软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(7):3003−3021 [doi: 10.13328/j.cnki.jos.007335] [CSTR: 32375.14.jos.007335]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# 面向函数内联场景的二进制到源代码函数相似性检测方法*

贾 昂, 范 铭, 徐 茜, 晋武侠, 王海军, 刘 烃

(西安交通大学 电子与信息学部, 陕西 西安 710049)

通信作者: 范铭, E-mail: mingfan@mail.xjtu.edu.cn

**摘 要**: 二进制到源代码函数相似性检测是软件组成成分分析的基础性工作之一. 现有方法主要采用一对一的匹配策略, 即使用单一的二进制函数和单一的源代码函数进行比对. 然而, 由于函数内联的存在, 函数之间的映射关系实际上表现为一对多——单一的二进制函数能够关联至多个源代码函数. 这一差异导致现有方法在函数内联场景下遭受了 30% 的性能损失. 针对函数内联场景下的二进制到源代码函数匹配需求, 提出了一种面向一对多匹配的二进制到源代码函数相似性检测方法, 旨在生成源代码函数集合作为内联二进制函数的匹配对象, 以弥补源代码函数库的缺失. 通过一系列实验评估了方法的有效性. 实验数据表明, 方法不仅能够提升现有二进制到源代码函数相似性检测的能力, 而且还能够找到内联的源代码函数, 帮助现有工具更好地应对内联挑战.

**关键词**: 二进制到源代码函数相似性检测; 函数内联; 源代码函数集合

**中图法分类号**: TP311

中文引用格式: 贾昂, 范铭, 徐茜, 晋武侠, 王海军, 刘烃. 面向函数内联场景的二进制到源代码函数相似性检测方法. 软件学报, 2025, 36(7): 3003–3021. http://www.jos.org.cn/1000-9825/7335.htm

英文引用格式: Jia A, Fan M, Xu X, Jin WX, Wang HJ, Liu T. Binary2Source Function Similarity Detection Method Under Function Inlining. Ruan Jian Xue Bao/Journal of Software, 2025, 36(7): 3003–3021 (in Chinese). http://www.jos.org.cn/1000-9825/7335.htm

## Binary2Source Function Similarity Detection Method Under Function Inlining

JIA Ang, FAN Ming, XU Xi, JIN Wu-Xia, WANG Hai-Jun, LIU Ting

(Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China)

**Abstract**: Binary2Source function similarity detection is regarded as one of the fundamental tasks in software composition analysis. In the existing binary2Source matching works, the 1-to-1 matching mechanism is mainly adopted, where one binary function is matched against one source function. However, it is found that such a mapping may be 1-to-n (one binary function is mapped to multiple source functions) due to the existence of function inlining. A 30% performance loss is suffered by the existing binary2Source matching methods under function inlining due to this difference. Aimed at the matching requirement of binary to source functions in the scene of function inlining, a binary2Source function similarity detection method for 1-to-n matching is proposed in this study, which is designed to generate source function sets as the matching objects for the inlined binary functions to make up for the lack of the source function library. The effectiveness of the proposed method is evaluated through a series of experiments. The experimental data indicate that the method can not only improve the existing binary2Source function similarity detection ability but also identify the inlined source code functions, helping the existing tools better cope with the challenges of inlining.

**Key words**: binary2Source function similarity detection; function inlining; source function set

现代软件发展过程中, 开源软件得到了越来越广泛的发展和使用, 对开源软件的复用也越来越普遍. 根据美国新思科技公司在 2022 年的报告, 96% 的软件在开发过程中都会使用到开源代码. 虽然开源代码的使用有助于软件

* 基金项目: 国家自然科学基金 (62232014, 62272377, 62372368, 62372367); 陕西省科学技术协会青年人才托举计划
本文由"新兴软件与系统的可信赖性与安全"专题特约编辑向剑文教授、陈厅教授、杨珉教授、周俊伟教授推荐.
收稿时间: 2024-08-22; 修改时间: 2024-10-15; 采用时间: 2024-11-25; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-04-09

公司缩短开发周期, 降低开发成本, 但是不合规的代码复用也会给软件带来安全方面的风险. 根据奇安信发布的《2022 中国软件供应链安全分析报告》, 在 2 254 个企业软件中, 平均每个项目使用了 127 个开源软件, 其中 86.4% 的项目存在已知的开源软件漏洞. 而源代码项目通过编译后, 通常会以二进制的形式发行以供公开使用. 由于代码复用, 代码中潜在的安全问题常常会传递到下游的软件公司和使用者中. 例如由于对 OpenSSL 项目的代码复用, 心脏流血 (heartbleed) 漏洞导致了超过 17% 的网站都面临着安全风险.

为了解决这些由代码复用所引入的问题, 二进制到源代码相似性检测方法[1−11]被提出并应用于包括代码搜索, 开源软件复用检测, 以及软件组成成分分析在内的多个领域. 函数作为二进制文件和源代码项目的基本组成部分, 其相似性检测结果决定了具体应用领域的效果. 通常, 二进制到源代码相似性检测方法会将待检测的二进制函数定义为查询函数, 而可能会复用的源代码函数作为目标函数. 当查询函数与目标函数相似时, 相似性检测方法则会认定为函数之间存在复用关系. 已有的方法通常认为复用函数之间包含了等价的函数语义, 因此通常使用一对一的匹配方式来对函数进行比较.

然而, 复用函数之间并不一定总是承载着等价的函数语义. 当函数内联发生时, 一个二进制函数通常由两个或者多个源代码函数编译生成. 此时, 若二进制函数在编译过程内联了源代码函数, 查询函数和目标函数之间的映射关系将会从一对一转变为一对多. 我们先前的工作[12]发现函数内联广泛存在于二进制文件中, 当使用 O3 优化级别时, 接近 40% 的二进制函数是通过内联生成的. 而现有的二进制到源代码相似性检测方法在处理含内联的二进制函数时, 性能损失高达 30%. 因此急需解决函数内联带来的“一对多”的匹配问题.

针对上述问题, 本文提出了一种名为 O2NMatcher 的方法, 该方法采用了“一对多”的匹配机制来匹配二进制函数与源代码函数. 图 1 展示了 O2NMatcher 的核心思想. 总体而言, 传统的“一对一”匹配方法通过比较二进制函数和源代码函数得到复用结果, 而 O2NMatcher 旨在生成源代码函数集合, 用以补充源代码函数的缺失, 从而来与内联的二进制函数进行对比. 源代码函数集合是一个由多个源代码函数组成的函数集合, 该源代码函数集合包含了该二进制函数对应的源函数以及参与内联的其它源函数, 因此其语义等同于内联后的二进制函数. 从而 O2NMatcher 可以帮助现有技术在函数内联的情况下进行二进制到源代码函数相似性的匹配.

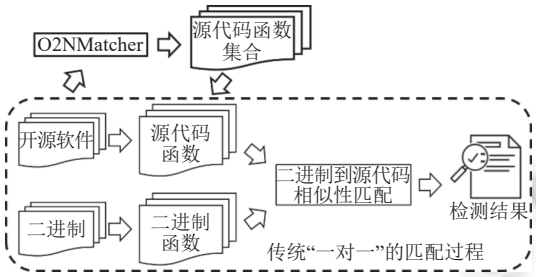



图 1 O2NMatcher 的“一对多”匹配思想

本文的主要工作和创新性贡献如下.

(1) 本文提出一种能够在函数内联场景下进行二进制到源代码函数相似性计算的检测方法 O2NMatcher, 该方法通过生成源代码函数集合, 有效提升了现有二进制到源代码函数相似性检测方法在内联场景下的检测能力.

(2) 为了使 O2NMatcher 能够准确地生成源代码函数集合, 本文进行了一项实证研究, 分析了不同编译设置下的内联关联性. 基于发现的内联规律, 将内联调用点的预测问题建模为一个多标签分类问题, 并提出了名为 ECOCCJ48 (ensemble of compiler optimization based classifier chains built with J48) 的模型.

(3) 通过对函数内联过程的观察, 提出了一种基于函数调用树的构建方法用于源代码函数集合的生成. 本文将源代码函数集合生成的过程转化为包含根节点选择和边扩展的内联函数调用图的生成问题, 最终生成的源代码函数集合能够有效地弥补源代码函数的缺失.

本文第 1 节介绍二进制到源代码相似性检测和函数内联的相关研究. 第 2 节介绍函数内联场景下的“一对

多”匹配问题. 第 3 节介绍本文在函数内联场景下进行的实证研究以及发现的内联规律. 第 4 节介绍面向函数内联场景的“一对多”的二进制到源代码函数相似性匹配方法. 第 5 节通过实验验证所提方法的有效性. 最后总结全文.

## 1 相关工作

### 1.1 二进制到源代码相似性检测

表 1 总结了现有的二进制到源代码相似性检测工作. 根据检测方法的不同, 这些工作可以分为 3 类: 基于特征的方法、基于编译的方法和基于深度学习的方法.

表 1 现有的二进制到源代码相似性检测工作

| 方法类型 | 方法名称 | 方法粒度 | 是否考虑内联 |
|---|---|---|---|
| 基于特征的方法 | BAT[1] | 文件级 | 否 |
| | Resource[2] | 文件级 | 否 |
| | JISIS14[3] | 函数级 | 否 |
| | BinPro[4] | 文件级 | 否 |
| | OSSPolice[5] | 文件级 | 否 |
| | SANER19[6] | 文件级 | 否 |
| | B2SFinder[7] | 文件级 | 否 |
| | B2SMatcher[8] | 文件级 | 否 |
| 基于编译的方法 | Buggraph[9] | 函数级 | 否 |
| | XLIR[10] | 函数级 | 否 |
| 基于深度学习的方法 | CodeCMR[11] | 函数级 | 否 |

基于特征的工作[1−8]通过比较提取的特征来进行匹配. 用于比较的特征主要包括字符串常量、数值常量、控制流结构以及函数参数信息. 表 1 中除 JISIS14 以外的所有工作几乎都用到了字符串进行直接比较, Resource、BinPro、B2SFinder 和 B2SMatcher 则是对数值常量也进行比较. 另外, OSSPolice、SANER19、B2SFinder 还引入了函数的控制信息, 而 JISIS14 和 B2SMatcher 还引入了函数的参数信息. 然而, 在函数内联场景下, 多个函数的函数特征会组合到一起, 甚至如数字常量和函数的控制信息等特征在组合之后还会发生改变. 而使用基于特征的工作进行检测时, 一个源函数的特征只是一个内联二进制函数的特征的一部分, 因此无法有效地检测出内联二进制函数的组成成分.

