软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(7):2808−2830 [doi: 10.13328/j.cnki.jos.007587] [CSTR: 32375.14.jos.007587]

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

CAnalyzer: 面向 C/C++源代码的软件成分分析技术*

徐美秋, 王舒婧, 王 莹, 于 海, 朱志良

(东北大学 软件学院, 辽宁 沈阳 110169)

通信作者: 王莹, E-mail: wangying@swc.neu.edu.cn

摘 要: 第三方库 (third-party library, TPL) 在软件开发中得到了广泛应用, 但也带来了安全漏洞和许可证冲突等风险. 为应对这些挑战, 软件成分分析 (software composition analysis, SCA) 技术应运而生, 旨在通过识别和分析软件中使用的开源组件及其依赖关系, 帮助开发者检测安全漏洞、过时补丁和许可证合规性问题, 确保软件供应链的安全. 然而, 现有 SCA 工具在 C/C++领域存在 3 大局限: 缺乏全面的 TPL 特征库、难以识别库粒度复用、对 TPL 依赖关系的分析能力不足. 针对上述问题, 提出 C/C++源代码软件成分分析技术——CAnalyzer, 用于软件库粒度的复用检测场景. CAnalyzer 通过整合 15 个平台的数据, 构建了一个包含 33 100 个 TPL 和 30 047 290 个函数的特征库, 并通过特征库预处理与多重阈值匹配策略, 显著提升了 TPL 检测的准确性. 此外, CAnalyzer 通过解析源码中的依赖指令, 实现了 TPL 间依赖关系的自动化构建. 实验结果表明, CAnalyzer 在 TPL 检测中的精确率为 90.63%, 召回率为 86.57%, 在两项指标上均优于现有方法 CENTRIS、TPLite 和 OSSFP. 在 TPL 依赖关系检测中, CAnalyzer 的召回率和精确率分别为 94.79% 和 98.99%. 目前, CAnalyzer 已被 OpenHarmony 社区采纳, 在 689 个代码仓库中识别出 166 项外部组件, 体现了其在开源社区管理中的应用价值.

关键词: 开源软件复用; 软件成分分析; 第三方库依赖

中图法分类号: TP311

中文引用格式: 徐美秋, 王舒婧, 王莹, 于海, 朱志良. CAnalyzer: 面向C/C++源代码的软件成分分析技术. 软件学报, 2026, 37(7): 2808–2830. http://www.jos.org.cn/1000-9825/7587.htm

英文引用格式: Xu MQ, Wang SJ, Wang Y, Yu H, Zhu ZL. CAnalyzer: Software Composition Analysis Technique for C/C++ Source Code. Ruan Jian Xue Bao/Journal of Software, 2026, 37(7): 2808–2830 (in Chinese). http://www.jos.org.cn/1000-9825/7587.htm

CAnalyzer: Software Composition Analysis Technique for C/C++ Source Code

XU Mei-Qiu, WANG Shu-Jing, WANG Ying, YU Hai, ZHU Zhi-Liang

(Software College, Northeastern University, Shenyang 110169, China)

Abstract: Third-party libraries (TPLs) are widely used in software development but also introduce risks such as security vulnerabilities and license conflicts. In response, software composition analysis (SCA) has emerged to help developers detect security vulnerabilities, outdated patches, and license compliance issues by identifying and analyzing open-source components and their dependencies, thereby ensuring the security of software supply chains. However, existing SCA techniques in the C/C++ domain face three major limitations: a lack of comprehensive TPL feature libraries, difficulty in detecting library-granularity reuse, and insufficient capability to analyze TPL dependencies. To address these limitations, a SCA technique for C/C++ source code—CAnalyzer—is proposed for software library-granularity reuse detection scenarios. By integrating data from 15 platforms, CAnalyzer builds a feature database containing 33 100 TPLs with 30 047 290 functions. Meanwhile, the precision of TPL detection is significantly improved through feature database preprocessing and

* 基金项目: 国家重点研发计划 (2024YFF0908000); 中国博士后科学基金 (2024M750375); CCF 华为胡杨林基金软件工程专项 (鸿蒙专题) (CCF-HuaweiSE20240210)
本文由“智能化基础软件可信构造和供应链安全”专题特约编辑王林章教授、司徒凌云研究员、钟浩研究员、 向剑文教授、张路教授推荐.
收稿时间: 2025-09-08; 修改时间: 2025-10-20; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-26
CNKI 网络首发时间: 2026-06-02

a multi-threshold matching strategy. Additionally, CAnalyzer analyzes dependency directives in the source code to automatically construct dependencies among TPLs. Experimental results show that CAnalyzer achieves a precision of 90.63% and a recall of 86.57% in TPL detection, outperforming CENTRIS, TPLite, and OSSFP in both metrics. In TPL dependency detection, CAnalyzer achieves a recall of 94.79% and a precision of 98.99%. Currently, CAnalyzer has been adopted by the OpenHarmony community and has identified 166 external components across 689 code repositories, demonstrating its practical value in open-source community management.

Key words: open-source software reuse; software composition analysis (SCA); third-party library (TPL) dependency

1 引 言

随着开源生态的蓬勃发展, 第三方库 (third-party library, TPL) 被广泛应用于软件开发, 极大地提高了开发效率[1,2]. 然而, 被复用的第三方库可能引入安全漏洞, 从而带来潜在的安全隐患[3−7]. 为应对这一问题, 软件成分分析 (software composition analysis, SCA) 技术应运而生. SCA 技术通过分析开源软件和商业软件中的源码、模块和框架, 识别并枚举被复用的第三方库, 检测已知的安全漏洞、未修补的漏洞及许可证合规性风险等问题, 从而确保软件供应链中第三方库的安全性与合规性. 本研究聚焦于针对 C/C++语言软件的源到源 (source-to-source, S2S) 的 SCA 技术, 该技术基于源代码构建特征库, 匹配目标代码特征与已知 TPL 代码特征的相似性来识别 TPL. 已提出多种基于 S2S 的 SCA 技术[5,8−15], 然而, 随着软件开发复杂性的增加以及 C/C++语言生态中长期存在的 TPL 管理碎片化问题, 现有的 SCA 技术仍面临以下限制.

(1) 基于单一 C/C++ TPL 托管仓库构建特征库导致漏报 TPL. 在 C/C++语言生态系统中, TPL 资源分布于多个托管仓库 (如 Debian、Conan、GitLab 等), 本研究调查显示, 目前有超过 15 个活跃的托管仓库包含 C/C++ TPL. 然而, 现有的 SCA 技术[13−15]仅依赖单一托管仓库 (如 GitHub) 收集 TPL 资源, 由于无法全面覆盖所有 C/C++ TPL, 从而导致对实际项目成分分析时出现漏检. 近期对 CENTRIS 的大规模实际项目评估[14]结果显示, 当复用 TPL 的阈值设置为 1% 时 (精确率仅为 17.7%), CENTRIS 仍漏报 30.7% 的 C/C++ TPL. 因此, 基于单一托管仓库构建的特征库不足以实现全面覆盖 C/C++ TPL.

(2) 相似性计算粒度与成分报告粒度不匹配造成误报 TPL. 现有 SCA 技术[13−15]通过逐一对比目标代码与已知 TPL 中的函数相似性 (函数粒度计算相似性), 并基于相似函数的数量和占比 (相对于 TPL 总函数数) 判断是否报告复用 TPL. 然而, 基于整个 TPL 函数数量计算阈值未能考虑函数在 TPL 文件中的分布, 如果与目标代码匹配的函数实际上来自 TPL 不同文件的常用函数, 会被误判为有效成分, 引发 SCA 误报 TPL.

(3) 缺乏 TPL 之间依赖关系的分析限制安全性和合规性检测的准确性. 现有的 SCA 技术[13−15]主要关注 TPL 成分的识别, 但未深入分析 TPL 之间的依赖关系, 导致以下两个主要问题: (a) 难以全面了解漏洞在各个 TPL 之间的传播路径和影响范围, 限制了安全团队根据漏洞传播路径优先处理关键问题的能力[16−19]; (b) 无法准确识别实际依赖路径中的许可证冲突、许可证兼容性等合规性问题, 进而增加了软件合规性风险.

然而, 为了克服这些限制, 在构建全面覆盖 C/C++ TPL 的特征库、优化 TPL 检测和增强依赖关系分析的过程中, 也伴随着新的技术挑战.

(1) 融合多个托管仓库构建 C/C++ TPL 特征库和不准确的函数诞生时间, 加剧了公共函数对 TPL 检测造成的干扰. 构建全面的 C/C++ TPL 特征库的直接方法是收集并整合所有 C/C++ TPL 托管仓库中的 TPL. 然而, 若未经处理将所有特征直接保存至特征库, 公共函数的干扰性将加剧. 本文所指的公共函数包括: TPL 间复用引入的克隆函数[13], 以及实现常见算法 (如排序、查找算法等) 的相似函数. 如果这些公共函数与目标代码匹配, SCA 技术可能错误地将包含这些函数的 TPL 判定为目标代码的有效成分. 为了减少公共函数引起的干扰, 已有研究引入了函数诞生时间 (function birth time)[13,14]帮助确定函数最早出现的库, 进而明确其原始 TPL 并排除误报 TPL. 然而, 在融合多个托管仓库构建的特征库中, 函数诞生时间的准确性受到挑战: (a) 非 Git 管理的托管仓库 (如通过 FTP Server 发布源代码的 C/C++ TPL) 难以获取函数诞生时间; (b) Git 管理的托管仓库中约 20.0% 的 C/C++ TPL 缺乏 Release/Tag, 导致获取不准确的函数诞生时间; (c) C/C++ TPL 迁移至其他托管仓库时, 丢失历史版本的函数诞生时间. 因此, 融合特征库中公共函数归属原始 TPL 的确认问题仍然是一个亟待解决的挑战.

(2) TPL 复用检测面临特征粒度选择问题, 粗粒度特征误报较少但易漏报, 细粒度特征漏报较少但误报增多. 在库粒度复用检测中, 粗粒度特征 (如文件粒度) 能够减少误报, 但缺乏应对代码结构变化的灵活性, 特别是当目标代码对复用文件进行函数级别的增删或重排时, 结构变化会导致文件相似性降低, 导致漏报 TPL. 而细粒度特征 (如函数粒度) 虽然能应对代码结构变化, 但在库粒度报告时容易产生误报 (见限制 1). 为了减少误报, 通常需要提高函数复用的数量阈值, 这会导致召回率下降, 因为一些少量复用情况可能被漏检. 因此, 如何平衡细粒度特征和粗粒度报告, 保持较低的误报率的同时提高召回率, 仍是一个亟待解决的问题.

(3) TPL 在目标代码中的结构变化增加了依赖分析的难度. TPL 的复用往往伴随修改, 例如, 原本属于同一文件的函数可能被分散到不同文件中, 或一个文件可能包含来自多个 TPL 的函数, 这使得 TPL 成分划分更加困难. 若划分不准确, 可能会导致依赖关系分析的冗余或遗漏. 此外, 若多个同名头文件存在于相同路径层级, 仅依赖“#include”语句进行分析将难以准确判断被包含的文件, 从而引发依赖关系解析错误, 进一步加大分析难度.

为解决以上挑战, 本文提出了 CAnalyzer, 一种有效的针对 C/C++源代码的 SCA 技术. CAnalyzer 以 C/C++源代码作为输入, 输出源代码中复用的 TPL 成分和 TPL 之间的依赖关系.

(1) 为了减少公共函数对 SCA 检测结果的干扰, CAnalyzer 对融合多托管仓库构建的 TPL 特征库中的函数进行了原始 TPL 归属识别处理. 首先, CAnalyzer 通过函数路径等标识信息, 过滤掉与特定 TPL 无关的噪声函数; 其次, 采用聚类算法将相似 TPL 归为同一“家族”. 每个家族内部的成员共享部分公共函数, 同时也包含各自独有的标志函数. 基于“被复用库中标志函数占比较低”的假设, CAnalyzer 分析各 TPL 之间共享公共函数的复用关系, 从而有效区分 TPL 自身代码与外部代码, 并剔除不属于特定 TPL 的公共函数. 最终构建的 TPL 特征能够更加准确地反映其真实的函数归属情况, 避免了公共函数对后续分析结果造成干扰.

(2) 为了应对特征粒度选择问题, CAnalyzer 采用函数粒度特征相似性计算, 同时结合函数、文件和库粒度多级复用阈值准确识别复用的 TPL. 首先, 将 TPL 库拆解为细粒度的函数级特征与目标代码对比, 从而确保即使复用的函数分散在不同的 TPL 文件中, 也能够准确识别其复用情况. 随后, 在函数复用比例达到设定阈值后, 进一步分析文件层面的复用比例; 最后, 综合考虑整库层面的复用比例, 判断目标代码是否复用了该 TPL. 若函数、文件和库这 3 个层次的复用比例均达阈值, 则可确认该 TPL 为目标代码的有效成分. 通过递进式分析, CAnalyzer 实现了从函数到文件再到整库的多级阈值识别策略, 既能灵活应对代码结构的变化, 又能有效降低误报.

(3) 为了应对目标代码中 TPL 结构变化带来的依赖分析问题, CAnalyzer 首先明确划分 TPL 模块, 并通过分析文件级依赖关系推导 TPL 间的依赖. 基于 SCA 结果, 结合目标代码的目录结构和元数据信息, CAnalyzer 采用 4 项划分标准, 将目标代码划分为不同的 TPL 模块, 每个模块包含多个文件, 从而将 TPL 间的依赖关系抽象为文件集之间的依赖关系. 通过分析#include 和 extern 语句, 识别模块间的直接依赖关系. 然而, 直接依赖关系可能会遗漏, 例如头文件冲突或 TPL 模块划分不准确时, 可能导致依赖关系无法正确识别. 为弥补这些遗漏, CAnalyzer 通过类、枚举、结构体等标识符推断间接依赖关系.

