- [13] Guo D, Ren S, Lu S, et al. Graphcodebert: Pre-training code representations with data flow[J]. arXiv preprint arXiv:2009.08366, 2020.
- [14] Peng D, Zheng S, Li Y, et al. How could neural networks understand programs?[C]//International Conference on Machine Learning. PMLR, 2021: 8476-8486.
- [15] Zhang J, Wang X, Zhang H, et al. A novel neural source code representation based on abstract syntax tree[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 783-794.
- [16] Alon U, Zilberstein M, Levy O, et al. code2vec: Learning distributed representations of code[J]. Proceedings of the ACM on Programming Languages, 2019, 3(POPL): 1-29.
- [17] Ben-Nun T, Jakobovits A S, Hoeffler T. Neural code comprehension: A learnable representation of code

- 440 semantics[J]. Advances in Neural Information Processing Systems, 2018, 31.
- [18] Cummins C, Fisches Z V, Ben-Nun T, et al. ProGraML: A graph-based program representation for data flow analysis and compiler optimizations[C]//International Conference on Machine Learning. PMLR, 2021: 2244-2253.
- [19] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//International symposium on code generation and optimization, 2004. CGO 2004. IEEE, 2004: 75-86.
- 445 [20] Křoustek J, Matula P, Zemek P. Retdec: An open-source machine-code decompiler[C]//July 2018. 2017.
- [21] Brauckmann A, Goens A, Ertel S, et al. Compiler-based graph representations for deep learning models of code[C]//Proceedings of the 29th International Conference on Compiler Construction. 2020: 201-211.
- [22] Cummins C, Fisches Z V, Ben-Nun T, et al. ProGraML: Graph-based deep learning for program optimization and analysis[J]. arXiv preprint arXiv:2003.10536, 2020.
- 450 [23] Faustino A. Graphs based on IR as Representation of Code: Types and Insights[C]//Proceedings of the 25th Brazilian Symposium on Programming Languages. 2021: 75-82.
- [24] Mikolov T, Chen K, Corrado G, et al. Efficient estimation of word representations in vector space[J]. arXiv preprint arXiv:1301.3781, 2013.
- [25] Pandey M, Sarda S. LLVM cookbook[M]. Packt Publishing Ltd, 2015.
- 455 [26] Bai Y, Ding H, Bian S, et al. Simgnn: A neural network approach to fast graph similarity computation[C]//Proceedings of the twelfth ACM international conference on web search and data mining. 2019: 384-392.
- [27] Busbridge D, Sherburn D, Cavallo P, et al. Relational graph attention networks[J]. arXiv preprint arXiv:1904.05811, 2019.
- 460 [28] Veličković P, Cucurull G, Casanova A, et al. Graph attention networks[J]. arXiv preprint arXiv:1710.10903, 2017.
- [29] Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks[J]. arXiv preprint arXiv:1609.02907, 2016.
- [30] Socher R, Chen D, Manning C D, et al. Reasoning with neural tensor networks for knowledge base completion[J]. Advances in neural information processing systems, 2013, 26.
- 465 [31] Puri R, Kung D S, Janssen G, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks[J]. arXiv preprint arXiv:2105.12655, 2021.