基于编译的工作[9,10]利用编译来沟通源代码与二进制文件, 从而进行二进制到源代码的匹配. Buggraph[9]首先使用与目标二进制文件相同的配置来编译源代码. 然后比较二进制函数以获得相似性. XLIR[10]将源代码和二进制文件都转换为 LLVM IR. 然后比较 LLVM IR 函数以获得相似性. 然而, 一方面, 由于编译选项的多样性, Buggraph 和 XLIR 的编译方法很难覆盖所有的内联情况; 另一方面, 并非所有的源代码项目都可以自动编译. 基于编译的工作在函数内联场景下面临着扩展性差的问题.

基于深度学习的工作[11]通过训练神经网络来进行二进制到源代码的匹配. CodeCMR[11]将二进制到源代码的匹配建模为一种跨模态检索任务, 其中源函数和二进制函数是神经网络的两个输入. 输出是源函数和二进制函数之间的相似性. 虽然 CodeCMR 目前在二进制到源代码相似性匹配中实现了最高的 *Recall*@1 值, 但是其训练过程并未考虑到函数内联的存在, 因此其在函数内联场景中也面临巨大的挑战.

我们先前的工作[12]曾对现有的二进制相似性方法在函数内联场景下进行了一个系统性的调研, 发现现有的二进制到源代码的相似性匹配工作都没有考虑到函数内联. 其中, 作为目前二进制到源代码相似性匹配的最新研究成果, BinaryAI 在函数内联场景上也遭受了 30% 的性能损失. 本文将从函数内联场景出发, 旨在补全现有方法的性能损失, 以提升现有方法在函数内联场景的检测准确率.

### 1.2 函数内联

函数内联技术是现代编译器优化策略中的重要组成部分, 其主要目标在于消除函数调用的开销, 同时为后续的优化操作创造有利条件. 该技术的核心在于将函数调用点处的代码替换为被调用函数的代码体, 从而在运行时避免了函数调用的间接跳转和参数传递等开销. 然而, 内联操作亦非毫无代价, 它可能导致目标代码体积增大, 进而增加存储空间, 同时延长编译时间, 增加了编译过程的复杂性. 因此, 如何在提高运行效率与控制资源消耗之间找到最佳平衡点, 成为了内联策略设计的关键所在[13−19].

在内联策略的制定与实施方面, GCC 和 LLVM 两大主流编译器展现出了不同的设计理念. GCC 采用了基于优先级的内联机制, 通过对每个函数调用点的内联收益进行评估, 结合预设的成本阈值, 动态决定内联操作的执行与否. 这一策略不仅考虑了内联带来的即时性能提升, 同时也兼顾了代码体积和编译时间的控制. 另一方面, GCC 针对小函数及单次调用函数等特殊情况, 提供了专门的内联优化选项, 进一步增强了其适应性与灵活性.

相对而言, LLVM 则采取了一种基于预测的内联方案. 通过分析函数的调用频率, LLVM 将函数区分为热点与冷点两类, 对热点函数采取积极的内联策略, 以期获得最大化的性能收益. 对于每个调用点, LLVM 都会细致计算内联的预期收益与成本, 只有当预期收益大于预期成本时, LLVM 才会执行内联操作.

### 1.3 函数内联场景下的二进制相似性检测工作

函数内联技术的引入, 为二进制代码相似性分析带来了前所未有的挑战. 以往的研究, 诸如 Bingo[20]和 Asm2Vec[21], 虽致力于解决函数内联环境下的二进制代码对比问题, 但其焦点主要集中在二进制到二进制的相似度检测, 未能深入探讨函数内联对二进制到源代码相似性检测的影响. 这些研究虽然在一定程度上提升了二进制代码层面的匹配精度, 却无法直接解决源代码级别的匹配挑战.

本文的前序工作[12]系统地评估了函数内联对二进制函数相似性分析的深远影响. 通过构建包含内联操作的数据集, 揭示了在函数内联存在的情况下, 传统匹配方法面临的性能瓶颈. 其结果显示, 多数现有技术在检测内联函数时, 匹配准确率显著下降, 性能损失高达 30%–40%. 然而, 本文的前序工作并未提出有效的解决方案, 这一难题仍悬而未决, 成为当前二进制到源代码相似性检测领域亟待突破的关键点.

鉴于函数内联技术对二进制到源代码相似性检测带来的挑战, 本研究致力于探索一种全新的解决方案, 旨在克服函数内联对匹配准确率的影响. 本文深入分析了函数内联的规律, 探究了其对二进制函数结构的改变, 以此为基础, 设计了一系列针对性的解决方法.

## 2 “一对多”匹配问题

本节首先对“一对多”匹配问题给出定义, 然后以实例来分析函数内联场景下的二进制到源代码相似性匹配问题, 最后介绍 O2NMatcher 针对上述问题所设计的方法流程.

### 2.1 问题定义

本文定义在函数内联场景下的二进制到源代码函数匹配目标如下: 给定一个发生内联的查询二进制函数 $q$ 和一组目标源代码函数集合 $T$, 函数内联下的二进制到源代码相似性检测旨在检索出源代码函数 $t$, 其中源代码函数 $t$ 通过内联其他源代码函数编译生成了二进制函数 $q$.

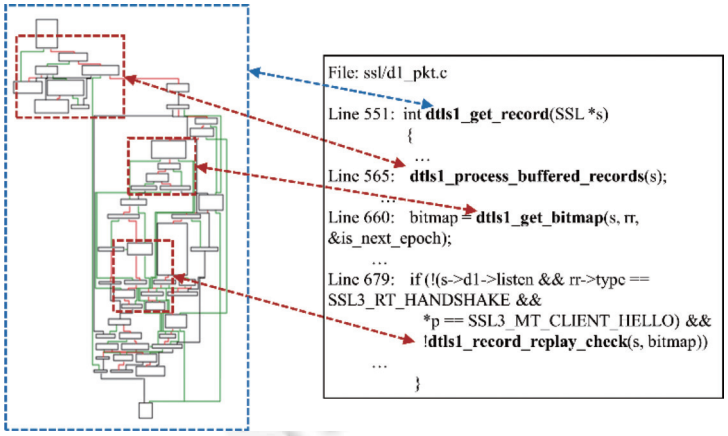
在普通场景下的代码搜索中, 对于提交查询的二进制函数, 仅需要检索到与二进制函数功能相同的源代码函数即可. 然而, 在内联场景下的代码搜索中, 寻找被内联的源代码函数对于二进制到源代码相似性检测同样重要, 本文所提出的 O2NMatcher 方法也可寻找其他被内联的源函数.

本文将研究的查询函数范围定位在由 GCC 和 Clang 编译的 C/C++代码所生成的二进制函数, 目标函数则是由上述 C/C++代码组成的源代码仓库. 二进制到源代码函数匹配的目标即为对于查询的二进制函数, 找到其对应的源代码仓库中的源代码函数.

### 2.2 案例分析

图 2 展示了函数内联场景下“一对多”匹配问题的一个例子. 函数 dtls1_get_record 是项目 OpenSSL 1.0.1j 中的

一个函数. 图 2(a) 展示了二进制函数 dtls1_get_record 的控制流图. 图 2(b) 展示其所对应的源函数代码. 虚线矩形和双头箭头表示从二进制函数到源代码函数的映射关系. 根据编译信息, 二进制函数 dtls1_get_record 是由源代码函数 dtls1_get_record 编译而来, 并且内联了多个其他源代码函数的内容, 包括 dtls1_process_buffered_records、dtls1_get_bitmap 和 dtls1_record_replay_check. 由于源代码函数 dtls1_get_record 包含了漏洞 CVE-2014-3571, 二进制函数 dtls1_get_record 也继承了这一漏洞.




(a) 二进制函数 dtls1_get_record 的 CFG (b) 源函数 dtls1_get_record

图 2 函数内联场景下的“一对多”匹配问题示例

然而, 当使用现有的二进制到源代码相似性检测技术来比较二进制函数 dtls1_get_record 和源代码函数 dtls1_get_record 时, 两个函数之间的相似度低于 50%, 导致无法检测出该漏洞. 通过编译信息可以看出, 尽管二进制函数 dtls1_get_record 是由源代码函数 dtls1_get_record 编译生成, 但其也包含了来自源代码函数 dtls1_process_buffered_records、dtls1_get_bitmap 和 dtls1_record_replay_check 的内容. 实际上, 漏洞函数的代码仅是二进制函数 dtls1_get_record 的一部分. 此时二进制函数 dtls1_get_record 和源代码函数之间已经成为了“一对多”的映射关系. 因此, 通过现有的“一对一”匹配机制难以识别出二进制函数所复用的源代码函数.

总体而言, 在函数内联场景下进行二进制到源代码函数相似性检测面临以下 3 个挑战.

挑战 1: 超出语料库问题. 由内联生成的二进制函数通常由一个以上的源代码函数生成. 因此, 在源代码函数中并不存在这样一个源代码函数, 其语义完全等同于查询的二进制函数. 因此, 含有内联的二进制函数在查询时面临着超出语料库问题, 即在开源软件语料库中不存在与查询二进制函数语义完全等价的源代码函数.

挑战 2: 函数内联的不透明性. 大多数开源软件项目无法实现全自动编译[7]. 因此即使拥有完整的源代码项目, 也很难推断函数内联将在何处发生. 这是因为内联操作并非直接在源代码上执行, 而是在编译过程中的中间语言阶段进行, 此时源代码已经经过了预处理、转换和优化[13]. 而内联策略决定是否进行内联的指标, 在源代码和中间语言上已经发生了改变.