为了评估 CAnalyzer, 我们构建了一个高质量的评估数据集, 分别从 GitHub 和 Gitee/OpenHarmony 收集了 200 个 C/C++语言项目的源代码, 涉及 515 个被复用的 C/C++ TPL. 实验结果表明, 在 TPL 检测任务中, CAnalyzer 的精确率和召回率分别达到了 90.6% 和 86.6%, 显著优于当前主流的 SCA 工具: CENTRIS[13]、TPLite[14]和 OSSFP[15]. 在依赖关系检测方面, 我们以 OpenHarmony 项目中的 1 784 对文件依赖关系作为标准集, CAnalyzer 实现了 94.8% 的精确率和 99.0% 的召回率, 充分满足自动化构建 TPL 依赖关系的需求. 此外, 我们还评估了特征库预处理和多级阈值策略对 TPL 检测性能的影响, 实验结果显示, 召回率保持在 86.6% 时, 经过多阶段预处理后的特征库将精确率提升 25.6%–74.0%. 在效率方面, CAnalyzer 在支持大规模特征库的同时, 实现了较高的检测效率 (969.5 s).

本文的主要贡献如下.

(1) 全面覆盖 C/C++ TPL 的特征库. 创建了一个 C/C++ TPL 索引库, 收集了来自 15 个托管仓库中的 C/C++ TPL 的源代码存储库地址等信息. 基于此索引库, 构建了一个大规模 C/C++ TPL 特征库, 包含 33 100 个 C/C++ TPL 中的 30 047 290 个函数的代码特征, 为软件成分分析提供了坚实的数据基础.

(2) 准确而可靠的 TPL 成分检测能力. CAnalyzer 通过特征库预处理和多级阈值识别 TPL, 实现了精准的软件

成分, 为开发者理解和分析软件组成提供了可靠的支持.

(3) 精准而完整的 TPL 依赖关系建模. CAnalyzer 通过分析源码文件中“#include”“extern”等指令以及标识符建立 TPL 间的依赖关系图. 该依赖关系图详细展示了软件项目中各个组件之间的依赖关系, 为安全漏洞传播和软件许可证合规性分析提供数据支持.

(4) OpenHarmony 社区认可的软件成分分析技术. CAnalyzer 已协助 OpenHarmony 社区中 689 个 C/C++代码仓库识别出 166 个外部 C/C++ TPL 成分, 其中 61 个获得开发者确认, 为首次标注. 结合 C/C++漏洞数据库, CAnalyzer 检测出 10 个同源安全漏洞, 其中 3 个已被社区安全响应工作组确认, 为有效漏洞. 因为本文提出的软件成分分析技术对 OpenHarmony 社区安全治理方面的贡献, 作者团队被 OpenHarmony 安全委员会授予“OpenHarmony 社区安全治理示范单位”. CAnalyzer 开源仓库地址: https://gitcode.com/openharmony-sig/compliance_composition_analysis.

2 背景与动机

2.1 公共函数干扰

本文所指的公共函数是指 TPL 间复用引入的克隆函数, 以及实现常见算法 (如排序算法) 等的相似函数. 这些函数模糊了 TPL 的边界, 影响现有 SCA 技术的有效性[13]. 图 1 展示了公共函数导致假阳性的一个典型场景: 目标代码 third_party_selinux 复用了 selinux 库, 而后者又包含 libsepol 库的代码. 由于特征库记录了多个复用 libsepol 的 TPL, 这些 TPL 共享部分相同的函数, 导致在成分分析过程中, 所有复用 libsepol 的 TPL 都与目标代码匹配, 错误地被识别为目标代码的一部分, 从而产生假阳性.

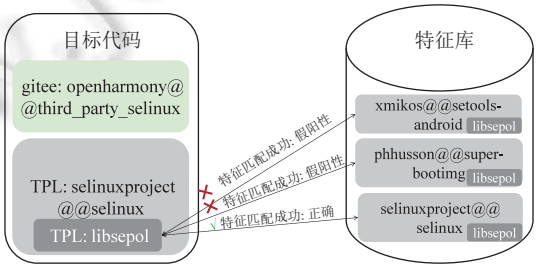


图 1 公共函数造成假阳性示例

为此, 已有研究引入了函数诞生时间[13,14], 公共函数被归属于函数诞生时间更早的库, 然而, 在融合多个托管仓库构建的特征库中, 函数诞生时间的准确性受到挑战.

(1) 版本管理方式的多样性使函数诞生时间难以获取. 尽管 Git 版本控制系统可提供精确的提交时间戳, 便于追踪函数的创建时间, 但调查结果显示, 约 38.6% 的 C/C++ TPL 未采用 Git 管理 (见表 1), 而是使用其他版本控制平台. 例如, libaiff[20]库托管于 SourceForge[21], 其版本控制信息的获取远不如 Git 便捷; C/C++ TPL termcap[22]通过 FTP 服务器 (ftp://ftp.gnu.org/gnu/termcap) 发布不同版本的源代码, 缺乏标准的时间戳信息支持, 进一步加剧了函数诞生时间的提取难度.

(2) 版本管理不完善导致获取不准确的函数诞生时间. 已有研究将软件版本的发布时间视为函数的诞生时间. 然而, 从多个托管仓库收集的 33 100 个采用 Git 版本管理的 C/C++ TPL 中 (见表 2), 20.0% 的 TPL 未标记任何 Release 或 Tag, 导致函数诞生时间难以准确确定. 对此, 已有研究者采用主分支 (master) 的最新提交时间作为近似代替, 但这一方法存在明显的局限性[15]. 如图 2 所示, SwiftShader 库[23]从未发布 Release 和 Tag. 当特征库同时包含了最新版本的 SwiftShader2 和其他 TPL A (复用早期版本的 SwiftShader1) 时, 函数诞生时间的误判会导致函数归属错误: 部分应属于 SwiftShade2 的函数被误认为 TPL A 的一部分, 导致目标代码在与 TPL A 的比较中产生假阳性; 同时, 这些函数被错误剔除, 可能导致目标代码无法匹配 SwiftShader2, 从而产生假阴性.

表 1 C/C++ TPL 索引库和特征库统计数据

托管仓库类型	托管仓库名称	C/C++ TPL索引数据库 (包含源代码存储库地址等信息)			特征数据库中C/C++ TPL数量
		Git管理存储库数量	非Git管理存储库数量	存储库总数量	
应用程序级托管仓库	Conan	903	119	1 022	569
	Vcpkg	1 253	308	1 561	726
	Clib	239	52	291	205
	Cppget	250	51	301	87
	Xrepo	290	39	329	146
	Buckaroo	274	11	285	140
	Hunter	251	111	362	80
操作系统级托管仓库	Ubuntu	11 531	3 546	15 077	2 511
	Debian	20 638	6 232	26 870	7 016
	Archlinux	4 982	8 163	13 145	1 123
	Fedora37	5 303	9 801	15 104	1 101
	openEuler	924	1 032	1 956	155
源代码级托管仓库	GitHub	—			16 756
	Gitee				851
	GitLab				1 634
总计		46 838 (61.4%)	29 465 (38.6%)	76 303	33 100

注: “—”表示该类源代码托管平台统一采用Git进行版本管理, 因此不再区分Git与非Git管理存储库, 也不统计相应数量

表 2 采用 Git 管理的 C/C++ TPL 统计数据

版本管理不完善的C/C++ TPL数量	版本管理完善的C/C++ TPL数量	函数总数量	公共函数数量
6 606	26 494	30 047 290	11 453 633

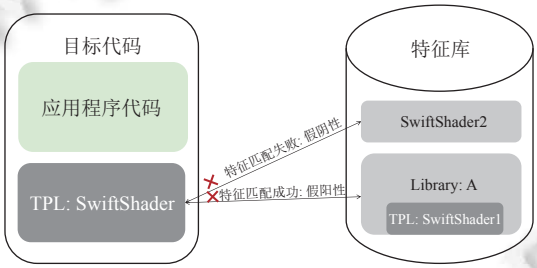


图 2 版本管理不善的情况示例

(3) 迁移托管仓库导致丢失 TPL 历史版本的函数诞生时间. 例如, 开源解压软件 7-Zip 最初仅在 SourceForge 托管源码[24], 后仅将最新版本迁移至 GitHub 的 ip7z@7zip[25]存储库. 与此同时, 多个非官方存储库 (如 p7zip-project@p7zip[26]) 已在迁移前出现, 与 ip7z@7zip 存在相同函数片段. 若以诞生时间判断函数归属, 会错误地将这些函数归为 p7zip-project@p7zip 的一部分, 进而影响到 SCA 结果准确性.

挑战 1. 利用函数诞生时间划分公共函数归属面临多重局限: (1) 非 Git 管理的 TPL 难以获取诞生时间; (2) TPL 版本管理不完善获取到不准确的诞生时间; (3) TPL 迁移托管库丢失诞生时间, 进而威胁到 SCA 结果的准确性.

2.2 依赖关系缺失

现有的 SCA 工具在揭示 TPL 之间复杂依赖关系方面存在不足, 流行的 SCA 工具如 CENTRIS 等, 通常未能提供目标代码中引用的 TPL 之间的依赖关系. 这给软件安全、合规性以及维护带来了诸多挑战. 以下两方面凸显了研究 TPL 依赖关系分析算法的紧迫性和重要性.

● 安全漏洞的影响范围分析. 在软件项目中, 外部 TPL 的引入虽为开发带来了便利, 但同时也可能不经意间引

入了潜在的安全漏洞, 威胁到软件的整体质量和稳定性. 通过跟踪 TPL 间的依赖关系, 可以揭示安全漏洞在软件项目中的传播途径, 有助于安全团队根据风险的严重程度优先处理漏洞, 确保最关键的资源得到及时有效保护.

● 许可证合规性审查的准确性分析. 由于 TPL 发布于多样化的许可证之下 (如 BSD、MIT、GPL 等), 其使用受到相应版权法及许可条款的严格约束, 这些条款要求开发人员必须严格遵守, 以确保项目的合法性与合规性. 然而, 由于无法追踪 TPL 间的依赖关系, 合规性审查往往仅限于项目自身及直接引用的 TPL 之间的许可证是否冲突, 而无法审查整个项目在组合使用多个 TPL 时的整体合规性.

例如, 某些许可证可能强制要求公开源代码, 但在缺乏依赖关系清晰图的情况下, 开发人员难以精确界定哪些代码片段需遵循此要求, 进而可能无意中违反许可协议, 引发法律争议. 历史上, 未能充分遵守 TPL 许可证的合规性要求, 诸如 Cisco 与 VMWare 因未遵守 Linux 内核的开源许可条款而陷入法律困境的案例屡见不鲜[5,27,28].

尽管某些包管理工具可以提供 TPL 之间的原生依赖关系, 但一旦 TPL 被引入目标代码中, 它便成为项目的一部分, 依赖关系应基于目标代码的实际情况分析, 而不只依赖原生关系. 目标代码中的 TPL 往往经过修改和整合, 原有的结构可能发生变化. 例如, 原本属于同一文件的函数可能被分散到多个文件中, 或者一个文件内可能包含来自多个 TPL 的函数, 这种结构变化使得在目标代码中准确划分 TPL 边界和分析依赖关系变得更加复杂.

挑战 2. 现有 SCA 工具缺乏对目标代码中 TPL 间依赖关系的分析, 限制了软件安全漏洞和许可证合规性审查的全面性. 此外, 由于代码结构的变化, 分析融入目标代码中的 TPL 依赖关系变得复杂, 难以准确划分 TPL 边界并追踪其依赖关系.

3 CAnalyzer 方法

为解决以上挑战, 我们提出了 CAnalyzer, 一种基于 S2S 的 SCA 技术, 用于分析 C/C++语言软件源代码中复用的 C/C++ TPL, 并深入分析这些 C/C++ TPL 之间的依赖关系, 为安全漏洞检测和许可证合规性审查提供准确的成分识别结果与依赖信息支撑. 图 3 展示了 CAnalyzer 的整体实现框架.

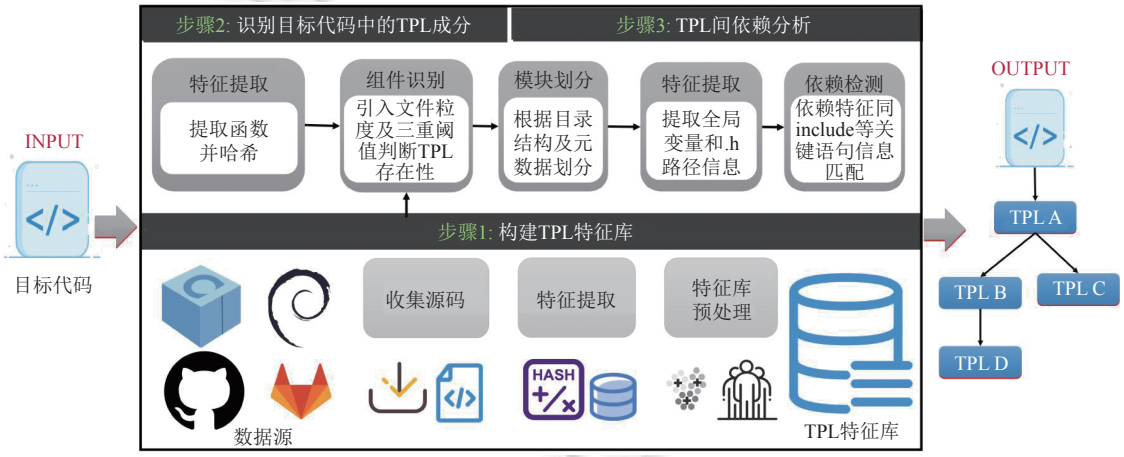


图 3 CAnalyzer 整体实现框架图

CAnalyzer 依托于一个大规模的高质量的 C/C++ TPL 特征库的支持, 覆盖了 15 个主流托管仓库中的 C/C++ TPL, 包含来自 33 100 个 C/C++ TPL 中的 30 047 290 个函数特征. 为解决普遍存在的公共函数和不准确的函数诞生时间对 SCA 检测结果造成的干扰 (引言中挑战 (1)), CAnalyzer 结合文件路径语义和相似 TPL“家族”来识别函数所属的原始 TPL, 进而减少公共函数造成误报 (步骤 1).

此外, 为了更精准地判断目标代码对 TPL 的复用 (引言中挑战 (2)), CAnalyzer 选取细粒度特征 (函数粒度) 执行相似性计算, 并结合多级复用阈值: 函数级→文件级→TPL 级, 来确定 TPL 是否被复用 (步骤 2).

最后, 为了全面而准确地分析这些被复用 TPL 之间的依赖关系 (引言中挑战 (3)), CAnalyzer 首先将目标代码

划分为 TPL 模块, 并且同时兼顾“#include”和“extern”等多种指令分析模块间的直接和间接的依赖关系, 为全面安全漏洞检测和许可证合规审查提供数据支持 (步骤 3).

3.1 构建 TPL 特征库

3.1.1 TPL 数据集构建

针对 C/C++ TPL 分散在不同托管仓库且存在重复的问题, CAnalyzer 多仓库联动的动态收集方法, 其核心思想是: 先通过仓库间信息整合避免重复下载, 再通过“滚雪球”式迭代逐步扩大收集范围, 从而在节省存储空间的同时覆盖尽可能多的 C/C++ TPL, 具体实施分为 3 个步骤.

步骤 1: 多仓库数据整合与去重. 根据已有研究[29]的发现, C/C++ TPL 通常托管于 3 类仓库 (见表 1). 因此, 本文从这 3 类仓库中选取了 15 个具有代表性的流行托管源来收集 C/C++ TPL. 对于每个 TPL, 我们统一提取基本元数据信息: TPL 名称、版本号、源代码存储库地址等. 由于同一个 TP 可能出现在不同的托管仓库中 (例如 OpenSSL 同时存在于 GitHub 与 Ubuntu 仓库), 为了避免重复收集, 本文首先采用自动化方式进行初步去重: 如果不同托管仓库中 TPL 的源代码仓库地址一致, 则判定为同一 TPL 并合并记录. 该过程无需人工判定, 能够高效地消除大部分冗余条目, 从而生成一个“索引库”, 相当于为每个 TPL 创建了一份“身份证目录”. 然而, 值得注意的是, 仅凭源代码仓库地址一致性并不能完全覆盖所有情况. 一些 TPL 可能托管于不同的源代码仓库地址, 但本质上仍然指向同一个 TPL, 对于这类情况, 在后续阶段 (参见第 3.1.3 节) 进一步通过函数级别的相似性分析来判定归属, 从而实现更精确的 TPL 去重.

步骤 2: 起始 TPL 集合选取. 由于 C/C++生态规模庞大且边界模糊, 直接收集所有可能的 TPL 在可行性和效率上都存在困难. 因此, 本文首先选取一批具有代表性的“种子库”作为切入点逐步扩展收集. 具体而言, 我们在 GitHub 上筛选出满足以下条件的 C/C++ TPL: (a) 星标数不少于 100 (受欢迎程度高); (b) 过去 1 年内有提交或版本更新 (可维护性好); (c) 在所有满足条件的项目中, 根据综合指标 (主要为星标数和贡献者数) 进行排序, 选取前 1 000 个 TPL 作为起始集合, 能够覆盖大多数在实际工程和科研场景中被频繁依赖的核心 TPL, 例如 libpq (数据库连接库)、OpenCV (图像处理库)、Boost (通用库集合) 以及 gRPC (远程过程调用框架) 等.

步骤 3: 动态迭代收集 TPL. 从种子库出发, 采用“解析依赖-搜索-获取”的循环机制逐步扩大收集范围 (后文图 4(a)). (1) 依赖解析: 使用 Ccscaner 工具[29]分析种子库的依赖关系, 找出它们需要调用的其他库 (如配置文件里声明的依赖项、代码中包含的头文件等). (2) 跨平台检索: 根据发现的依赖项名称, 同时在两个渠道查找具体代码: 自建的索引库 (快速匹配已知库); GitHub/GitLab/Gitee 仓库 (搜索相关存储库). (3) 去重下载: 对找到的依赖项仓库, 通过比对目录结构, 确保不重复下载相同的库. 新获得的库会加入收集列表, 成为下一轮解析的起点. 通过这种“滚雪球”式的收集方式, CAnalyzer 实现了存储效率与覆盖广度的均衡, 既收录了主流常用 C/C++ TPL, 也通过依赖挖掘发现了许多非热门但被实际项目引用的长尾库 (指的是使用频率较低、关注度不高、但在特定场景中仍被实际项目引用的小众第三方库, 如嵌入式领域专用算法库).

3.1.2 TPL 特征提取

TPL 的特征是其代码及多种属性的集合, 用于支持与目标代码的匹配. 我们利用 Ctags[30]工具解析 TPL, 提取函数、类、变量等代码元素的名称、文件位置和行号等信息. 开源软件更新时, 并非所有源代码都会重新开发. 不同版本之间存在公共部分. 传统的 SCA 工具会为每个版本存储一次函数记录 (如图 5(a) 所示). 这不仅浪费存储空间, 还导致多版本之间的公共部分同目标代码重复比较.为减少冗余比较并降低存储和计算复杂度, 我们采用了增量式存储策略[13]. 具体来说, 我们只存储跨版本重复函数的单一记录, 并附加版本和路径信息. 最终, TPL 特征库的状态如图 4(b) 所示, TPL 存储结构如图 5(b) 所示. 每个 TPL 主要包含以下特征.

- 源码摘要. 去除空格、换行符和注释后, 计算函数的 TLSH[31]哈希值, 记录版本及路径, 用于匹配目标代码.
- 版本信息. 记录所有版本的时间和名称, 用于评估存储库的活跃程度.
- 存储库地址. 包含下载地址, 支持使用 Git 命令克隆与更新.
- 存储库目录. 简化目录结构为字符串, 便于识别目录相似的存储库是否为同一 TPL.

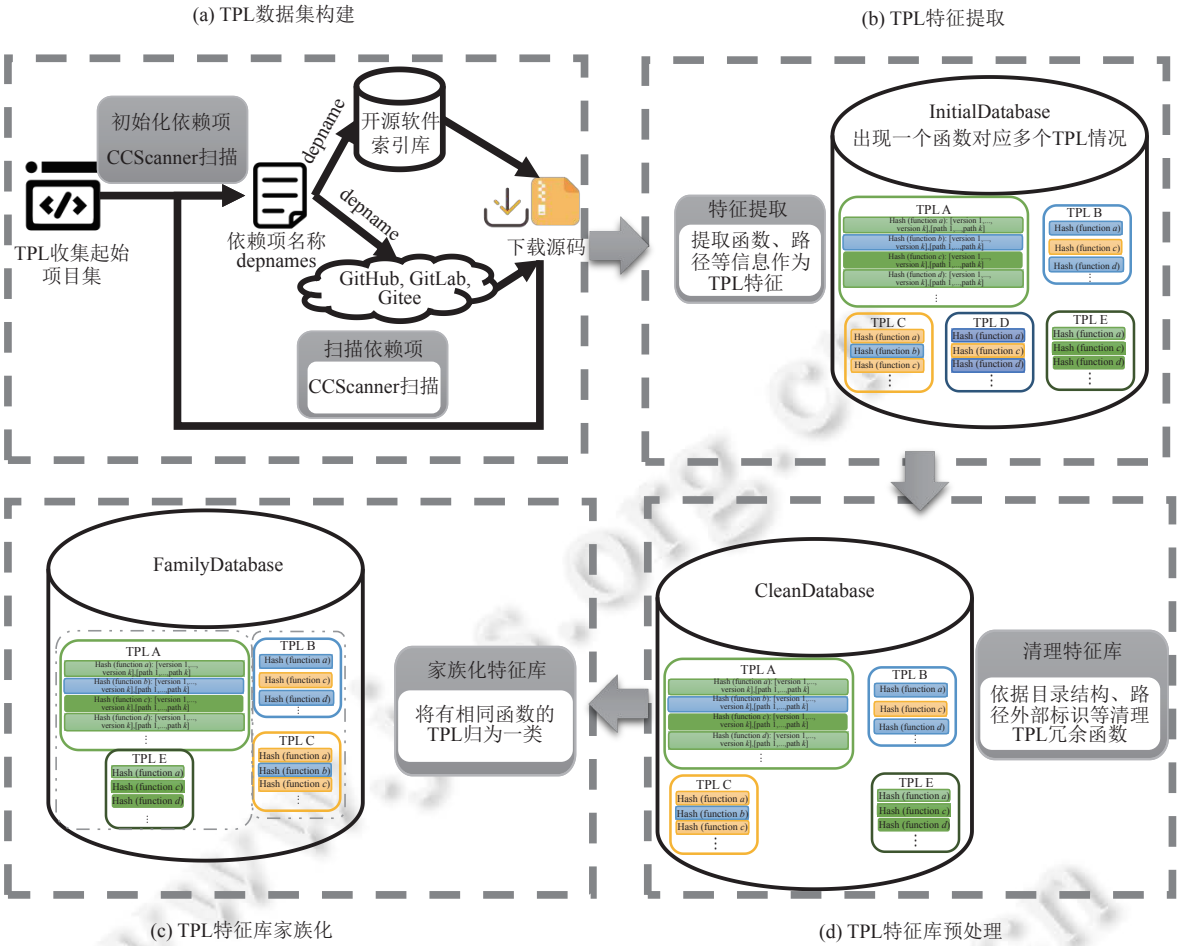


图 4 C/C++ TPL 索引库和特征库构建流程

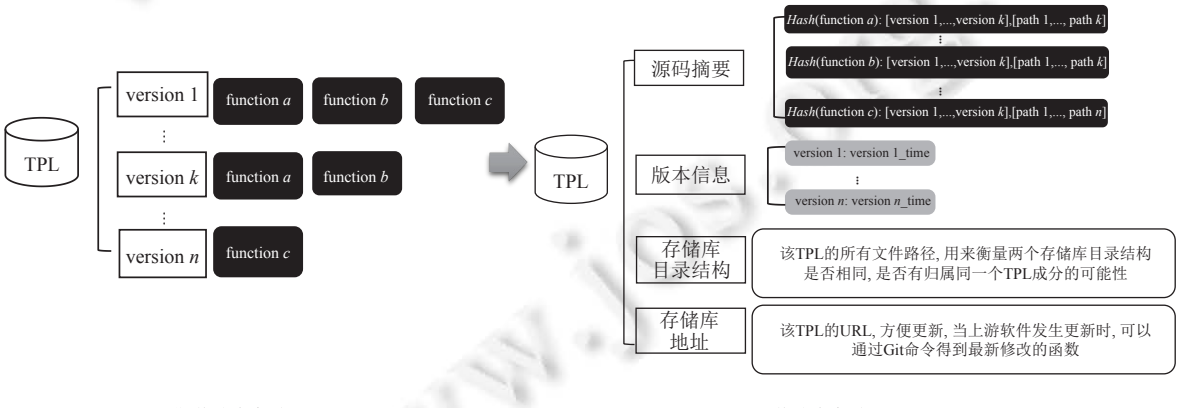


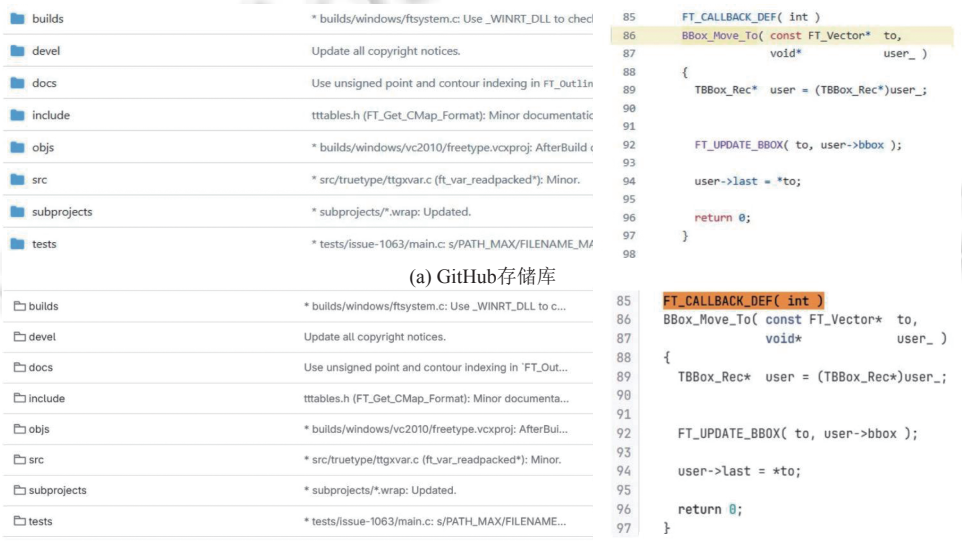
图 5 C/C++ TPL 特征存储结构

为了确保 TPL 特征的代表性和准确性, 我们排除了源码中的测试函数. 测试函数通常用于验证功能、性能和稳定性, 不直接参与核心功能实现, 且一般位于特定的测试目录 (如“./test”). 因此, 在构建源码摘要时, 我们移除了这些测试函数, 以避免引入不必要的干扰.