挑战 3: 二进制文件中函数内联的多样性. 二进制文件可以通过各种编译设置以及不同的内联策略生成, 这导致二进制文件中存在着多种多样的内联结果. 在具体实现中, 几乎不可能手动总结所有这些内联模式, 这使得大规模处理“一对多”匹配面临着巨大挑战.

### 2.3 O2NMatcher 方法设计

为了应对挑战 1, O2NMatcher 通过构建源代码函数集合来作为内联二进制函数的匹配对象. 源代码函数集合不再是源代码仓库中的单一函数, 而是由初始源代码函数和参与内联的其他源代码函数组成. 例如在图 2 的例子中, O2NMatcher 会生成一个由源函数 dtls1_get_record, dtls1_process_buffered_records, dtls1_get_bitmap, 以及 dtls1_record_replay_check 组成的源代码函数集合, 由于该源代码函数集合和二进制函数 dtls1_get_record 具有相

等的语义, 因此当添加该源代码函数集合至开源软件语料库后, 不再面临超出语料库问题. O2NMatcher 的整体设计见第 4 节.

为了应对挑战 2, O2NMatcher 训练了模型 ECOCCJ48 用于自动化的函数内联预测. 模型 ECOCCJ48 将函数内联的预测问题建模为一个多标签分类问题, 并构建一个内联数据集用于训练和测试. 第 3.1 节介绍数据集的构建过程, ECOCCJ48 的训练和测试过程见第 4.1、4.2 节.

为了应对挑战 3, 本文对不同编译设置下函数内联之间的关联性展开了分析, 发现了在编译器家族和不同优化选项下的内联规律, 并依照此规律将 ECOCCJ48 设计为一个基于分类器链和二元关联的多标签分类模型. 第 3.2 节是函数内联规律分析, ECOCCJ48 模型的具体设计见第 4.2 节.

## 3 内联规律分析

在介绍 O2NMatcher 的具体设计之前, 本文先对函数内联的规律展开一系列调研.

### 3.1 数据集构建与标注

本文使用了 GNU 组件中的 51 个项目, 并使用 8 个编译器 (GCC-4.9.4, GCC-5.5.0, GCC-6.4.0, GCC-7.3.0, Clang-4.0, Clang-5.0, Clang-6.0, Clang-7.0), 4 个编译选项 (O0, O1, O2, O3) 编译到 X86-64 架构下, 共生成 7 520 个二进制文件和 1 130 467 个二进制函数. 编译过程使用了 Binkit[22]提供的编译工具进行构建.

对于源代码项目和编译后得到的二进制文件, 本文使用我们先前工作[12]中的内联函数标注方法, 以得到二进制函数和源代码函数之间的映射关系. 具体而言, 首先从二进制文件的. debug_line 段抽取了二进制地址到源文件代码行的行映射关系, 然后在源代码端抽取了源代码行和源函数的映射关系, 在二进制端抽取了二进制地址和二进制函数的映射关系, 最后将二进制函数和源代码函数通过二进制地址到代码行的映射进行拓展, 即可获得二进制函数和源代码函数的映射关系.

如果一个二进制函数映射到多个源代码函数上, 它将被识别为一个内联的二进制函数. 而为了进一步标注发生内联的函数调用, 本文继续分析了这些调用的行映射关系. 简单来说, 当存在一个二进制函数调用, 其地址映射到某个源代码调用位置时, 那么这个源代码调用位置是没有被内联的. 相反, 如果不存在二进制调用位置的地址能映射到某个源代码调用位置上, 那么这个源代码调用位置就是被内联的.

经过标注后, 数据集中共识别出了 211 680 个发生内联的二进制函数. 其中在编译过程中, 有 909 597 个函数调用未被内联, 173 453 个函数调用被内联.

### 3.2 内联规律分析

源代码函数集合需要对应发生内联的二进制函数, 而内联的函数调用决定了内联的二进制函数的生成. 因此, 哪些函数调用会被内联实际上决定源代码函数集合将如何构建. 基于上述逻辑, 本节将分析不同编译设置下的内联函数调用之间的关联性, 以便于后续源代码函数集合构建方法的设计.

#### 3.2.1 不同编译选项下的函数内联关系

图 3 展示了在 8 个编译器和 4 种优化选项下内联函数调用的分布情况. 对于每个编译器, 图中统计了高优化级别下的内联函数调用与低优化级别下的内联函数调用的交集, 并在使用不同颜色进行表示. 例如, 在 GCC-4.9.4 中, O3 优化级别柱状图的红色部分代表了在使用 O3 和 O0 优化时均被内联的函数调用, 蓝色部分代表了在使用 O3 和 O1 优化时均被内联的函数调用, 黄色部分代表了在使用 O3 和 O2 优化时均被内联的函数调用, 而绿色部分则代表了仅在使用 O3 优化时被内联的函数调用.

可以看出, 随着优化级别的提高, 内联函数调用的数量呈递增叠加趋势. 如图 3 所示, 在高优化级别下的内联函数调用大量继承了低优化级别下的内联函数调用. 例如, 在 GCC-4.9.4 中, 使用 O1 优化时有 4 156 个函数调用被内联, 其中 3 058 个函数调用 (占 75%) 在使用 O2 优化时也被内联, 2 956 个函数调用 (占 71%) 在使用 O3 优化时被内联. 总体而言, 低优化级别下被内联的函数调用中有 75.7% 在使用高优化级别时也被内联 (低优化级别内联的 105 553 个函数调用中有 79 862 个在高优化选项下也被内联).

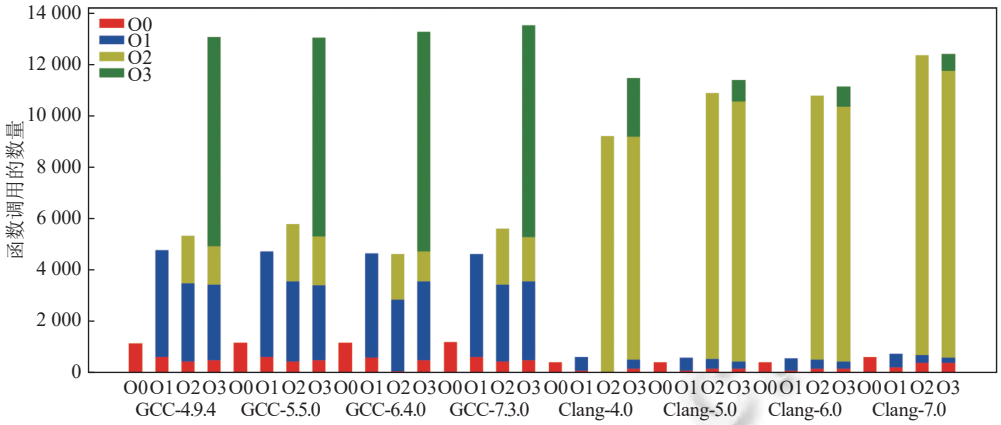



图 3　不同编译选项之间的函数内联关系

### 3.2.2　不同编译器下的函数内联关系

为了进一步分析不同编译器间函数内联的相关性, 本节使用公式 (1) 计算它们之间的相似度:

$$Sim_{cp1-cp2}=\frac{\sum_{i}^{opt}|ICS_{cp1}\cap ICS_{cp2}|}{\sum_{i}^{opt}\left(|ICS_{cp1}|+|ICS_{cp2}|\right)}\times 2 \tag{1}$$

在公式 (1) 中, $ICS$ 代表被内联的函数调用, $cp1$ 和 $cp2$ 分别代表两个要比较的编译器. $|ICS_{cp1}\cap ICS_{cp2}|$ 表示在相同优化级别下, 同时在 $cp1$ 和 $cp2$ 中被内联的函数调用数. $|ICS_{cp1}|$ 和 $|ICS_{cp2}|$ 分别代表在 $cp1$ 和 $cp2$ 中被内联的函数调用数. $\sum_{i}^{opt}\left(|ICS_{cp1}|+|ICS_{cp2}|\right)$ 意味着总结所有优化级别的比较结果.

图 4 中展示了 8 个编译器之间的相似结果, 其中颜色越深, 表示相似程度越高.

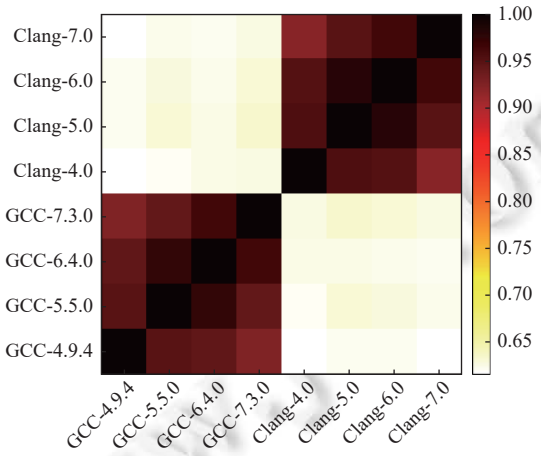



图 4　不同编译器之间的函数内联关系

如图 4 所示, 同一编译器家族的编译器做出类似的内联决策. 例如, GCC-4.9.4 和 GCC-5.5.0 中 95% 的内联函数调用是一致的. 总体而言, GCC 编译器中 95.2% 函数调用的内联结果是一致的, 而在 Clang 编译器中 94.9% 函数调用的内联结果是一致的.

尽管在同一编译器的不同优化级别以及同一家族的编译器间, 内联行为显示出明显的相关性, 但不同编译器

家族 (GCC 和 Clang) 之间的相关性并不明确. 因此, 后续 O2NMatcher 方法设计中会着重考虑以上的相关性, 以提高 O2NMatcher 在多种类函数内联情况下的处理能力.

## 4 面向函数内联场景的二进制到源代码函数相似性检测方法

如图 5 所示, 面向函数内联场景的二进制到源代码函数相似性检测方法. O2NMatcher 的工作流程分为 3 个部分: 特征提取、内联函数调用预测及源代码函数集合生成.

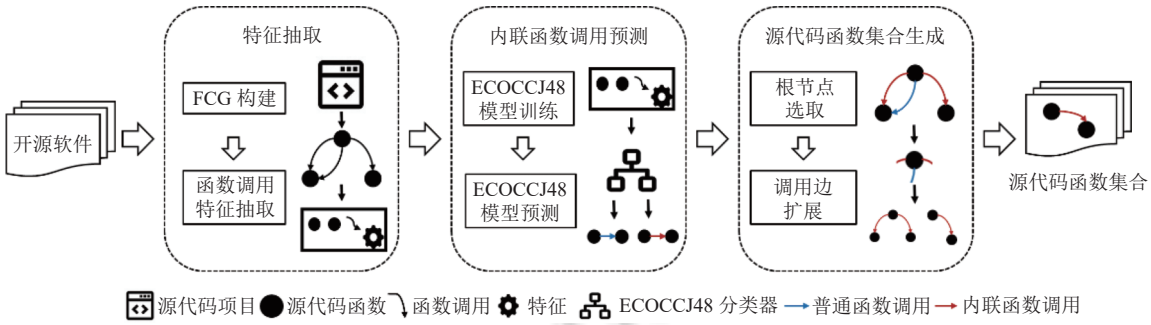



图 5 O2NMatcher 的工作流程

由于发生过内联的二进制函数是通过内联了某些函数调用而生成的, O2NMatcher 首先识别这些内联的函数调用, 然后执行内联操作以生成源代码函数集合.

### 4.1 特征抽取

通常讲, 编译器一般通过衡量内联某个函数调用的成本和收益来决定是否进行内联. 因此, 本节选取了一些能够影响内联成本和收益的特征. 所选特征列于表 2 中.

表 2 内联函数调用预测预测所选特征

| 对象 | 部分 | 特征 |
|---|---|---|
| 调用函数与被调用函数 | 函数体 | 语句总数, while语句数, switch语句数, case语句数, if语句数, for语句数, 返回语句数, 声明语句数, 表达式语句数 |
| | 函数定义 | Inline关键字数量, Static关键字数量 |
| | 函数调用 | 调用次数, 被调用次数 |
| 调用指令 | 位置 | 路径长度, 是否位于for循环内, 是否位于while循环内, 是否位于switch语句内, 是否位于if条件内 |
| | 参数 | 参数总数量, 常量参数数量 |

特征主要来自于两个部分: 调用函数与被调用函数, 以及调用指令.

#### 4.1.1 调用函数与被调用函数

调用函数与被调用函数的属性会影响内联被调用函数的成本和收益. 成本主要来源于函数内联带来的二进制文件的体积膨胀, 而收益主要来源于内联所减少的函数调用开销. 本节将从函数体、函数定义及函数调用这 3 个部分出发, 介绍本文所选取的用于内联函数调用预测的特征.

在函数体中, O2NMatcher 提取了不同类型的指令数量作为特征. 例如, 语句总数表示所有语句的数量, 而 while 语句数表示 while 循环语句的数量. 这些指令的计数代表了函数的体积大小, 这表明了内联该函数的成本. 简单来说, 被调用函数越复杂, 内联该函数调用的成本就越高.

例如在图 2 中, 函数 dtls1_get_record 有接近 170 行代码, 而函数 dtls1_process_buffered_records 只有不到 30 行代码, 相比之下, 函数 dtls1_process_buffered_records 被内联的成本远远低于函数 dtls1_get_record, 因此在图 2 中, 函数 dtls1_process_buffered_records 被内联到了函数 dtls1_get_record 中.

在函数定义中, O2NMatcher 提取了“Inline”和“Static”两个重要关键字的使用次数作为特征. 关键字“Inline”是对编译器的建议, 表明当其他函数调用此函数时, 应将其内联. 由 Static 修饰的函数仅可由同一编译单元中的其他函数访问. 这些关键字直接或间接地影响编译器的内联决策.

同样在图 2 中, 函数 dtls1_get_record 没有被定义为“Static”函数, 而函数 dtls1_process_buffered_records, 函数 dtls1_get_bitmap, 函数 dtls1_record_replay_check 都被定义为“Static”函数, 相比之下, Static 函数更容易被内联到其他函数中.

在函数调用中, O2NMatcher 提取了调用函数与被调用函数的调用次数和被调用次数作为特征. 直观上, 如果一个被调用函数被众多调用者函数调用, 将其内联到所有调用者中的成本会随着调用者数量的增加而增加. 相反地, 当一个函数仅被调用一次时, 内联只会带来优化的好处而不会增加代码大小, 这是进行内联的理想情况.

#### 4.1.2 调用指令

对于调用指令, 调用指令的位置和函数调用的参数信息是影响函数内联实施的两个重要因素. 例如, 如果一个函数调用位于由“for”或“while”定义的循环中, 内联该函数调用将显著减少函数调用的时间. 此外, 如果一个函数调用包含常量参数, 并且这个参数能帮助确定被调用函数中的某些分支, 那么内联该函数只需要内联对应的分支内容, 这也会使内联这些函数的成本有所降低.

### 4.2 内联函数调用预测

内联函数调用预测旨在预测开源软件中的内联函数调用, 主要包括两个部分: 模型训练和模型测试. 从流程上来讲, 本节首先设计一个分类器并在训练数据集上对其进行训练, 然后使用这个分类器来预测测试数据集中的函数调用的标签.

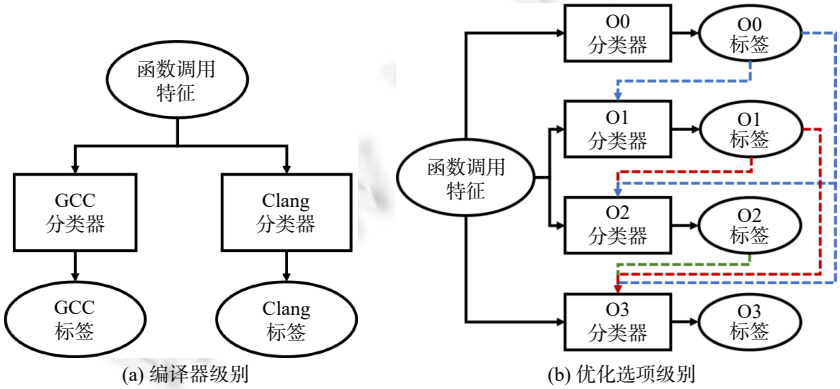
#### 4.2.1 模型训练

不同的编译器家族和不同的优化级别会导致不同的内联函数调用. 考虑到在每种编译设置下, 每个函数调用都将有一个相应的内联标签, 本文将内联函数调用预测问题视为一个多标签分类 (multi-label classification, MLC) 问题.

在多标签分类中, 预测的是多个标签的存在与否, 并且可以同时为一个样本分配多个标签. 例如, 由于在本章的数据集中, 每个函数调用在 8 种编译器×4 种优化级别=32 种编译配置下被编译, 每种编译情况下都有对应的内联情况, 因此它将有 32 个标签, 每个对应一种编译配置.

如第 3.2 节所发现的, 随着优化级别的增加, 内联函数调用呈递增叠加态势, 且同一编译器家族的编译器做出类似的内联决策. 考虑到内联决策间的关联性, 本文设计了一个名为 ECOCCJ48 的多标签分类器, 它使用二元关联 (binary relevance) 为不同编译器家族预测标签, 并使用分类器链 (classifier chains) 为不同优化级别预测标签.

如图 6(a) 所示, 编译器级别的分类器为 GCC 和 Clang 编译器家族分别设计了对应的优化级别分类器, 这两个分类器分别独立训练 (二元关联). 由于同一编译器家族的编译器做出类似的内联决策, ECOCCJ48 整合了同一编译器家族下的函数调用, 标签数量将从 32 减少到 8 (2 种编译器家族×4 种优化级别).




(a) 编译器级别 (b) 优化选项级别

图 6 ECOCCJ48 架构图

在每个优化级别分类器中, 标签在不同优化级别间存在序列依赖, 如图 6(b) 所示. 例如, O2 中进行内联函数调用预测的输入是函数调用特征及已为 O0 和 O1 预测的标签. 考虑到在 O0 和 O1 中进行的内联决策通常也会出现在 O2 中, ECOCCJ48 的架构可以利用优化间的内联关联性来产生更准确的预测结果.

此外, O2NMatcher 还使用集成方法 (ensemble method) 来增强分类器的性能. 在性能方面, 集成学习方法要优于单模型学习方法[23]. O2NMatcher 首先在随机选取的训练数据集上训练基分类器, 然后通过聚合基分类器的预测来预测标签. 由于基本分类器可以在不同的语料库上进行训练, 它们能够捕捉到一些稀有的内联模式.

#### 4.2.2 模型预测

在模式测试阶段, 给定一个开源软件项目, O2NMatcher 首先提取所有函数调用并构建其函数调用图 (function call graph, FCG). 然后, 对于 FCG 中的每个函数调用, O2NMatcher 提取相应的特征. 将这些特征作为输入到 ECOCCJ48 中, 最后得到每个 O2NMatcher 在所有编译设置下的标签.

例如, 当 O2NMatcher 在两种编译器 (GCC 和 Clang) 和 4 种优化级别 (O0、O1、O2、O3) 上训练模型时, 开源软件项目中的每个函数调用将有 8 个内联标签, 每个标签对应一个编译设置.

获取每个函数调用的标签后, O2NMatcher 创建与 8 种编译设置相对应的 8 个 FCG. 然后, O2NMatcher 在这些 8 个 FCG 中标记内联的函数调用. 最后, 将得到 8 个有标签的 FCG 用于进一步的源代码函数集合生成.