3.1.3 TPL 特征库预处理

在初步构建的特征库中, 由于多个 C/C++ TPL 可能共享相同函数, 公共函数的广泛分布模糊了 TPL 边界, 影响 SCA 的准确性. 为此, 我们进行了多阶段处理: 首先, 通过目录结构等线索初步过滤, 去除不属于特定 TPL 的函数 (图 4(c)); 随后, 将相似 TPL 归类, 并清理跨 TPL 共享的公共函数 (图 4(d)). 经过这些步骤, 每个 TPL 的特征更加精准地反映其代码特性.

阶段 1: 基于明确线索的初步过滤. 我们首先为特征库的所有函数建立一张索引, 包含: (1) 出现频率: 包含相同函数哈希的库的数量; (2) 路径信息: 库名称及函数在该库出现的路径. 遍历函数索引时, 对于出现频率为 1 的函数, 因其能够明确标识特定的 TPL, 充分反映 TPL 特性, 因此保留在源码摘要中; 而对于出现频率超过 1 的函数, 我们认为它们是公共函数, 并进行以下两步过滤操作. (a) 过滤外部库函数. 本步骤移除直接源自外部库的函数, 这些函数可通过路径标识 (如“arangodb/3rdParty/abseil-cpp”) 明确区分. 由于其复用关系已在路径中体现, 因此不作为 TPL 特征. 我们通过匹配路径中的外部标识 (如 3rdParty、external、deps 等) 来识别并移除这些外部库的函数. (b) 过滤同一 TPL 不同存储库的重复函数. 在开源生态中, 同一 TPL 可能会有多个代码存储库, 这些存储库分布在 GitHub、GitLab 等平台, 由不同的组织或个人维护, 以满足跨平台、编程语言或特定应用场景的需求. 尽管它们的物理位置不同, 但目录结构和函数实现往往高度相似. 如图 6 所示, freetype[32]的多个存储库包含相同的函数 (如 FT_CALLBACK_DEF). 为了消除这类公共函数的干扰, 我们对存储库的目录结构进行聚类分析 (K-means), 对于目录结构相似且包含相同函数的库, 我们筛选出活跃的存储库作为代表, 并保留其中的函数, 剔除其他冗余库中的重复函数.



(a) GitHub存储库

(b) GitLab存储库

图 6 相同 C/C++ TPL 在不同存储库中的重复函数示例

阶段 2: 相似 TPL 归类与公共函数剔除. 除显式地引入外部代码外, 还存在隐式代码克隆现象, 即开发者直接复制外部库代码而未标明来源[13]. 此外, 一些基础算法 (如排序、查找) 常被开发者在不同项目中独立实现, 加剧了函数跨库复用的现象[15]. 这些公共函数因缺乏明确来源标识, 增加了区分难度. 为此, 我们通过分析共享这些函数的 TPL 复用关系, 追踪代码来源, 揭示公共函数的真实归属.

首先, 通过公式 (1) 构建 TPL 相似度矩阵. 公式 (1) 中的 A 和 B 分别代表需要计算的两个 TPL 的函数哈希值集合. 然后, 利用层次聚类 (hierarchical clustering) 算法将相似的 TPL 归为同一“家族”, 每个家族内部共享一定数量的公共函数, 并包含各 TPL 独有的标志函数.

$$J(A,B)=\frac{|A\cap B|}{|A\cup B|} \tag{1}$$

在聚类后的家族中, 如果 TPL X 复用了 TPL S, 则 TPL X 既包含一些公共函数, 也保留了独有的标志函数. 因此, TPL X 的标志函数比例较高. 而 TPL S 由于被多个 TPL 复用, 包含了更多的公共函数, 标志函数比例较低. 由此可见, 通常被复用的 TPL 标志函数比例较低. 基于这一假设, 我们将公共函数归属于标志函数比例较低的 TPL S. 为了量化 TPL 中标志函数的比例, 我们引入两个指标: *FILF* (function inverse library frequency) 和 *DI* (distinctness index). *FILF* 是衡量一个函数在多个 TPL 中普遍性的统计量. 它基于逆文档频率 (IDF) 的概念, 其中, N 代表该类中 TPL 总数, $df(f)$ 代表函数 f 出现的频率. *FILF* 值越高, 表示该函数在少数几个 TPL 中出现, 因此具有更高的独特性, 可视为标志函数.

$$FILF(f)=\log\left(\frac{N}{1+df(f)}\right) \tag{2}$$

DI 是衡量一个 TPL 在特定家族中标志函数比例的量化指标. 它通过对 TPL 中所有函数的 *FILF* 值求和, 再除以 TPL 中函数的总数计算得出. *DI* 值越高, 表示该 TPL 对公共函数的依赖越小, 标志函数比例越高.

$$DI(\text{TPL})=\frac{\sum_{i=1}^{Totals}FILF(f_i)}{Totals} \tag{3}$$

在同一“家族”内, 根据每个 TPL 的 *DI* 值升序排序, 较低的 *DI* 值通常对应更少的标志函数和更高的复用性. 因此, 基于被复用库中标志函数比例较低的原则, 我们将公共函数优先归属于 *DI* 值较低的 TPL. 从排序列表中的第 1 个 TPL 开始, 标记其所有函数为该 TPL 的标志函数. 如果下一个 TPL 中存在已被其他 TPL 标记的公共函数, 则将这些函数剔除. 该过程持续到遍历完所有 TPL 为止.

3.2 目标代码成分识别

为了更好地适应软件库粒度复用检测场景, 我们利用 Ctags[30]和 TLSH[31]算法将目标代码表示为特征集合, 并设计了一套多层次、多阈值的评估体系, 用于将目标代码特征与特征库中的 TPL 进行相似性匹配. 仅当特定 TPL 同时满足三重阈值时, 才能确认其是目标代码的有效成分.

(1) 函数级评估. 设定函数级阈值 α, 通过计算目标代码中与特定 TPL 匹配的函数数量 (*MFuncs*) 占该 TPL 每个版本平均函数数量 (*AFuncs*) 的比例, 初步筛选目标代码潜在复用的 TPL.

$$\frac{MFuncs}{AFuncs}\geqslant\alpha \tag{4}$$

(2) 文件级评估. 设定文件级阈值 β, 用于判断目标代码与特定 TPL 文件中匹配函数数量 (*MFuncsFile*) 占该文件总函数数量 (*TFuncsFile*) 的比例是否达到预设阈值. 若该比例超过 β, 则认为目标代码实际复用了该文件.

$$\frac{MFuncsFile}{TFuncsFile}\geqslant\beta \tag{5}$$

(3) 库级评估. 设定库级阈值 θ, 根据目标代码实际复用的 TPL 文件数量 (*RFiles*) 占该 TPL 总文件数量 (*TFiles*) 的比例, 综合评估目标代码对该 TPL 的整体复用程度. 若该比例超过 θ, 则最终确认目标代码复用了该 TPL.

$$\frac{RFiles}{TFiles}\geqslant\theta \tag{6}$$

3.3 TPL 依赖检测

3.3.1 TPL 成分模块划分

分析目标代码中的 TPL 依赖关系时, 我们关注的是 TPL 在引入后作为目标代码一部分的实际表现, 而非其原生环境中的依赖关系. 为重新分析 TPL 之间的关联, 我们需要基于 SCA 结果将目标代码分解为不同的 TPL 模块. 如图 7(a) 所示, 成分识别已明确目标代码中引入的 TPL 及其与特定文件的复用关系. 接下来, 我们分析与 TPL 相关的复用文件, 判定哪些文件属于特定 TPL, 并将其归入相应的 TPL 模块. 为此, 提出了 4 项标准, 用于判定文件是否应归属于特定 TPL 模块.

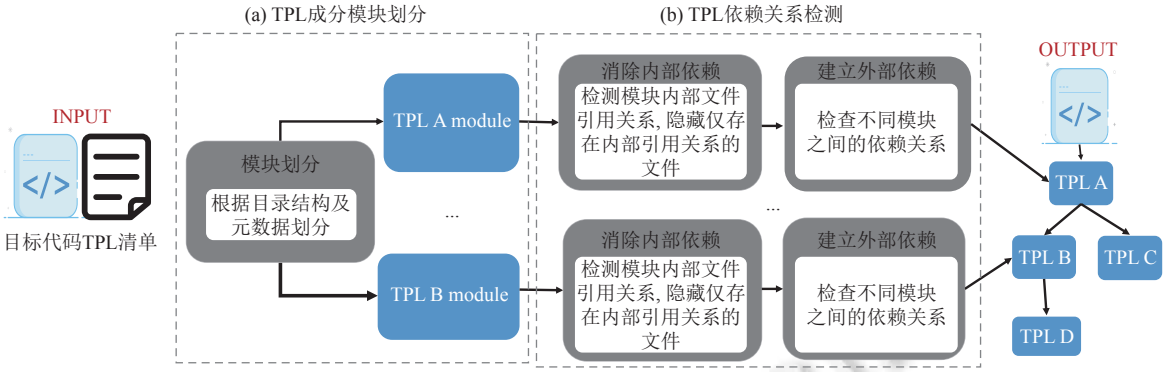


图 7 TPL 依赖关系检测框架

● 标准 1: 基于目录命名的 TPL 组成部分判定. 若复用文件的路径中包含 TPL 名称, 则可以直接判定该文件为该 TPL 的一部分. 例如, spirv-tools 复用文件路径“skia/third_party/externals/spirv-tools/source/opt/optimizer.cpp”中包含“spirv-tools”, 因此该文件应归属于 spirv-tools 库.

● 标准 2: 基于 TPL 头文件位置的 TPL 组成部分判定. 若复用文件所在目录包含 TPL 的头文件 (.h 文件), 则该目录下的文件可归属于该 TPL. 例如, 若目录中包含 libfoo.h 文件, 且该目录下文件与 libfoo 发生代码复用, 则该目录下的复用文件应归属于 libfoo 库.

● 标准 3: 基于 TPL 元数据文件位置的 TPL 组成部分判定. 若复用文件所在目录包含 TPL 的元数据文件 (如 README、LICENSE、COPYING), 则该文件应归属于相应 TPL. 为提高判定的准确性, 我们将检查元数据文件的内容, 验证其中是否包含 TPL 名称. 如果验证成功, 确认元数据文件属于该 TPL, 并将此目录下的复用文件归为该 TPL 的组成部分.

● 标准 4: 基于函数复用数量的 TPL 组成部分判定. 在缺乏明确线索的情况下, 我们依赖函数复用频率来判定复用文件是否属于 TPL 的实际组成部分. 具体方法是统计每个复用文件所在目录的复用频率 (复用频率=该目录中与 TPL 相关的复用函数总数/该目录中函数总数), 然后选择复用频率最高的目录, 将此目录下的复用文件归为该 TPL 的组成部分.

对于每个 TPL, 若其某些复用文件已通过前 3 个标准之一归类, 则剩余未分类的文件不再视为该 TPL 的组成部分. 由于这些文件与 TPL 的关联度较低, 因此不予归属. 仅当所有复用文件无法通过前 3 个标准判定时, 才使用第 4 个标准来确定哪些文件属于该 TPL 的组成部分. 此外, 已归类的文件不会被重新划分.

3.3.2 TPL 依赖关系检测

在 CAnalyzer 中, 目标软件的每个源文件首先通过 TPL 检测阶段被映射至其所属的 TPL. 由此, 每个 TPL 可抽象为一个包含多个源文件的“文件集”. 在依赖关系检测阶段, CAnalyzer 对文件级依赖 (包括头文件引用、符号引用与链接依赖) 进行静态分析, 并将这些依赖关系在文件集层面进行聚合, 从而构建出 TPL 间的依赖关系图. 为建立这一映射关系, 我们首先形式化定义软件项目中任意文件对之间的依赖判定规则.

(1) 文件对依赖关系分析. 在同一软件项目中, 文件 *A* 的执行依赖于文件 *B* 的存在或正确性时, 文件 *A* 对文件 *B* 就存在依赖关系. 此类依赖关系可以细分为以下几种.

(a) 直接依赖关系. 文件 *A* 通过显式引用文件 *B* 的资源而形成依赖, 通常采用以下两种方式.

● #include 指令声明. 文件 *A* 通过“#include”指令包含文件 *B* (如.h 或.hpp), 从而能够访问文件 *B* 中声明的变量、函数或类型定义. 如图 8 所示, 在 C++项目中, Main.cpp 包含了 Utility.h, 形成直接依赖关系. 我们通过解析源码中的“#include”语句, 并匹配头文件路径, 来确认文件 Main.cpp 是否依赖文件 Utility.h. 对于存在多个同名头文件的情况, 遵循 GNU GCC[33]的路径解析规则, 选择路径最近或明确指定的头文件. 如果路径冲突无法解决, 则将此类冲突留待后续间接依赖分析.

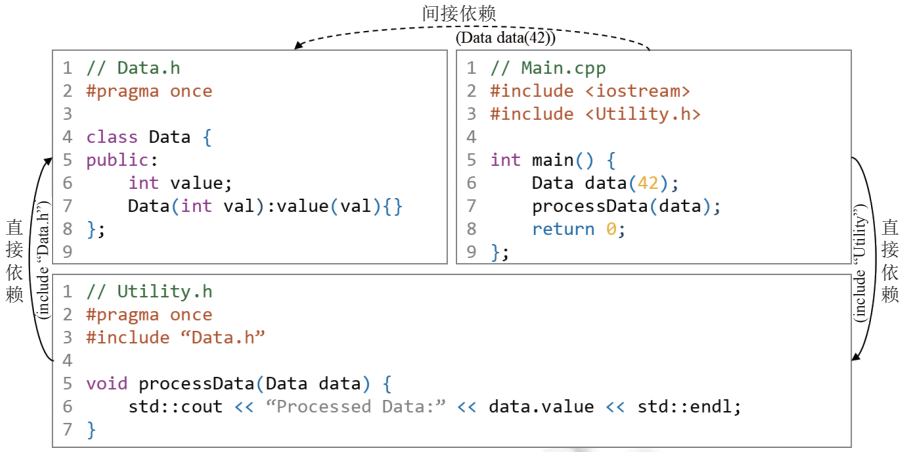


图 8 文件依赖关系示例

● extern 关键字声明. 文件 *A* 通过"extern"声明引用文件 *B* 中的外部变量或函数, 从而允许文件 *A* 使用文件 *B* 中的全局变量或函数. 我们使用 Ctags 工具生成代码索引, 提取所有"extern"相关的语句, 记录变量和函数的名称及其定义和声明位置. 通过解析这些信息, 可以确认文件 *A* 是否引用了文件 *B* 中的外部资源, 从而建立它们之间的直接依赖关系.

(b) 间接依赖关系. 间接依赖指的是文件 *A* 未直接通过"#include"或"extern"引用文件 *B*, 而是通过中间文件形成依赖链. 如图 8, Main.cpp 直接依赖 Utility.h 文件, 而 Utility.h 又依赖 Data.h 文件, 因此 Main.cpp 与 Data.h 之间形成了间接依赖.

在分析间接依赖时, 我们关注文件 *A* 和文件 *B* 通过类、结构体、枚举等标识符的引用关系, 因为这些标识符的定义和使用可以分布在不同文件中, 揭示文件间的关联性. 如果文件 *A* 使用了文件 *B* 中定义的标识符 (如类、结构体或枚举), 并且它们的命名空间和作用域一致, 就可以推断文件 *A* 间接依赖于文件 *B*. 宏标识符和命名空间标识符不在分析范围内, 因为宏仅在当前文件有效, 而命名空间可能在不同文件中重复或非连续定义, 不能准确表示文件间的依赖关系.

CAnalyzer 的目标是在基于大规模特征库的场景下, 实现对 C/C++第三方库的高效自动化识别与依赖关系构建, 为保证高可扩展性与执行稳定性, CAnalyzer 未采用程序级语义分析 (比如, 完整控制流和函数调用链分析), 以避免此类分析带来的高时间与空间开销 (比如跨单元的指针别名分析与上下文敏感调用图构建). 我们认为, 引入语义特征有望进一步提升依赖分析的精度. 未来工作中, 将探索在当前框架上实现轻量级语义增强, 在保持系统检测效率与可部署性的同时, 进一步提高依赖分析的准确性与上下文感知能力.

(2) TPL 依赖检测. 检测过程如图 7(b) 所示, 通过文件依赖关系推断 TPL (即 TPL 成分模块) 之间的依赖关系.

(a) 消除内部依赖. 首先排除同一模块内文件的依赖关系. 若两个文件属于同一 TPL 模块, 则忽略其相互依赖, 避免内部依赖干扰分析结果.

(b) 建立外部依赖, 针对每个 TPL 模块, 分析其与其他模块间的文件依赖关系. 若模块 *A* 与模块 *B* 中的文件存在直接或间接依赖, 则判定两模块间存在依赖关系. 为提高效率, 一旦确立模块间的依赖关系, 无需重复分析同一模块对的其他文件依赖.

基于上述流程, 最终, CAnalyzer 输出目标 C/C++项目中复用的第三方库列表以及第三方库之间的依赖关系图.

4 评 估

本文提出的 CAnalyzer 是一种针对 C/C++源代码的软件成分分析工具, 检测目标代码中复用的 C/C++ TPL, 并分析 TPL 之间的依赖关系. 为了验证 CAnalyzer 的有效性, 我们设计了 5 个研究问题 (research question, RQ).

● RQ1: CAnalyzer 的参数如何影响 TPL 检测能力? 通过统计 100 个项目的参数 α (公式 (4)) 、β (公式 (5)) 和 θ (公式 (6)) 的数据, 设定合理的参数范围, 并设置不同的参数组合, 对其余 100 个项目进行分析.

● RQ2: CAnalyzer 的 TPL 检测能力与相关工具相比表现如何? 我们使用 100 个项目, 分别通过 CAnalyzer、CENTRIS[13]、TPLite[14]和 OSSFP[15]进行检测, 并对比检测结果与标准集数据, 分析精确率和召回率.

● RQ3: CAnalyzer 的特征库预处理和 TPL 识别算法的改进对 SCA 能力有何影响? 我们设置了 3 种特征库和两种成分识别方法的 6 种组合, 并在 100 个项目上进行了检测, 分析这些改进对检测精确率和召回率的影响.

● RQ4: CAnalyzer 是否能准确检测文件集之间的依赖关系? 通过解析 OpenHarmony 4.0 中 14 783 个 BUILD.gn 文件, 获得 1 784 对文件集为标准数据, 评估 CAnalyzer 在直接和间接依赖检测方面的准确性.

● RQ5: CAnalyzer 的检测效率与相关工具相比表现如何? 我们使用 100 个项目, 分别通过 CAnalyzer、CENTRIS[13]、TPLite[14]和 OSSFP[15]进行执行效率分析, 包括: 特征提取和 TPL 检测的执行时间.

4.1 标准集构建

我们从 GitHub 随机筛选了包含 TPL 复用结构 (如 arangodb/3rdParty/abseil-cpp) 的项目. 同时, 为了验证工具在实际场景中的有效性, 我们特别关注了 OpenHarmony 4.0 的真实项目, 筛选出具有明确构建配置 (如 BUILD.gn) 和依赖声明 (如 README.OpenSource) 的项目, 并通过分析其目录结构和配置文件构建标准集. 在库粒度复用的场景中, 开发者通常采用特定的组织方式来存放复用的 TPL 代码, 如“arangodb/3rdParty/abseil-cpp”. 我们将这种映射关系 (如 arangodb 与 abseil-cpp) 视为标准集数据, 表示 arangodb 复用了组件 abseil-cpp.

(1) 排除非源代码目录. 若子目录仅包含头文件或其他语言文件, 则认定该映射关系无效并排除.

(2) 版权文件存在. 如果相应目录下存在版权信息 (如 LICENSE、COPYING 文件) , 通常包括供应商名称、作者名称和库名称, 则认为该映射关系有效, 并将其视为标准集数据.

(3) 自述文件存在. 如果相应目录下存在 TPL 的 README 文件, 同样认定映射关系有效. OpenHarmony 4.0 的真实项目使用 README.OpenSource 文件来标识软件复用, 记录了复用组件名称, 这为我们提供了一组映射关系作为标准集数据. 根据上述过程和规则, 我们审查了 1 000 多个 C/C++项目, 筛选出 200 个可用于构建标准集的软件项目, 并识别出 515 组复用关系 (见表 3).

表 3 实验标准集

项目来源	项目总数	BUILD.gn数量	构建模块数量	依赖关系数量	复用TPL数量
Gitee/OpenHarmony	100	14 873	12 277	1 784	515
GitHub	100	—	—	—	

注: “—”表示不适用. GitHub实验项目仅用于TPL复用检测评估, 不涉及BUILD.gn文件、构建模块及依赖关系统计

需要指出的是, 我们在标准集构建中优先采纳了“显式声明来源”的外部依赖, 如目录结构、版权文件或自述文件中直接标识出的复用关系, 能够确保标准集的可靠性和可验证性, 避免因人工推断或不确定性信息而引入噪声. 然而, 这一策略可能会遗漏部分未显式声明的 TPL, 例如, 某些开发者直接拷贝了第三方库的部分代码片段但未在目录或说明文件中保留来源标识. 我们在“有效性威胁”中明确讨论了该问题.

4.2 评估指标

评估中, 我们采用以下指标全面衡量 CAnalyzer 工具的性能.

(1) 真阳性 (*TP*): CAnalyzer 正确检测出的 TPL 成分, 这些成分在标准集中明确标记; 或工具正确识别出 source 模块依赖于 dep 模块的关系, 与标准集一致.

(2) 假阳性 (*FP*): CAnalyzer 检测到的 TPL 成分, 但这些成分在标准集中未被标记为 TPL; 或工具错误地表示 dep 模块依赖于 source 模块的关系, 而标准集中并不存在该依赖.

(3) 假阴性 (*FN*): 标准集中明确标记的 TPL 成分, 但 CAnalyzer 未能检测到; 或工具未能识别标准集中记录的模块间依赖关系.

(4) 精确率 (*Precision*): $Precision = \dfrac{TP}{TP + FP}$, 该指标衡量 CAnalyzer 识别 TPL 成分或检测依赖关系的准确性,

以评估其在实际应用中的可靠性和效果.

(5) 召回率 (*Recall*): $Recall = \frac{TP}{TP+FN}$, 该指标评估 CAnalyzer 全面检测 TPL 成分及其依赖关系的能力.

4.3 参数设置分析实验

为了探究 RQ1, 我们通过调整 α (公式 (4))、β (公式 (5)) 和 θ (公式 (6)) 这 3 个关键参数的不同组合, 评估其对检测结果的影响. α、β 和 θ 分别表示从函数、文件及库维度判断软件复用的阈值 (见第 3.2 节). CAnalyzer 将 α 设置为 0.1, 假设当两个项目之间的函数重合度超过 10% 时, 可能发生软件复用. α 采用固定值主要基于以下考虑: α 的作用在于过滤掉零散的偶然函数匹配, 10% 已被既有研究证明是合理的经验阈值[13,14]. CAnalyzer 研究重点在于探索文件级 β 与库级 θ 阈值组合对检测结果的影响而非重复已有研究对 α 的参数敏感性进行验证. 因此, 为避免引入额外变量, CAnalyzer 保持 α 不变专注于 β 与 θ 的参数设置实验.

为了全面评估参数的影响, 我们首先从标准集中的 200 个项目中随机抽取 100 个项目进行参数设置分析实验 (涵盖 314 个组件), 其余 100 个项目则用于有效性对比实验. 这样的划分的目的: (1) 避免数据泄漏. 通过将数据集一分为二, 确保参数调优过程中使用的数据不会与有效性评估的数据重叠, 从而保证实验结果的独立性与可靠性; (2) 提高泛化性. 在一个子集上探索合理的阈值区间, 在另一独立子集上验证工具的性能, 可以更好地评估所选参数在不同项目上的适应性. 在随机划分这 200 个项目时, 我们采用分层抽样策略, 确保两个来源在参数实验集和效果验证集中的分布比例保持一致, 避免因来源差异导致的偏向性. 最终, 参数实验结果显示: 这些项目平均复用 TPL 79% 的文件, 且这些文件中有 80% 的函数被复用. 基于该统计分布, 我们将 β 与 θ 的取值范围设定为 [0.5, 0.6, 0.7, 0.8, 0.9, 1]. 如图 9 所示, 不同的 β 和 θ 组合对精确率 (*Precision*) 和召回率 (*Recall*) 有显著影响. 随着 β 和 θ 值的增加, 精确率通常提高 (因为更严格的阈值减少了假阳性), 但召回率下降 (因为一些真实的 TPLs 未达到阈值). 这表明 β 和 θ 在精确率与召回率之间存在反向作用.

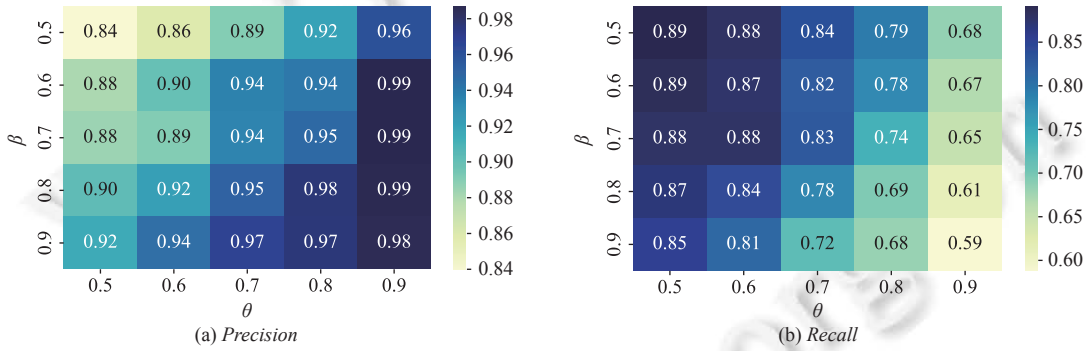


图 9 不同 β 和 θ 组合精确率和召回率分布

● 实验结论 (RQ1). CAnalyzer 的主要参数 (α、β 和 θ) 对其检测 TPLs 的能力具有显著影响. β 和 θ 在精确率与召回率之间存在反向关系. 为了平衡精确率和召回率, 我们使用调和平均值 $F1$ 分数[34]作为综合评价指标. 计算结果表明, 当 β=0.7 且 θ=0.6 时, $F1$ 分数达到最高值 0.88, 表明该参数组合在精确率和召回率之间取得最佳平衡.

4.4 有效性对比实验

本节实验旨在回答 RQ2, 我们将 CAnalyzer 与 CENTRIS[13]、TPLite[14]、OSSFP[15]在标准集上进行比较. 剔除第 4.3 节中用于统计阈值的项目后, 我们使用这些工具对剩余的 100 个项目进行检测, 并分析实验结果, 探讨假阳性和假阴性出现的原因.