### 4.3 源代码函数集合生成

在获得标记的 FCG 后, 需要在其中生成相应的源代码函数集合. 图 7(a) 展示了一个有标签的 FCG 示例. 图中用红色边表示内联调用, 蓝色边表示普通调用. 红色圆圈代表有内联调用的函数, 黑色圆圈代表仅有普通调用的函数.

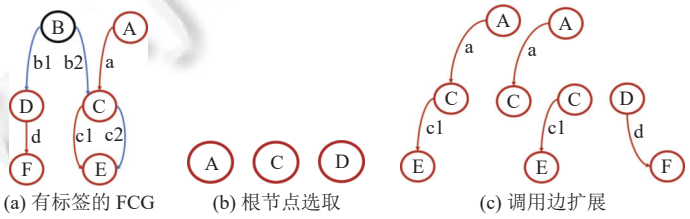



图 7 源代码函数集合生成示例

在 GCC 和 Clang 的函数内联实施中, 当启发式规则决定对给定的调用图边进行内联时, 被调用者的节点会被克隆以表示由内联器稍后生成的新函数副本, 因此, 所有被内联的函数实际上是被克隆之后添加到内联该函数的主函数中, 因此 O2NMatcher 需要将每个内联函数添加至主函数中, 以生成一个由主函数和内联函数组成的新函数.

另外, 如果一个仅被调用一次的函数被内联了, 那么该函数将不会作为独立的函数在二进制文件中被编译. 这表明在函数内联过程, 还会删除一些函数, 而这些函数将不能作为主函数内联更多函数.

参照上述 GCC 和 Clang 中的函数内联过程, 并且考虑到函数内联的过程就是将内联函数调用所连接的函数重新组合成一个函数的过程, 本文将源代码函数集合生成的过程转化为包含根节点选择和边扩展的内联函数调用图的生成问题. 根节点选择目的在于选择可以作为主函数的源代码函数节点, 而边扩展则是将每个内联函数添加至主函数中, 以生成对应的源代码函数集合.

#### 4.3.1 根节点选择

首先, 本文将内联子图定义为由内联调用及其关联函数构成的 FCG 的子图. 例如, 在图 7(a) 中有两个内联子图, 分别是 (D, F) 和 (A, C, E). 简单来说, 只要内联子图中的节点能满足以下两个条件之一, 它就可以作为根节点来生成子树.

(1) 节点是内联子图的根节点.

(2) 节点是非根节点, 它有到其他节点的内联调用, 并且还有其他节点通过普通调用连接到它.

如图 7(b) 所示, 使用规则 (1), 可以识别出两个节点 A 和 D. 应用规则 (2), 可以识别出一个节点 C. 其中, A 和 D 分别是它们各自内联子图的根节点, 因此内联将从这些位置开始.C 还被正常函数 B 调用, 因此不会被删除, 所以需要保存其副本以生成一个新的内联树.

如果一个节点不满足这两个条件, 它要么是没有向其他节点发出内联边的节点 (比如图 7(a) 中的 B、E 和 F), 要么是在内联子图中但仅被内联边调用的节点 (如图中的 F). 前者没有可以内联到源代码函数集合的节点, 而后者则会始终被内联到它的调用者中, 并且不会作为独立的函数在二进制文件中被编译.

而在图 2 的例子中, O2NMatcher 所识别的内联函数子图只包含了 dtls1_get_record dtls1_process_buffered_records, dtls1_get_bitmap, dtls1_record_replay_check 这 4 个函数, 函数 dtls1_get_record 对其他 3 个函数存在内联调用关系, 因此只有函数 dtls1_get_record 可以作为根节点.

#### 4.3.2 调用边扩展

调用边扩展同样遵循两条规则.

(1) 如果调用者和被调者之间只有内联边, 则遍历内联子图中根节点可达的所有节点.

(2) 如果调用者与被调者之间同时存在内联边和普通边, 则生成两种版本的源代码函数集合. 一种沿着内联边继续包含后续节点, 另一种在遇到普通边时停止.

以 D 为根节点时, 子树中只有内联边, 因此根节点 D 进行边扩展后生成源代码函数集合“D→F”. 使用 A 为根节点时, 节点 C 和 E 之间存在 c1 内联边和 c2 普通边两种边, 于是为根节点 A 生成源代码函数集合 “A→C→E”和“A→C”, 如图 7(c) 所示. 对于根节点 C, 则会生成源代码函数集合“C→E”.

如果内联子图中存在循环, O2NMatcher 会记录已遍历的节点并在遇到已记录的节点时停止探索该路径.

而在图 2 的例子中, 4 个函数之间只存在内联边, 因此通过上述步骤, 可以生成源代码函数集合 (dtls1_get_record + dtls1_process_buffered_records + dtls1_get_bitmap + dtls1_record_replay_check), 以用于后的内联二进制函数匹配.

得到源代码函数集合后, O2NMatcher 使用 Understand 进行源代码函数内联, 针对每个源代码函数集合生成最终含内联的源代码函数, 以作为内联二进制函数匹配的目标. 需要注意的是, 源代码函数集合的生成是在进行二进制到源代码函数相似性检测之前进行的, 在生成过程中并不需要二进制函数的相关信息.

## 5 实验分析

本节主要对 O2NMatcher 进行相关实验评估. 第 5.1 节介绍实验设置. 第 5.2 节对 O2NMatcher 的检测精度进行衡量. 第 5.3 节对 O2NMatcher 的运行时效进行分析. 第 5.4 节对 O2NMatcher 选取的特征进行贡献度的分析. 第 5.5 节对实验结果进行讨论.

### 5.1 实验设置

#### 5.1.1 评估基准

鉴于 O2NMatcher 旨在作为现有二进制到源代码函数匹配研究的补充, 本文需要选定一项研究作为二进制到源代码相似性匹配的基准方法来评估 O2NMatcher 的提升效果. 目前存在多种二进制到源代码相似性检测的方法[1−11], 其中, CodeCMR 达到了最准确的函数级匹配精度. 此外, CodeCMR 还提供了一个名为 BinaryAI 的工具, 方便本文进行深入的评估工作.

#### 5.1.2 对比方法

在 O2NMatcher 的实验衡量中, 本文选取了 Bingo[20]和 Asm2Vec[21]作为对比方法. Bingo 和 Asm2Vec 采用手工设计的规则来模拟函数内联的过程, 然后比较内联后的函数以获取相似度. 由于他们旨在解决函数内联下的二进制到二进制匹配问题, 本文迁移了它们的规则来为二进制到源代码相似性检测生成源代码函数集合.

在 ECOCCJ48 的实验衡量中, 本文选取了 5 种 MLC 方法作为对比方法. Bogatinovski[24]对目前已有的多标签

分类方法进行比较研究, 并总结了 5 种最先进的 (state-of-the-art, SOTA) 方法: RFPCT[25]、RFDTBR[26]、ECCJ48[27]、EBRJ48[27]和 AdaBoost[28]. 其中, ECCJ48 同样使用了分类器链的结构来进行内联函数调用预测, 但这些标签间的依赖关系是随机定义的.

5.1.3 参数设置

实验采用十折交叉验证来评估 O2NMatcher 的有效性. 具体来说, 从数据集中随机选取 90% 的项目作为训练集, 剩余 10% 作为测试集. O2NMatcher 中的分类器在训练集上训练并为测试集产生源代码函数集合.

测试集的二进制文件被剥离符号信息后, O2NMatcher 即使用 BinaryAI 将二进制函数与源代码函数及源代码函数集合匹配. 在实验过程中, O2NMatcher 仅采用 BinaryAI 发布的模型而不进行重新训练. 这一过程重复 10 次, 并计算平均指标.

在 ECOCCJ48 的基分类器 J48 设置中, 决策树的默认最大深度设置为无限大, 这与 RFDTBR, ECCJ48, EBRJ48 的默认设置一致. AdaBoost 也使用了其默认设置, 其决策树深度为 1. 而 RFPCT 则是使用了其论文提供的默认设置.

5.1.4 评估指标

实验采用 *Recall*@1 来衡量 O2NMatcher 的效果, 它在现有二进制到源代码相似性检测研究中被广泛应用. *Recall*@1 表示在返回的第 1 个源函数中找到匹配函数的比例.

为评估 O2NMatcher 的成本, 本文引入了一个“集合增加比例”的指标, 计算生成的源代码函数集合数量与原始源函数数量之比. 源代码函数集合增加会扩大语料库规模, 因此会增加查询时间, 并降低匹配性能.

为了评估 ECOCCJ48 的效果, 本文使用了多标签分类中常用的 3 个指标: 准确率、召回率和 $F1$ 分数. 具体度量指标如表 3 所示. 这些指标在多标签分类评估中与单标签分类相似, 可以通过取所有标签的平均值得到.

表 3 多标签分类相关度量指标

| 名词 | 缩写 | 定义 |
|---|---|---|
| 真阳例 | *TP* | 被正确检测为内联调用的内联调用的数量 |
| 真阴例 | *TN* | 被正确检测为普通调用的普通函数的数量 |
| 假阴例 | *FN* | 被错误检测为普通调用的内联调用的数量 |
| 假阳例 | *FP* | 被错误检测为内联调用的普通调用的数量 |
| 误报率 | *FPR* | $FP/(FP+TN)$ |
| 漏报率 | *FNR* | $FN/(TP+FN)$ |
| 准确率 | *P* | $TP/(TP+FP)$ |
| 召回率 | *R* | $TP/(TP+FN)$ |
| $F1$分数 | $F1$ | $2PR/(P+R)$ |

5.1.5 具体实现

在数据集标记过程中, O2NMatcher 利用 Understand 解析源代码项目, 通过 IDA Pro 反汇编二进制文件. 在函数调用图构建时, 同样使用 Understand 构造源代码 FCG, 使用 IDA Pro 构建二进制 FCG. 在函数调用特征提取上, 使用 tree-sitter 提取调用函数/被调用函数中的相关特征及调用指令特征, 并使用 Understand 提取函数调用特征. 模型训练阶段, 使用 Python 工具包 scikit-multilearn 实现 ECOCCJ48 及其他多标签分类方法. 论文的数据集和源代码可在 https://github.com/island255/binary2source-matching-under-function-inlining 获取.