● 与 CENTRIS 比较. 鉴于 CENTRIS 已发布特征库和源代码, 我们直接在其特征库的基础上对 100 个项目进行了检测. CENTRIS 通过函数诞生时间来判定公共函数归属, 认为诞生时间较早的库更可能是公共函数的“发源地”. 因此, 特征库构建时, 晚诞生库中的公共函数被视为非关键特征并剔除. 当目标代码与 TPL 相同的函数占比达到 10% 时, 认为该 TPL 是目标代码的有效成分.

如表 4 所示, CENTRIS 从 100 个项目中识别出了 184 个组件, 精确率为 51.63%, 召回率为 47.26%. CENTRIS 仍依赖于函数诞生时间来处理公共函数的归属问题, 正如第 2.1 节所述, 函数诞生时间的局限性使得公共函数的归属判断不够准确, 从而引发假阳性. CAnalyzer 引入了文件粒度分析和更严格的阈值标准来判断 TPL 的存在性, 从而在减少假阳性方面表现优异, 精确率达到了 90.63%. 此外, 由于 CENTRIS 的特征库仅包含 10 294 个 TPL, 导致部分真实组件被遗漏, 从而产生了大量假阴性 (*FN*), 降低了召回率. 而 CAnalyzer 使用了更为全面的特征库, 召回率达到了 86.57%, 高于 CENTRIS.

表 4 CAnalyzer 与相关工具有效性对比实验结果

工具	*TP*	*FP*	*FN*	*Precision* (%)	*Recall* (%)
CENTRIS[13]	95	89	106	51.63	47.26
TPLite[14]	172	102	29	62.77	85.57
OSSFP[15]	126	79	75	61.46	62.69
CAnalyzer	174	18	27	90.63	86.57

● 与 TPLite 比较. TPLite 公开了源代码, 但未发布其特征库. 因此, 我们重新执行了 TPLite 的特征库构建过程, 并使用与 CAnalyzer 相同的 TPL 数据集, 最终构建了一个包含 32 706 个 TPL 的特征库. TPLite[14]在 CENTRIS 基于函数诞生时间推导 TPL 依赖关系的基础上, 将特征库构建成有向无环图, 并结合 PageRank[35]算法和节点入度的中心性过滤器来矫正函数诞生时间推导的不准确依赖关系. 与 CENTRIS 相同, TPLite 也使用 10% 的阈值判断 TPL 是否为目标代码的有效成分.

如表 4 所示, CAnalyzer 在精确率上明显优于 TPLite. 其大多数假阳性 (*FP*) 案例, 源于报告的组件复用了真实组件的部分代码. 这表明, 尽管 TPLite 采用了中心性过滤器来弥补函数诞生时间的局限性, 但它仍无法准确归属公共函数, 从而导致如图 1 所示的假阳性问题. TPLite 重点关注 TPL 在全局图中的中心性, 通过评估 TPL 在图中的重要性来决定保留哪些依赖关系. 如果某个 TPL 在全局图中的连接度不足, 相关依赖可能会被误删, 从而导致函数被错误地归类为其他 TPL. 例如, 某个仅用于特定领域的核心 TPL, 仅由少数其他 TPL 复用, 但其在全局图中的重要性较低, 指向该 TPL 的依赖关系可能会被删除, 进而导致这些函数被错误地归类为其他 TPL.

相比之下, CAnalyzer 采用局部分析方法, 避免全局噪声干扰. 它将特征库划分为多个代码相似性较高的“家族”, 并在每个家族内部分析 TPL 之间的依赖关系, 确保每个 TPL 自身的函数得到保留, 并去除非 TPL 的函数. 因此, 在计算 TPL 和目标代码的重叠比例时, 分母能够准确反映每个 TPL 的实际函数数量, 避免了因过高的预设阈值或匹配比例过小而导致的假阴性 (*FN*) 问题. 实验结果进一步验证, 尽管 CAnalyzer 使用了严格的多重阈值, 但仍能保留召回率.

● 与 OSSFP 比较. 由于该商业工具未发布其特征库和源代码, 我们将 100 个项目上传到 OSSFP 网站在线检测, 并使用标准集和 OSSFP 生成的结果, 手动检查其准确性. OSSFP 将 TPL 的函数划分为克隆函数、支持函数、通用函数和核心函数, 通过设定阈值和函数诞生时间来去除前 3 类函数, 最终仅基于核心函数生成 TPL 特征, 以解决 TPL 之间的内部克隆问题. OSSFP 认为只要目标代码与 TPL 中的任一函数相同, 该 TPL 是目标代码的有效成分. 根据 OSSFP 生成的结果, 我们确认了 126 个正确案例, 精确率和召回率分别为 61.46% 和 62.69%, 如表 4 所示. OSSFP 结果中出现了大量如 git、lib 以及类似${library}的特殊变量和单个字母 (如 m) 形式的组件, 这些组件可能是基于目标代码的目录结构进行分析的结果, 因此, 误报 (FP) 案例显著增加. 此外, 由于 OSSFP 仅凭单个函数相同即认为目标代码复用了 TPL, 这不满足库粒度复用检测的需求, 进而导致了 FP 的增加.

相较于 CAnalyzer, OSSFP 的召回率较低, 可能是由于特征库的范围不完整. OSSFP 基于 GitHub 的 23 427 个 C/C++存储库构建了 TPL 特征库, 但其规模仍小于我们从 15 个平台收集的 33 100 个 TPL 特征库. 此外, 由于 OSSFP 未公开更多 TPL 复用细节, 其结果的准确性只能通过 TPL 名称与标准集中的组件名称进行对比, 检查过程可能引入误差, 从而影响了精确率和召回率的判断.

● 假阳性 (*FP*) 分析. CAnalyzer 的假阳性主要源于公共函数归属的不准确, 具体表现如下.

(1) 目标代码复用公共函数导致误报和漏报. 目标代码仅复用了家族内的公共函数, 而未包含任何标志函数. 在特征库预处理中, 这些公共函数被划归到家族内 *DI* 值较低的 TPL, 目标代码更容易匹配到该 TPL, 导致假阳性. 同时, 这些函数从真实 TPL 中被剔除, 无法正确识别真实复用组件, 导致假阴性. 例如, 在 OpenCC[36]案例中, 它复用了 pybind11[37]的部分代码, 而这些代码由于其广泛性被算法视为家族内 *DI* 值较低的 hera[38]的标志函数, 并从 pybind11 中剔除. 最终, 目标代码匹配过程中产生了假阳性 (匹配 hera) 和假阴性 (未匹配 pybind11).

(2) 混合语言软件干扰. 收集 TPL 时, 由于未能区分纯 C/C++软件与混合语言软件, 特征库中可能包含混合语言项目的 C/C++部分. 在特征匹配过程中, 这些混合语言软件的 C/C++部分与目标代码的特征相似, 从而引发假阳性结果. r-cran-fs[39] (一个混合了 C 与 R 语言的软件) 在与 hermes[40]的对比中就出现了这样的假阳性情况.

由于 CAnalyzer 的误报主要集中于 TPL 家族内部的公共函数匹配, 即不同衍生库或复用库之间共享的通用函数实现. 从工程实践角度分析, 这类误报通常不会引发严重的安全或合规性判断错误, 原因是: (1) 被误判的库往往与真实复用库处于同一功能家族, 其许可证、依赖关系和安全风险往往高度一致; (2) 误报位置多为算法或工具性函数 (如字符串处理、数学运算等), 在工程层面并不影响后续的依赖追踪、漏洞定位或许可证分析结论. 因此, 这类误报更多影响检测报告的精度指标, 而非决策结论的可靠性.

● 假阴性 (*FN*) 分析. 所有的假阴性案例均源于 CAnalyzer 静态阈值的影响. CAnalyzer 引入了文件粒度和更为严苛的多重阈值标准来判定 TPL 的存在性. 虽然这一标准能有效减少误报, 但同时也可能忽略一些相似度较低但仍存在复用关系的 TPL, 导致假阴性.

● 实验结论 (RQ2). CAnalyzer 的精确率和召回率分别为 90.63% 和 86.57%. 通过引入文件粒度分析和严格的多重阈值标准来判定 TPL 的存在性, CAnalyzer 有效减少了假阳性. 此外, CAnalyzer 拥有更全面的 TPL 特征库, 提升了查全能力. 与 CENTRIS、TPLite 和 OSSFP 相比, 在精确率和召回率上表现更优.

4.5 消融实验

为了验证 RQ3, 我们设计了消融实验, 结合不同的特征库处理方式和 TPL 识别方法, 应用于 100 个项目, 分析这些改进对成分识别能力的影响. 我们将影响 SCA 准确性的因素分为两类: 特征库质量和 TPL 识别方法. 我们设置了经过 3 种处理方式的特征库: 无预处理 (DataBase)、第 1 阶段预处理 (DataBase-I) 和第 2 阶段预处理 (DataBase-II). 在 TPL 识别方法上, 分别采用单阈值 (Detection, 使用 α) 和多重阈值 (Detection-I, 使用 α、β、θ). 通过这 6 种组合实验, 分析不同设置对检测精确率的影响.

● 特征库质量的影响分析. 未经过预处理的特征库精确率较低 (16.66%–50.00%, 见表 5), 主要原因在于其包含大量公共函数 (38%), 这导致复用真实组件代码的库被误报, 如图 1 所示, 从而增加了假阳性. 如图 10 所示, 特征库经过每个处理阶段后, 精确率显著提升. 这表明, 每个阶段都有效地为公共函数找到合理归属, 剔除了 TPL 中非自身的函数, 降低了公共函数在特征库中的比例 (见表 6), 从而减少了其对 SCA 结果准确性的干扰.

表 5 不同影响因素组合下实验结果 (%)

序号	特征库类型	TPL识别方法	精确率	召回率
1	DataBase	Detection	16.66	96.29
2	DataBase	Detection-I	50.00	92.62
3	DataBase-I	Detection	27.12	92.59
4	DataBase-I	Detection-I	65.00	89.89
5	DataBase-II	Detection	37.08	90.74
6	DataBase-II	Detection-I	90.63	86.57

注: DataBase: 没有经过任何预处理的特征库; DataBase-I: 经过第1阶段预处理的特征库; DataBase-II: 经过第2阶段预处理的特征库; Detection: 未使用多重阈值的TPL检测逻辑; Detection-I: 使用多重阈值的TPL检测逻辑

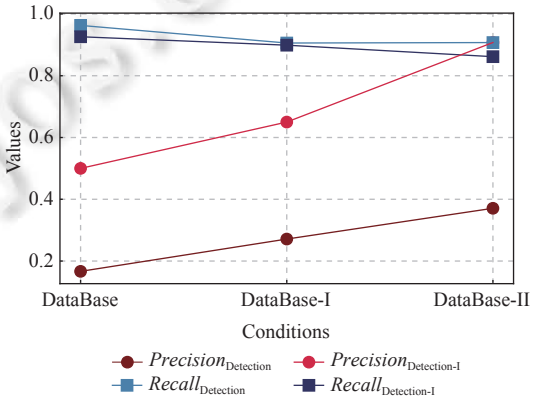


图 10 不同特征库条件下精确率和召回率趋势

● TPL 识别方法的影响分析. 如图 11 所示, 在 3 种特征库设置下, 单阈值检测的精确率均较低. 这是因为基于函数数量的单阈值只能判断目标代码与外部组件部分函数的重合, 难以准确评估整个库的复用情况. 相比之下, 多重阈值不仅关注函数级别的相似度, 还结合文件级别的相似度, 综合考虑目标代码与组件的重合比例, 更全面地反映组件的复用情况. 因此, 多重阈值在所有特征库设置下的精确率普遍较高 (50.00%–90.63%). 然而, 静态阈值的限制可能导致一些真实复用案例被排除, 从而损失部分召回率.

表 6 不同特征库的函数规模

函数规模	DataBase	DataBase-I	DataBase-II
标志函数数量	18 593 657	20 690 362	23 854 290
公共函数数量	11 453 633	5 373 301	2 209 373
公共函数占比 (%)	38	21	8
函数总数	30 047 290	26 063 663	26 063 663

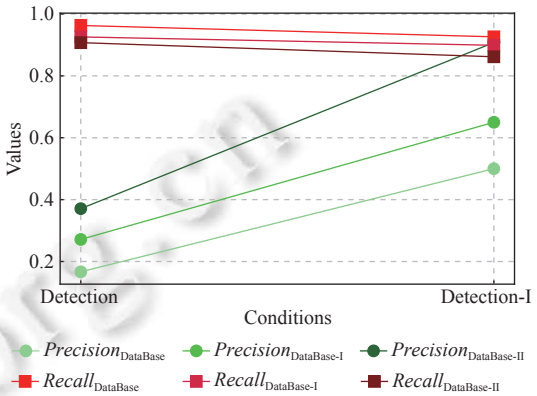


图 11 不同成分识别算法下精确率和召回率趋势

● 实验结论 (RQ3). 由于噪声干扰, 无预处理的特征库精确率较低 (16.66%–50.00%). 而预处理可以有效清理公共函数, 提高检测精确率. 使用多重阈值来匹配 TPL 能提升精确率, 但会损失部分召回率. 结合高质量特征库和多重阈值检测, 可最大程度减少假阳性和假阴性, 显著提升 SCA 准确性.

4.6 依赖检测评估实验

为了探究 RQ4, 我们设计了针对文件集 (由同一 TPL 所属文件构成) 之间依赖检测的实验, 以验证 CAnalyzer 在聚合到 TPL 粒度后能否准确构建 TPL 之间的依赖关系. 实验数据来源于 OpenHarmony 4.0 的 BUILD.gn 文件, 该文件详细定义了项目的构建规则和配置信息. 其中, “source”字段指定了构建模块所需的源文件, 而“deps”字段则列出了当前模块所依赖的其他模块或源文件, 从而形成了“source”文件集与“deps”文件集之间的依赖关系.

如表 3 所示, 我们解析了 14 873 个 BUILD.gn 文件, 从中提取了 12 277 个构建模块, 并筛选出同时包含非空“source”和“deps”字段的模块, 构建了 6 999 对模块依赖关系. 通过解析“source”字段获取模块的源文件集, 并根据“deps”字段识别直接依赖的文件集, 同时构造间接依赖关系. 例如, 若模块 A 的“deps”指向模块 B, 模块 B 的“deps”指向模块 C, 则 A 的“source”与 C 的“source”之间存在间接依赖. 由于路径变量解析失败或文件缺失, 最终保留 1 609 对直接依赖和 175 对间接依赖模块, 共形成 1 784 对模块依赖关系, 用于评估工具的依赖检测能力.

如表 7 所示, CAnalyzer 在检测文件集依赖关系时表现优异, 其精确率为 94.79%, 召回率为 98.99%. 在 1 784 对依赖关系中, 工具通过#include、extern、struct、class 和 enum 等多种方式准确检测了直接依赖中的 1 591 对, 以及全部间接依赖, 误报 97 对, 漏报 18 对. 在所有正确识别出的直接依赖中, 69.26% 的直接依赖由#include 建立, 其余 30.74% 的直接依赖是由 extern 语句建立起来的, 体现了 CAnalyzer 在依赖检测的全面性和准确性.

表 7 文件集间依赖关系检测实验结果

TP (直接依赖)		*TP* (间接依赖)			*FP*	*FN*	*Precision* (%)	*Recall* (%)
由#include建立	由extern建立	由struct建立	由class建立	由enum建立				
1 102	489	7	167	1	97	18	94.79	98.99

● 假阳性 (*FP*) 分析. 在 97 个假阳性中, 问题源于间接依赖检测的偏差. 由于采用字符串解析的方法, 系统会误将某些相似的字符串当作依赖关系. 例如, 假设 A 文件集中的某个函数 logInfo() 输出了一个字符串“UserData is

invalid", 而 B 文件集中存在一个类 UserData. 工具会误判为 A 文件集间接依赖了 B 文件集中的 UserData 类, 原因是 A 中的输出字符串包含了 B 中类名.

实际上, logInfo() 函数仅是将一个错误信息输出到日志中, 并没有真正使用 UserData 类. 尽管字符串中包含类名, 但并不意味着这两个文件之间存在依赖关系. 这种浅层匹配方法在复杂的代码环境中容易产生误判, 特别是当类名或对象名出现在日志输出、错误信息或字符串中时. 未来, 我们计划引入数据流图分析等更精确的方法, 以提高依赖关系识别的准确性.

● 假阴性 (*FN*) 分析. 假阴性现象主要由工具在解析 BUILD.gn 文件中的动态定义路径或引用时的局限性引起. 这些动态定义通常包括依赖变量的路径配置 (例如$root_path/some_file.cc), 或通过脚本生成的模块依赖列表 (例如 deps=[get_deps_from_script()]). 由于这些路径或引用的具体值需在构建时动态解析, 工具在静态分析阶段难以获取完整信息, 从而导致文件集不完整, 进而无法建立文件集之间的依赖关系. 这些问题凸显了在 TPL 依赖检测中准确划分 TPL 成分模块的重要性, 未来研究可以通过改进模块划分策略, 减少假阴性.

● 实验结论 (RQ4). CAnalyzer 能够准确检测文件集之间的依赖关系, 精确率为 94.79%, 召回率为 98.99%. 在假设 TPL 模块划分准确的前提下, 每个 TPL 模块可视为一个独立的文件集, 因此 CAnalyzer 同样能够高效识别和提取 TPL 之间的依赖关系.

4.7 效率对比实验

为了探究 RQ5, 我们评估 CAnalyzer 的效率, 与已有的 SCA 工具进行对比实验, 通过评估 SCA 技术两个执行时间: (1) 特征提取时间, 指的是对目标源代码提取特征的效率; (2) TPL 检测时间, 指的是基于特征库进行成分识别的效率. 由于源代码文件的大小与处理时间存在较强的相关性, 因此, 评估结果报告的是经过大小归一化处理的每个源代码项目的平均执行时间. OSSFP 提供商业版工具, 没有提供特征提取和 TPL 检测的独立时间, 因此我们对比其总时间.

如表 8 所示, 3 个工具 (CENTRIS、TPLite 和 CAnalyzer) 在特征提取阶段的执行时间相近, 分别为 67.99、109.32、67.99 s. 这是因为它们均采用函数粒度作为特征提取的基本单元, 且使用了高效的源代码解析工具. 具体而言, CENTRIS 和 CAnalyzer 基于 Ctags[30]进行特征提取, 而 TPLite 则使用 Tree-sitter[41], 两者均为主流且性能优越的解析器. 在 TPL 检测阶段, 检测时间与特征库规模密切相关: 特征库越大, 匹配耗时越长. CAnalyzer 维护了一个大规模的特征库, 涵盖来自 33 100 个 C/C++ TPL 的 30 047 290 条函数级特征, 远超 CENTRIS 所使用的特征库 (覆盖 10 294 个 C/C++ TPL). TPLite 的检测耗时最长, 达到 3 107.68 s, 主要原因在于其采用基于图的特征匹配算法, 该方法在结构匹配精度上具有优势, 但计算复杂度远高于基于哈希的相似性匹配策略. 相比之下, 商用工具 OSSFP 具有最短的检测时间 (68.73 s), 其采用查询式检测方式替代了对特征库的逐一比对, 显著提高了处理效率. 然而, 这种方式也带来了精确率和召回率的下降, 在检测效果上存在明显局限性.

表 8 CAnalyzer 与相关工具执行效率对比实验结果 (s)

工具	特征提取时间	TPL检测时间	总时间
CENTRIS	67.99	23.99	91.98
TPLite	109.32	3 107.68	3 217.00
OSSFP	—	—	68.73
CAnalyzer	67.99	969.45	1 037.44

注: "—"表示无法获取相应数据. OSSFP仅提供在线检测服务, 无法获得其内部执行过程, 因此无法分别统计特征提取时间和TPL检测时间

● 实验结论 (RQ5). CAnalyzer 在特征提取阶段表现最优, 耗时仅 67.99 s; 同时, 在基于大规模特征库 (包含 30 047 290 个函数特征) 执行 TPL 检测的条件下, 仍展现出具有竞争力的检测效率, 检测时间为 969.45 s.

5 讨 论

● 技术局限性. 尽管 CAnalyzer 在 C/C++ TPL 检测与依赖分析中取得了良好效果, 但仍存在以下 3 个方面的

局限性. (1) 阈值设定对检测性能的影响. 在软件库粒度的复用检测场景中, 阈值的选取直接影响精确率与召回率之间的权衡. 较为严格的阈值设置有助于提升检测结果的准确性, 但可能导致部分复用实例漏检, 从而降低召回率. 鉴于本文聚焦于软件库粒度的复用分析, 未采用较低阈值以避免引入过多误报. 未来工作中, 可根据实际应用需求, 灵活调整阈值组合, 以实现精确率与召回率的平衡. (2) 模块划分的模糊性问题. 当前版本的 CAnalyzer 采用经验性规则对模块进行划分, 在大多数情况下能够取得合理结果, 但对于部分结构复杂或交叉复用的代码文件, 可能出现划分边界不清的问题, 进而影响依赖关系的识别精度. 后续工作将探索更精细的模块划分方法, 例如基于函数调用图的分析手段, 以提升成分划分的准确性, 降低由划分误差引起的干扰. (3) 依赖关系检测的局限性. CAnalyzer 当前通过扫描“#include”“extern”等语句及标识符匹配方式进行依赖关系识别, 能够有效覆盖大部分源码级依赖. 然而, 该方法无法识别运行时可能触发的动态依赖; 同时, 在具有相同标识符名称的模块之间, 基于字符串的匹配机制可能引入误报. 为提升依赖分析的准确性, 后续研究将考虑引入控制流图或函数调用图等更具语义信息的分析手段.

● 有效性威胁. 本研究在评估 CAnalyzer 的过程中仍面临以下潜在有效性威胁: 首先, 为了验证 CAnalyzer 的准确性, 我们对数据集中项目的复用情况进行了人工核查. 然而, 由于部分项目缺乏正式的软件物料清单 (software bill of materials, SBOM), 且无法获得完整的开源依赖库列表, 可能存在未纳入标准集的真实复用实例. 这类隐性复用的遗漏可能导致 CAnalyzer 及对比工具的精确率评估出现偏差. 其次, 我们将 CAnalyzer 与 CENTRIS、TPLite 和 OSSFP 这 3 种代表性工具进行对比分析, 旨在评估在多源托管仓库构建的 C/C++ TPL 特征库场景下, 舍弃函数诞生时间、结合特征库预处理与多重阈值策略的方法, 在软件库粒度复用检测中的有效性表现. 需要指出的是, 本研究并不否定上述工具的整体性能, 而是希望通过对比说明 CAnalyzer 在特定检测场景下具备更优的检测效果.

6 相关工作

6.1 软件成分分析

软件成分分析 (SCA) 技术用于识别开源软件及商业软件中涉及的源代码、模块和框架等组成成分, 分析其构成与依赖关系, 并检测是否存在已知的安全漏洞和潜在的许可证合规问题, 确保软件供应链中组件的安全. 随着现代软件开发对第三方库依赖程度的不断提升, SCA 在软件开发生命周期中发挥着日益关键的作用.

根据软件成分分析的输入形式和分析目标的不同, SCA 技术可分为源到源 (source-to-source, S2S) 分析和二进制到源 (binary-to-source, B2S) 分析两类. B2S 分析指从已编译的二进制文件中识别其使用的外部组件, 主要应用于逆向工程、安全审计和漏洞分析等场景. 典型工具包括 OSSPolice[5]、BinPro[42]、BinaryAI[43]等. 相比之下, S2S 分析通过直接解析源代码, 识别项目中的依赖关系、导入的库文件和头文件等, 从而识别引用的外部组件和第三方库, 广泛应用于开源治理、许可证合规检测等领域. 本研究聚焦于 C/C++语言项目的 S2S 成分分析方法, 旨在提升软件组件的可见性和可管理性. 因此, 本文将重点回顾与 S2S 相关的研究工作.

近年来, 针对不同应用场景, S2S 领域发展出多种 SCA 工具. 部分商业工具, 如 OWASP[44]和 Sonatype[45]依赖 SBOM 识别项目中使用的第三方库, 但由于开源 C/C++代码库普遍缺乏标准化的 SBOM 文件, 这类工具在源代码检测中的适用性有限. 另一些工具侧重于识别代码中易受攻击的脆弱部分[4,46−53], 如漏洞函数匹配等方法, 虽能有效定位潜在漏洞, 但仅针对特定易受攻击的 TPL, 无法提供完整的成分分析, 覆盖范围有限. 此外, 基于代码克隆检测的工具 (如 BlackDuck[9]) 通过对比目标代码与 TPL 函数的相似性判断是否存在复用. 然而, 并非所有函数都具有唯一性, 一些公共函数可能被多个 TPL 共享, 易导致复用分析中的误判, 引发假阳性. 为提升 SCA 技术的准确性, 亟需在特征库构建阶段提取具有唯一性判别能力的 TPL 特征, 以降低公共函数干扰.

为解决公共函数干扰问题, CENTRIS[13]基于“被复用的 TPL 通常具有较早函数诞生时间”的假设, 通过分析函数的诞生时间推导 TPL 间的依赖关系, 以区分外部引入函数与自身函数, 并剔除冗余外部函数. TPLite[14]在此基础上引入基于 PageRank 算法[35]的中心性过滤器, 用于识别并过滤无效依赖, 从而弥补函数诞生时间方法的局限. OSSFP[15]将 TPL 函数划分为克隆函数、支持函数、通用函数和核心函数, 并通过设定阈值和函数诞生时间过滤

前 3 类, 仅保留核心函数构建特征. 然而, 不同 TPL 中公共或支持函数的占比差异较大, 静态阈值可能误删有价值的函数特征. 上述方法均依赖函数诞生时间来处理公共函数归属问题, 但正如本文第 2.1 节所述, 函数诞生时间易受版本控制工具和项目版本管理策略影响, 可能无法准确获取甚至缺失, 进而影响 SCA 检测的准确性.

相较于上述方法, 本文提出的方案避免了函数诞生时间带来的不确定性. CAnalyzer 通过明确标识信息 (如函数路径) 过滤与特定 TPL 无关的噪声函数, 并将相似 TPL 划分为“家族”, 家族内成员共享部分公共函数, 同时保留各自独有的标志函数. 基于被复用 TPL 中标志函数占比较低的假设, CAnalyzer 通过分析共享函数的复用关系, 更有效地区分 TPL 自身代码与外部引入代码, 从而确保每个 TPL 特征准确反映其独有的代码结构. 该方法克服了时间信息缺失或不准确带来的局限.

在目标代码成分识别方面, CENTRIS 和 TPLite 设定函数数量阈值, 当目标代码与 TPL 相同的函数数量达到阈值时, 认为 TPL 是目标代码的有效成分. 而 OSSFP 则完全放弃阈值, 只要有一个函数相同即视为 TPL 存在. 尽管它们在处理代码片段引用时表现较好, 但在软件库粒度复用检测的场景中, 容易因部分函数相同而错误地判定 TPL 的存在, 从而增加假阳性. 为了避免这一问题. CAnalyzer 采用了多重阈值策略, 若函数复用比例、文件复用比例和库复用比例均达到预设阈值, 则确认该 TPL 为目标代码的有效成分, 以满足软件库粒度复用检测的需求.