整个程序以 Python 编写, 并在配备了 Ubuntu 18.04 系统、Intel Xeon Gold 6266C 处理器和 1 024 GB DDR4 RAM 的工作站上进行所有实验.

### 5.2 检测精度分析

本节将对 O2NMatcher 进行检测精度的分析. 由于 O2NMatcher 中包含了一个内联函数调用预测的方法 ECOCCJ48, 其中 ECOCCJ48 的分类精度在很大程度上决定了 O2NMatcher 的检测精度, 因此本节将分别对

O2NMatcher 和 ECOCCJ48 的检测精度进行实验评估.

5.2.1 O2NMatcher 的检测精度分析

在 O2NMatcher 的检测精度评估中, 首先衡量了 O2NMatcher 与 BinaryAI 在函数内联场景下的二进制到源代码相似性搜索的效果, 以评估 O2NMatcher 对现有“一对一”匹配工作的提升效果. 随后, 本节将 O2NMatcher 和两个内联策略方法 Bingo 和 Asm2Vec 对比, 以评估 O2NMatcher 的有效性. 最后, 本节将生成的源代码函数集合与内联的二进制函数进行了对比, 分析 O2NMatcher 在寻找内联源代码函数上的准确率.

表 4 展示了 BinaryAI 和 O2NMatcher 在匹配内联二进制函数时的实验效果. 首先, 当对比 O2NMatcher 与 BinaryAI 的检测效果时, O2NMatcher 在检测内联的二进制函数时给 BinaryAI 带来了显著的提升, *Recall*@1 由 34.50% 上升到 40.93%. 在函数内联场景下, O2NMatcher 对 BinaryAI 在 *Recall*@1 上的提升幅度为 (40.93%–34.50%)/34.50%=18.6%. 表 4 中也展示了 Bingo 和 Asm2Vec 的匹配效果, 可以看到, Bingo 和 Asm2Vec 只能使 BinaryAI 的性能分别提高了 0.98% 和 0.46%.

表 4 O2NMatcher 的实验结果 (%)

| 指标 | BinaryAI | Bingo | Asm2Vec | O2NMatcher |
|---|---|---|---|---|
| *Recall*@1 | 34.50 | 34.84 | 34.65 | 40.93 |
| 集合增加比例 | 0.00 | 23.78 | 22.09 | 87.63 |

通过结果可以看出, Bingo 和 Asm2Vec 人工设计的规则很难适应多种分类器, 多种优化选项下的函数内联情况, 而 O2NMatcher 通过识别内联函数调用并进一步模拟内联实施构建源代码函数集合的方法能够更好拟合各种编译设置下的函数内联情况. 这是因为, 在 O2NMatcher 的内联预测部分, 通过对多种编译条件下的函数内联规律进行归纳和整合, O2NMatcher 所设计的多标签分类方法 ECOCCJ48 能够学习更多编译选项的函数内联规律, 从而识别更多的内联函数调用, 最终构建出更为完整和准确的源代码函数集合. 相比 Bingo 和 Asm2Vec, 由于 O2NMatcher 能够生成与内联二进制函数更为接近的源代码函数集合, 因此其在二进制到源代码函数相似性匹配场景下能够实现更为精确的匹配效果.

尽管源代码函数集合能够帮助提升现有二进制到源代码相似性检测工作的性能, 但源代码函数集合也会增加查询二进制函数的匹配时间. 如表 4 所示, 对于含有内联的二进制函数, O2NMatcher 需要额外生成 87.63% 的源代码函数集合. 考虑到内联的二进制函数仅占二进制函数的一部分, 由于源代码函数集合所增加的匹配时间也是可以接受的.

接下来, 本节评估了 O2NMatcher 在寻找内联源代码函数上的准确率. 本文利用发生内联的二进制函数与其对应源代码函数集合之间的相似度来衡量针对特定发生内联的二进制函数所找到的源代码函数的完整性. 具体来说, 本文采用公式 (2) 计算这一相似度:

$$Sim_{BFI\text{-}SFS}=\max_{SFSs}\left(\frac{|BFI\cap SFS|\times 2}{|BFI|+|SFS|}\right) \tag{2}$$

其中, *BFI* 代表含有内联的二进制函数, *SFS* 表示源代码函数集合. $|BFI\cap SFS|$ 表示同时出现在 *BFI* 和 *SFS* 中的源函数数量. $|BFI|$ 和 $|SFS|$ 分别代表发生内联的二进制函数映射到的源函数数量以及生成的源代码函数集合中的源函数数量. max 意味着从属于 *BFI* 根函数生成的所有 *SFS* 中选择最大相似度. 如果给定的 *BFI* 没有生成任何 *SFS*, 此相似度将为 0.

图 8 展示了所有内联二进制函数与源代码函数集合之间的相似度的分布情况. 如图所示, 超过 50% 的内联二进制函数能找到相似度超过 80% 的源代码函数集合. 33.5% 的内联二进制函数能找到完全匹配的源代码函数集合. 这说明对于内联的二进制函数来说, 其中 50% 的二进制函数能够找到参与其编译过程的 80% 的源代码函数, 33.5% 的二进制函数能够找到生成该二进制函数的所有源代码函数. 因此, O2NMatcher 生成的源代码函数集合能帮助已有工作为内联的二进制函数找到更多其复用的源代码函数.

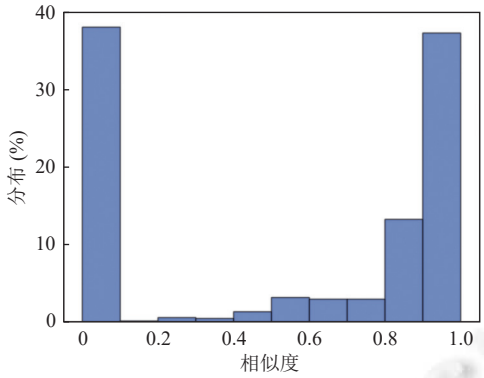



图 8　内联二进制函数与源代码函数集合之间相似度的分布

5.2.2　ECOCCJ48 的检测精度分析

在 ECOCCJ48 的检测精度评估中, 本节选择了 5 种现有的 MLC 方法作为对比, 并使用了准确率、召回率和 *F*1 分数等衡量指标, 对 ECOCCJ48 在内联函数调用预测这一任务上进行了实验评估.

图 9 展示了针对内联函数调用预测的评估结果. 横轴代表基分类器的数量, 纵轴代表相应的衡量指标. 对于精确率、召回率和 *F*1 分数, 数值越高越好. 对于除 AdaBoost 之外的大多数 MLC 方法, 随着基分类器数量的增加, 这些指标也会提高. AdaBoost 通过增加额外的基分类器来适应错误分类的样本, 当基分类器数量增多时, AdaBoost 会对训练集过度拟合, 从而对测试集的泛化能力减弱.

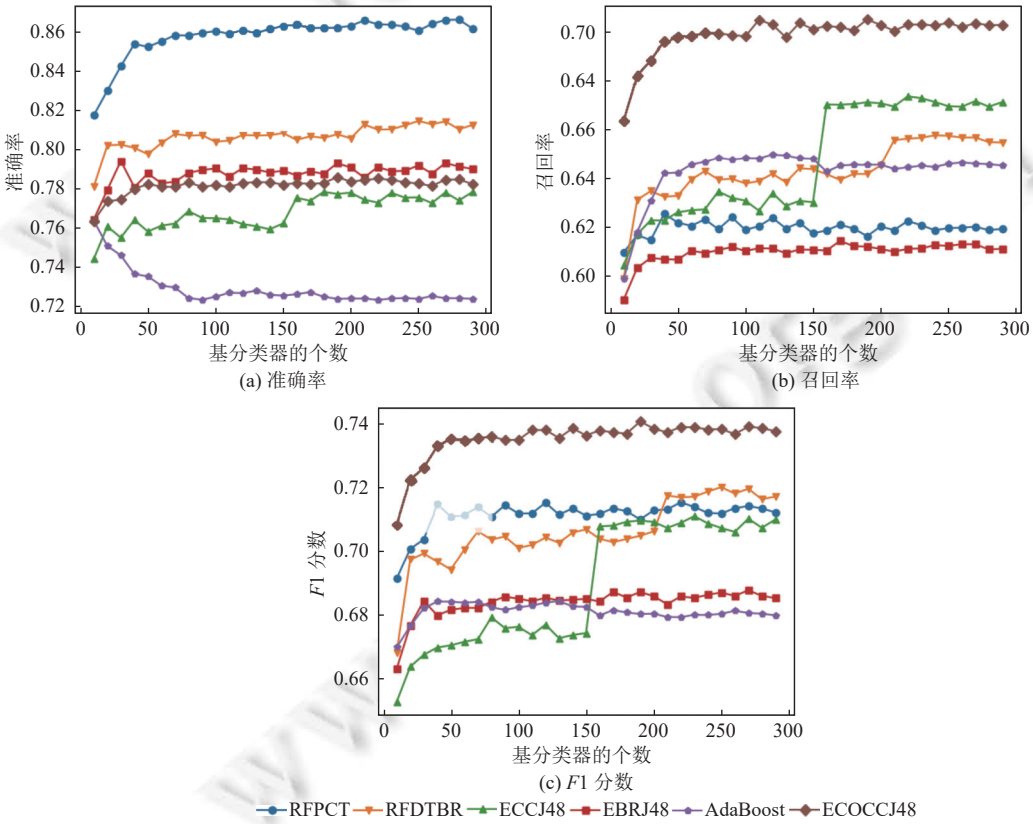



图 9　ECOCCJ48 和其他多标签分类方法的衡量结果

总的来说, 当使用最佳的基分类器数量 (AdaBoost 为 50, 其余为 300) 时, ECOCCJ48 在所有方法中获得了最高的 $F1$ 分数. 虽然 RFPCT、RFDTBR 和 EBRJ48 等方法在精确率上更高, 但 ECOCCJ48 在召回率方面表现更优. 总体而言, ECOCCJ48 能够实现 78% 的准确率和 70% 的召回率, 最终能够实现 0.74 的 $F1$ 分数, 领先所有其他方法.

ECOCCJ48 的卓越性能主要得益于其基于内联规律的分类器链结构设计. 相比于 RFDTBR、EBRJ48、AdaBoost 等不利用标签依赖关系的方法, ECOCCJ48 通过引入标签依赖关系, 有效地分解了内联规则的学习任务. 高优化选项下的内联规则通常是在低优化选项基础上增加了一些新规则, 而 ECOCCJ48 中从高优化到低优化的标签依赖关系使其在学习高优化选项时能够利用已从低优化选项中学到的规则. 因此, 在高优化选项下, ECOCCJ48 只需专注于学习新增的规则即可. 相比之下, RFDTBR、EBRJ48 和 AdaBoost 模型在学习高优化选项时缺乏低优化选项的知识积累, 因此在高优化选项下的内联规则学习中面临较大困难.

此外, 尽管 RFPCT 和 ECCJ48 也在不同优化选项间采用了分类器链模型, 但它们并未精确地利用优化选项之间的依赖关系, 因而无法达到与 ECOCCJ48 相同的分类性能. 由于内联规则是从低优化到高优化逐步叠加形成的, 采用逆序或无关的顺序构建的分类器链模型无法有效捕捉这一顺序依赖规律.

总体来看, ECOCCJ48 能够发现更多隐藏的内联函数调用, 并能迅速捕捉内联模式.

### 5.3 时间开销分析

O2NMatcher 在实施过程需要首先训练内联函数调用模型, 然后使用模型进行预测, 最后根据内联标签生成函数集合. 因此, 本节记录了包括模型训练、模型测试和源代码函数集合生成 3 个不同阶段的运行时间, 进一步评估了 O2NMatcher 的时间开销.

图 10(a) 展示了 ECOCCJ48 及其相关 MLC 方法的训练时间. 横轴表示基分类器的数量, 纵轴表示以 s 为单位的训练时间. 在所有 MLC 方法中, AdaBoost 通常只会所选取一层的决策树作为基分类器, 而 ECOCCJ48, ECCJ48, EBRJ48 和 RFDTBRAdaBoost 的基分类器通常为多层的 J48 决策树, 因此 AdaBoost 的模型训练时间最短, 少于 100 s. ECOCCJ48 的训练时间与 ECCJ48, EBRJ48 和 RFDTBR 相近, 四者均需约 1 000 s 来训练模型. 而 RFPCT 使用树集合方法进行训练, 其树的深度更深, 因此 RFPCT 的训练时间最长, 超过 10 000 s. 考虑到 ECOCCJ48 模型只需训练一次且能够离线训练, ECOCCJ48 的训练时间成本是可以接受的.

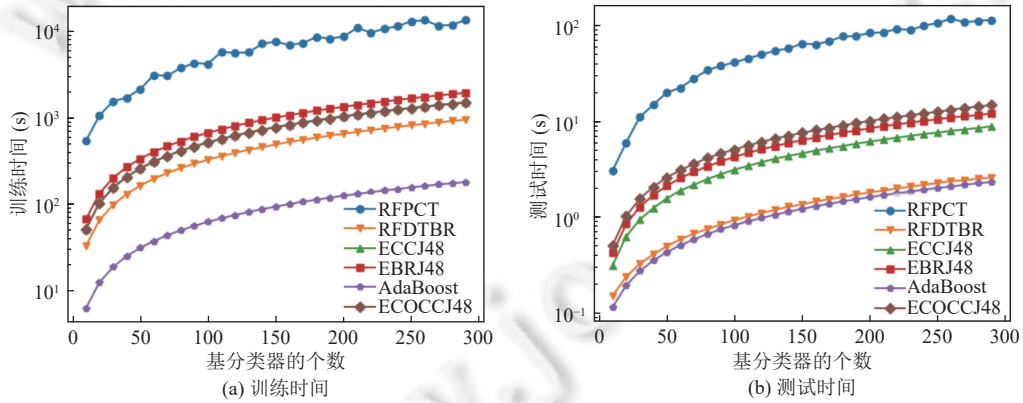



图 10 O2NMatcher 和相关工作的训练时间和测试时间

图 10(b) 显示了 ECOCCJ48 及其相关 MLC 方法的预测时间. 和训练时间类似, AdaBoost 的测试时间最快, ECOCCJ48 与 ECCJ48, EBRJ48 和 RFDTBR 随后, RFPCT 的测试时间最长. 不同的是, 由于测试过程只需要将样本输入到分类器中, 不需要改变参数, 因此多数模型仅需几秒钟即可预测项目中的所有内联函数调用. 例如, 由 50 个基分类器训练的 AdaBoost 仅需 0.4 s, 由 300 个基分类器训练的 ECOCCJ48 则需要 15.2 s. 需要注意的是, 当仅使用 50 个基分类器时, ECOCCJ48 就能达到相对较高的性能, 此时预测仅需 2.6 s.

此外, 本节也记录了 O2NMatcher 的源代码函数集合生成时间. 通常, 为一个项目生成所有源代码函数集合所需的时间不超过 10 s. 而且, 源代码函数集合生成过程可以在后台执行, 并且源代码函数集合只需在执行二进制到源代码函数匹配前只需生成一次即可.

### 5.4 特征贡献度分析

在第 4.1 节中, O2NMatcher 为内联函数调用预测选择了几个特征. 本节将分析这些特征的贡献, 以揭示 ECOCCJ48 是如何决定进行内联操作的.

图 11 可视化了 ECOCCJ48 中学习到的部分分类规则, 在被调用者特征中, Static 关键字在前几层的内联调用预测中起着重要作用. 可以看出, 当被调用函数被 Static 关键字定义时, 该函数更倾向于被内联. 这是因为 Static 关键字定义的函数通常只在一个源文件范围内可见, 这意味着它们不会成为链接阶段的负担, 并且它们的内联不会增加其他模块的代码大小. 此外, Static 函数通常包含较简单的逻辑, 因此更适合内联, 以减少函数调用开销并提高执行效率.

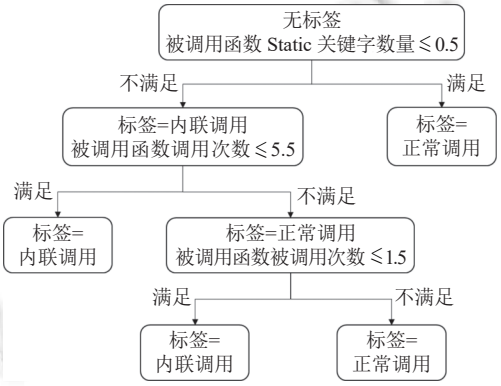



图 11 内联函数调用预测的决策树规则

另外, 函数调用图中的函数调用特征在分类内联函数调用时也扮演着重要角色. 例如, “被调用函数被调用次数≤1.5”意味着仅被调用一次的函数应被内联. 这条规则反映了这样的事实: 如果一个函数只被调用了一次, 那么内联该函数可能会减少函数调用开销, 从而提高程序的执行速度. 在这种情况下, 内联并不会显著增加代码大小, 因此是一个合理的选择.

此外, “被调用函数调用次数≤5.5”表明如果一个被调用函数自身又调用了很少的其他函数 (即该函数本身不是复杂的调度者), 那么它更有可能被内联. 这样的函数通常具有较为简单的逻辑, 内联它可以减少函数调用的开销, 而不会显著增加代码的大小. 换句话说, 如果一个被调用函数本身调用了多个其他函数, 那么它可能是一个复杂的调度函数. 这样的函数如果被内联, 可能会导致代码膨胀, 并且不一定会带来显著的性能提升. 因此, 当一个函数只调用了少数几个其他函数时, 它更有可能作为一个功能函数被内联到调用它的函数中去. 这样不仅可以减少函数调用的开销, 还能简化代码逻辑.

### 5.5 结果讨论

#### 5.5.1 源代码函数集合的覆盖率分析

O2NMatcher 虽然能够生成大量的源代码函数集合以供内联二进制函数比对, 但生成的源代码函数集合仍存在部分缺失. 如图 8 所示, 有 38% 的内联二进制函数找不到生成的源代码函数集合. 这些遗漏主要归因于构建的源代码函数调用流图的不完整性. 在 C/C++中, 预处理器指令被广泛使用, 其中条件编译指令 (如#ifdef、#if、#elif、#else、#endif) 会导致编译区域的可变性[29]. 程序可能会根据不同的编译环境将不同的源代码部分编译成二进制. 覆盖所有可能的编译结果较为困难. 因此, 大多数现有的 C/C++静态分析器无法确定哪些代码参与了编译. 编译区域的错误识别将导致构建的 FCG 不完整且不准确, 这进一步影响了生成的源代码函数集合的质量.

例如, aoGetsText 是 Sharutils-4.15.2 中的一个源函数, 它位于一个#if 指令下. 尽管它已被内联到许多二进制函数中, 但像 Understand 这样的现有静态分析工具却认为这部分代码不会参与编译. 结果, 构建的 FCG 遗漏了函数 aoGetsText. 因此, 由于缺少源函数 aoGetsText, O2NMatcher 无法创建相应的源代码函数集合.

5.5.2 源代码函数集合的负面效果分析

正如上文所述, 源代码函数集合所带来的第 1 个负面影响是匹配时间的增加. 查询的二进制函数不仅要与源函数比较, 还要与源代码函数集合比较, 这增加了匹配的成本. 然而, O2NMatcher 需要为普通函数生成的源代码函数集合仅占 26.81%, 这只会引起较小的匹配开销.

源代码函数集合带来的另一个负面影响是在一定程度上影响了普通二进制函数的匹配效果 (*Recall*@1 从 91.2% 下降到 89.5%). 但是, 当汇总普通函数和内联函数的结果时, 整体性能实际上是提高的 (*Recall*@1 从 72.8% 增加到了 73.8%).

5.5.3 内联规模对 O2NMatcher 的影响

在对 O2NMatcher 衡量过程中, 本文发现, 在不同内联程度的二进制函数中, O2NMatcher 的性能表现也不同. 相比于内联了数十上百个函数的二进制函数, O2NMatcher 更容易为内联了数个函数的二进制函数生成对应的源代码函数集合. 这是因为, 在内联了数个函数的二进制函数, O2NMatcher 只需要预测出数个内联的函数调用即可生成准确的源代码函数集合. 而在内联了数十上百个函数的二进制函数中, O2NMatcher 很难预测正确所有的内联函数调用, 因此, O2NMatcher 在匹配大规模内联的二进制函数时仍面临着一些挑战.

## 6 总 结

针对内联场景下一对多匹配的二进制到源代码函数相似性检测问题, 本文提出了 O2NMatcher, 通过生成源代码函数集合作为内联场景下二进制函数的匹配对象, 以弥补源函数中匹配对象缺失的问题. 为了构建源代码函数集合, 本文首先标注了内联数据集中函数调用的内联情况, 然后抽取了对应函数调用的属性特征和调用特征, 并且设计了一种多标签分类方法用于内联函数调用预测. 再后, 对于用于组成成分分析的源代码项目, 使用已有的多标签分类模型预测内联的函数调用. 最后, 基于内联的函数调用, 本文提出了一种基于函数调用树的构建方法完成了源代码函数集合的生成. 本文通过几项实验评估了 O2NMatcher, 结果表明 O2NMatcher 能够使现有二进制到源代码相似性检测工作在检测内联函数方面取得 18.6% 的提升.

在未来工作中, 一方面, 将继续致力于实现更为准确的源代码依赖关系解析, 以实现更为全面的函数调用关系抽取, 来提升 O2NMatcher 进行内联函数调用预测的准确性. 另一方面, 函数内联给包括但不限于漏洞检测, 第三方库复用检测等二进制相似性检测的各种场景都带来了对应的挑战, 未来将针对这些挑战提出对应的解决方法.

**References:**

[1] Hemel A, Kalleberg KT, Vermaas R, Dolstra E. Finding software license violations through binary code clone detection. In: Proc. of the 8th Working Conf. on Mining Software Repositories. Honolulu: ACM, 2011. 63–72.

[2] Rahimian A, Charland P, Preda S, Debbabi M. RESource: A framework for online matching of assembly with open source code. In: Proc. of the 5th Int'l Symp. on Foundations and Practice of Security. Montreal: Springer, 2012. 211–226. [doi: 10.1007/978-3-642-37119-6_14]

[3] Kim D, Cho S, Han S, Park M, You I. Open source software detection using function-level static software birthmark. Journal of Internet Services and Information Security, 2014, 4(4): 25–37. [doi: 10.22667/JISIS.2014.11.31.025]

[4] Miyani D, Huang Z, Lie D. BinPro: A tool for binary source code provenance. arXiv:1711.00830, 2017.

[5] Duan RA, Bijlani A, Xu M, Kim T, Lee W. Identifying open-source license violation and 1-day security risk at large scale. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 2169–2185. [doi: 10.1145/3133956.3134048]

[6] Feng MY, Mao WX, Yuan ZM, Xiao Y, Ban G, Wang W, Wang SY, Tang Q, Xu JH, Su H, Liu BH, Huo W. Open-source license violations of binary software at large scale. In: Proc. of the 26th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Hangzhou: IEEE, 2019. 564–568. [doi: 10.1109/SANER.2019.8667977]

[7] Yuan ZM, Feng MY, Li F, Ban G, Xiao Y, Wang SY, Tang Q, Su H, Yu CD, Xu JH, Piao AH, Xuey J, Huo W. B2SFinder: Detecting

open-source software reuse in COTS software. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). San Diego: IEEE, 2019. 1038–1049. [doi: 10.1109/ASE.2019.00100]

[8] Ban G, Xu LL, Xiao Y, Li XH, Yuan ZM, Huo W. B2SMatcher: Fine-grained version identification of open-source software in binary files. Cybersecurity, 2021, 4(1): 21. [doi: 10.1186/s42400-021-00085-7]

[9] Ji YD, Cui L, Huang HH. BugGraph: Differentiating source-binary code similarity with graph triplet-loss network. In: Proc. of the 2021 ACM Asia Conf. on Computer and Communications Security. Hong Kong: ACM, 2021. 702–715. [doi: 10.1145/3433210.3437533]

[10] Gui Y, Wan Y, Zhang HY, Huang HF, Sui YL, Xu GD, Shao ZY, Jin H. Cross-language binary-source code matching with intermediate representations. In: Proc. of the 2022 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Honolulu: IEEE, 2022. 601–612. [doi: 10.1109/SANER53432.2022.00077]

[11] Yu ZP, Zheng WX, Wang JQ, Tang QY, Nie S, Wu S. CodeCMR: Cross-modal retrieval for function-level binary source code matching. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: ACM, 2020. 326.

[12] Jia A, Fan M, Jin WX, Xu X, Zhou ZH, Tang QY, Nie S, Wu S, Liu T. 1-to-1 or 1-to-n? Investigating the effect of function inlining on binary similarity analysis. ACM Trans. on Software Engineering and Methodology, 2023, 32(4): 87. [doi: 10.1145/3561385]

[13] Theodoridis T, Grosser T, Su ZD. Understanding and exploiting optimal function inlining. In: Proc. of the 27th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2022. 977–989. [doi: 10.1145/3503222.3507744]

[14] Damásio T, Pacheco V, Goes F, Pereira F, Rocha R. Inlining for code size reduction. In: Proc. of the 25th Brazilian Symp. on Programming Languages. Joinville: ACM, 2021. 17–24. [doi: 10.1145/3475061.3475081]

[15] Gupta P, Jha A, Gupta B, Sumpi K, Sahoo S, Chalapathi MMV. Techniques and trade-offs in function inlining optimization. EAI Endorsed Trans. on Scalable Information Systems, 2024, 11(4): 1–7. [doi: 10.4108/eetsis.4453]

[16] Weingarten ME, Theodoridis T, Prokopec A. Inlining-benefit prediction with interprocedural partial escape analysis. In: Proc. of the 14th ACM SIGPLAN Int'l Workshop on Virtual Machines and Intermediate Languages. Auckland: ACM, 2022. 13–24. [doi: 10.1145/3563838.3567677]

[17] Ben-Asher Y, Faour N, Shinaar O. Mutual inlining: An inlining algorithm to reduce the executable size. In: Proc. of the 2022 CS & IT Conf. 2022. 1–16. [doi: 10.5121/csit.2022.120601]

[18] Muts K, Falk H. Multi-criteria function inlining for hard real-time systems. In: Proc. of the 28th Int'l Conf. on Real-time Networks and Systems. Paris: ACM, 2020. 56–66. [doi: 10.1145/3394810.3394819]

[19] Romano A, Wang WH. When function inlining meets WebAssembly: Counterintuitive impacts on runtime performance. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 350–362. [doi: 10.1145/3611643.3616311]

[20] Chandramohan M, Xue YX, Xu ZZ, Liu Y, Cho CY, Tan HBK. BinGo: Cross-architecture cross-os binary search. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 678–689. [doi: 10.1145/2950290.2950350]

[21] Ding SHH, Fung BCM, Charland P. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: Proc. of the 2019 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2019. 472–489. [doi: 10.1109/SP.2019.00003]

[22] Kim D, Kim E, Cha SK, Son S, Kim Y. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. IEEE Trans. on Software Engineering, 2023, 49(4): 1661–1682. [doi: 10.1109/TSE.2022.3187689]

[23] Moyano JM, Gibaja EL, Cios KJ, Ventura S. Review of ensembles of multi-label classifiers: Models, experimental study and prospects. Information Fusion, 2018, 44: 33–45. [doi: 10.1016/j.inffus.2017.12.001]

[24] Bogatinovski J, Todorovski L, Džeroski S, Kocev D. Comprehensive comparative study of multi-label classification methods. Expert Systems with Applications, 2022, 203: 117215. [doi: 10.1016/j.eswa.2022.117215]

[25] Kocev D, Vens C, Struyf J, Džeroski S. Tree ensembles for predicting structured outputs. Pattern Recognition, 2013, 46(3): 817–833. [doi: 10.1016/j.patcog.2012.09.023]

[26] Tsoumakas G, Katakis I. Multi-label classification: An overview. Int'l Journal of Data Warehousing and Mining, 2007, 3(3): 1–13. [doi: 10.4018/jdwm.2007070101]

[27] Read J. Scalable multi-label classification [Ph.D. Thesis]. Hamilton: University of Waikato, 2010.

[28] Schapire RE, Singer Y. Improved boosting algorithms using confidence-rated predictions. In: Proc. of the 11th Annual Conf. on Computational Learning Theory. Madison: ACM, 1998. 80–91. [doi: 10.1145/279943.279960]

[29] Kenner A, Kästner C, Haase S, Leich T. TypeChef: Toward type checking #ifdef variability in C. In: Proc. of the 2nd Workshop on Feature-oriented Software Development. Eindhoven: ACM, 2010. 25–32. [doi: 10.1145/1868688.1868693]

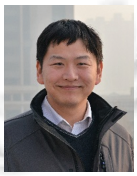
**贾昂**(1996－), 男, 博士生, CCF 学生会员, 主要研究领域为软件供应链安全.

**晋武侠**(1989－), 女, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为程序分析, 微服务, 软件架构, 软件工程.

**范铭**(1991－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为移动软件安全, 隐私保护合规性, 可解释性 AI 技术, AI 安全.

**王海军**(1983－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为软件供应链安全, 区块链安全, 程序分析, 逆向工程, 模糊测试.

**徐茜**(1996－), 女, 博士生, CCF 学生会员, 主要研究领域为漏洞检测, 软件组成成分分析.



**刘烃**(1981－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为信息物理融合系统安全, AI 软件工程.