CNEPS[54]为 C/C++源码中引入的 TPL 生成依赖视图. 其基本流程是: 首先将目标代码划分为模块, 每个模块由一个头文件和若干源文件组成; 然后基于成分分析结果选取主导组件作为依赖图中的节点; 最后, 通过扫描模块中的#include 指令并与其他模块的.h 文件对比, 建立模块间的依赖关系. 然而, 该方法存在一定的局限性. 首先, CNEPS 在遇到头文件名冲突时, 其准确性可能显著下降, 导致依赖关系识别错误. 其次, CNEPS 基于 CENTRIS 的成分分析结果存在偏差, CNEPS 在确定模块的主导组件时, 可能会放大这些不确定性. 例如, 即使模块中仅有一个函数与某个组件相同, CNEPS 也可能将该组件错误地标定为主导组件, 进而造成节点冗余和依赖边误判. 此外, 由于 CNEPS 的成分分析需要在每个模块上多次执行, 效率较低, 在大规模软件项目上成本较高. 相比之下, 本文提出的 CAnalyzer 基于一次性成分分析结果, 通过目录结构与显式元信息对 TPL 模块进行划分, 并将依赖关系建模为文件集之间的关系, 这一方法显著降低了计算开销, 并在构建依赖图时同时考虑#include 与 extern 指令; 当存在头文件冲突时, CAnalyzer 进一步利用标识符验证识别间接依赖, 从而避免了 CNEPS 面临的精度问题. 本文中没有将 CAnalyzer 与 CNEPS 进行直接对比是由于实验可行性的限制, 其环境配置与输入预处理要求与我们构建的数据集不完全兼容. 因此, 我们通过方法特性和局限性进行系统性讨论. 未来, 我们计划在统一基准条件下进一步探索与 CNEPS 的对比, 并解决其环境配置和输入预处理与我们数据集不兼容的问题.

6.2 代码克隆检测

代码克隆检测是指在软件系统内部或不同系统之间定位相同或相似的代码片段, 这些代码片段被称为“克隆”. 克隆通常是在开发者通过复制、粘贴和修改已有代码来实现代码复用时产生的. 随着软件系统日益庞大和复杂, 代码的重复和共享成为一种常见现象. 代码克隆不仅可能导致冗余的代码库和增加维护成本, 还可能在软件安全性、性能优化等方面带来挑战. 因此, 代码克隆检测在软件工程领域, 尤其是 SCA 中, 扮演着重要角色.

代码克隆分为 4 种类型[55]. 基于文本相似性, 区分 Type I、Type II 和 Type III 型代码克隆. 如果两个代码片段的功能相同或相似, 即它们具有相似的前置条件和后置条件, 则称它们为语义克隆 (Type IV). 在软件成分分析中, 我们关注的是目标代码是否包含与特定 TPL 相似或相同的源代码, 以进一步判断是否复用了该 TPL. 因此, 我们更关注代码的结构相似性, 而非语义上的变动. 在这一过程中, 我们主要关注的是 Type I 和 Type II 型代码克隆.

目前, 有许多工具可以检测 Type I 和 Type II 类型的克隆. 其中, 基于文本的工具中, Johnson[56]使用指纹技术对源代码进行比较, Ducasse 等人[57]则使用动态模式匹配进行行级比较. NiCad[58]是一种混合型工具, 采用最长公共子序列算法进行代码比较. 基于令牌的工具如 SourcererCC[10], 核心思想是将源代码转换为令牌序列, 这些令牌通常包括关键字、标识符、操作符和分隔符等语言元素. 通过比较令牌序列的相似度, 来识别代码克隆. 此外, 基于树和程序依赖图 (PDG) 的工具[59,60], 在语法和语义识别方面相较于文本方法具有更强的能力. 然而, 这些方法的性能开销较大, 难以在大规模代码库中高效应用.

为此, 我们采用了基于签名的代码克隆检测方法[4]. 具体而言, 我们将每个函数转换为哈希值, 并通过比较这些哈希值来检测目标项目中是否存在与预先收集的代码库中相似或重复的函数. 在面对微小代码变动时, 与基于文本和基于令牌的方法相比, 这种方法能够保持较高的准确性和健壮性, 因此更适用于软件成分分析.

如果预先收集的代码库中的函数缺乏代表性, 直接将这些函数映射到源库并视作目标项目的复用 TPL, 可能会导致大量假阳性; 如果代码库不完整, SCA 过程可能遗漏 TPL 成分, 导致假阴性. 因此, 在应用代码克隆检测算法进行成分识别之前, 必须构建一个全面的 TPL 代码库, 并保留能够充分表征 TPL 特征的函数. 本文设计并实现了一种高效且灵活的数据收集机制. 该机制涵盖了操作系统中央仓库、应用程序级包管理器以及代码仓库这 3 个层面, 选择了 15 个托管仓库作为 C/C++ TPL 的来源. 通过整合这些托管仓库提供的数据, 我们从 33 100 个 C/C++ TPL 中提取了 30 047 290 个函数特征, 进而构建大规模覆盖全面 C/C++ TPL 的特征库. 同时, 引入预处理步骤, 确保特征库中每个 TPL 的函数特征能够唯一标识 TPL.

7 总 结

本文提出并实现了针对 C/C++源代码的软件成分分析工具 CAnalyzer, 面向软件库粒度的复用检测场景, 旨在精准识别目标软件源代码中依赖的 C/C++ TPL 及其之间依赖关系. CAnalyzer 通过构建大规模、高质量的 TPL 特征并引入预处理机制, 有效缓解了公共函数对检测结果的干扰. 在此基础上, 进一步引入多重阈值策略, 增强了工具对实际工程场景的适应性, 并设计了基于文件间关系的依赖分析方法. 实验结果表明, CAnalyzer 在 TPL 识别准确性方面优于代表性工具 CENTRIS、TPLite 和 OSSFP, 展现出良好的检测效果与性能表现. 作为对现有 SCA 技术的系统性增强和适用性提升, CAnalyzer 在提高开源项目成分识别精度与依赖关系建模能力方面表现优越, 为开源软件的透明管理与软件供应链安全保障提供了有力支持. 未来, 我们将进一步拓展 CAnalyzer 在其他程序语言项目中的适用性, 并探索更高效的特征提取、特征库构建与依赖建模机制, 以持续提升 SCA 技术在实际软件开发场景中的实用价值.

References

[1] The GitHub Blog—Thank you for 100 million repositories. 2025. https://github.blog/news-insights/company-news/100m-repos/

[2] Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J. DéjàVu: A map of code duplicates on GitHub. Proc. of the ACM on Programming Languages, 2017, 1(OOPSLA): 84. [doi: 10.1145/3133908]

[3] Li HZ, Kwon H, Kwon J, Lee H. CLORIFI: Software vulnerability discovery using code clone verification. Concurrency and Computation: Practice and Experience, 2016, 28(6): 1900–1917. [doi: 10.1002/cpe.3532]

[4] Kim S, Woo S, Lee H, Oh H. VUDDY: A scalable approach for vulnerable code clone discovery. In: Proc. of the 38th IEEE Symp. on Security and Privacy (SP). San Jose: IEEE, 2017. 595–614. [doi: 10.1109/SP.2017.62]

[5] Duan RA, Bijlani A, Xu M, Kim T, Lee WK. 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] Kim S, Lee H. Software systems at risk: An empirical study of cloned vulnerabilities in practice. Computers & Security, 2018, 77: 720–736. [doi: 10.1016/j.cose.2018.02.007]

[7] Alqahtani SS, Eghan EE, Rilling J. Recovering semantic traceability links between APIs and security vulnerabilities: An ontological modeling approach. In: Proc. of the 2017 IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST). Tokyo: IEEE, 2017. 80–91. [doi: 10.1109/ICST.2017.15]

[8] Zhan X, Fan LL, Liu TM, Chen S, Li L, Wang HY, Xu YF, Luo XP, Liu Y. Automated third-party library detection for Android applications: Are we there yet? In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. ACM, 2020. 919–930. [doi: 10.1145/3324884.3416582]

[9] BlackDuck. True scale application security for your software. 2025. https://www.blackduck.com/

[10] Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV. SourcererCC: Scaling code clone detection to big-code. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: ACM, 2016. 1157–1168. [doi: 10.1145/2884781.2884877]

[11] Scantist, an open source management platform. 2025. https://app.scantist.io/

[12] 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]

[13] Woo S, Park S, Kim S, Lee H, Oh H. CENTRIS: A precise and scalable approach for identifying modified open-source software reuse. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 860–872. [doi: 10.1109/ICSE43902.2021.00083]

[14] Jiang L, Yuan HC, Tang QY, Nie S, Wu S, Zhang YQ. Third-party library dependency for large-scale SCA in the C/C++ ecosystem: How far are we? In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 1383–1395. [doi: 10.1145/3597926.3598143]

[15] Wu JH, Xu ZZ, Tang W, Zhang L, Wu YM, Liu CY, Sun KR, Zhao LD, Liu Y. OSSFP: Precise and scalable C/C++ third-party library detection using fingerprinting functions. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 270–282. [doi: 10.1109/ICSE48619.2023.00034]

[16] Snyk. 2023 state of open source security report. 2023. https://go.snyk.io/state-of-open-source-security-report-2023.html

[17] Xia BM, Bi TT, Xing ZC, Lu QH, Zhu LM. An empirical study on software bill of materials: Where we stand and the road ahead. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 2630–2642. [doi: 10.1109/ICSE48619.2023.00219]

[18] Zimmermann M, Staicu CA, Tenny C, Pradel M. Small world with high risks: A study of security threats in the npm ecosystem. In: Proc. of the 28th USENIX Security Symp. Santa Clara: USENIX, 2019. 995–1010.

[19] Sonatype. State of the software supply chain. 2025. https://www.sonatype.com/state-of-the-software-supply-chain/Introduction

[20] Libaiff. aifftools. 2007. https://sourceforge.net/projects/aifftools/files/

[21] SourceForge. The complete software platform. 2025. https://sourceforge.net/

[22] GNU. Termcap. 2025. ftp://ftp.gnu.org/gnu/termcap

[23] Google. SwiftShader. 2025. https://github.com/google/swiftshader

[24] SourceForge Repository. 7-Zip. 2025. https://sourceforge.net/projects/sevenzip/files/7-Zip/

[25] ip7z. 7zip. 2025. https://github.com/ip7z/7zip

[26] p7zip-project. p7zip. 2025. https://github.com/p7zip-project/p7zip

[27] LinuxDevices. Cisco settles with FSF on GPL violations. 2009. https://linuxdevices.org/cisco-settles-with-fsf-on-gpl-violations/

[28] ZDNet. VMware sued for failure to comply with Linux license. 2015. https://www.zdnet.com/article/vmware-sued-for-failure-to-comply-with-linuxs-license/

[29] Tang W, Xu ZZ, Liu CW, Wu JH, Yang SG, Li Y, Luo P, Liu Y. Towards understanding third-party library dependency in C/C++ ecosystem. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 106. [doi: 10.1145/3551349.3560432]

[30] Universal Ctags. Ctags. 2025. https://github.com/universal-ctags/

[31] Oliver J, Cheng C, Chen YG. TLSH—A locality sensitive hash. In: Proc. of the 4th Cybercrime and Trustworthy Computing Workshop. Sydney: IEEE, 2013. 7–13. [doi: 10.1109/CTC.2013.9]

[32] GitHub Repository. Freetype. 2025. https://github.com/freetype/freetype

[33] GCC, the GNU compiler collection. 2025. https://gcc.gnu.org/

[34] *F*-score. 2025. https://en.wikipedia.org/wiki/F-score

[35] Page L, Brin S. The PageRank citation ranking: Bringing order to the Web. Stanford InfoLab, 1999. [doi: 10.1007/978-3-319-08789-4_10]

[36] BYVoid. OpenCC. 2025. https://github.com/BYVoid/OpenCC

[37] pybind. pybind11. 2025. https://github.com/pybind/pybind11

[38] Anigmetov A. hera. 2025. https://github.com/anigmetov/her

[39] Debian Salsa. r-cran-fs. 2025. https://salsa.debian.org/r-pkg-team/r-cran-fs

[40] Facebook. Hermes. 2025. https://github.com/facebook/hermes

[41] Tree-sitter. tree-sitter. 2025. https://github.com/tree-sitter/tree-sitte

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

[43] Jiang L, An JW, Huang HH, Tang QY, Nie S, Wu S, Zhang YQ. BinaryAI: Binary software composition analysis via intelligent binary source code matching. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 224. [doi: 10.1145/3597503.3639100]

[44] OWASP. OWASP dependency-track. 2025. https://owasp.org/www-project-dependency-track

[45] Sonatype. OSS index analyzer. 2025. https://jeremylong.github.io/DependencyCheck/data/ossindex.html

[46] Xiao Y, Chen BH, Yu CD, Xu ZZ, Yuan ZM, Li F, Liu BH, Liu Y, Huo W, Zou W, Shi WC. MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In: Proc. of the 29th USENIX Security Symp. (USENIX Security 20). USENIX, 2020. 1165–1182.

[47] Li Z, Lu S, Myagmar S, Zhou Y. CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. on Software Engineering, 2006, 32(3): 176–192. [doi: 10.1109/TSE.2006.28]

[48] Jiang LX, Su ZD, Chiu E. Context-based detection of clone-related bugs. In: Proc. of the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. Dubrovnik: ACM, 2007. 55–64. [doi: 10.1145/1287624.1287634]

[49] Gabel M, Yang JF, Yu Y, Goldszmidt M, Su ZD. Scalable and systematic detection of buggy inconsistencies in source code. In: Proc. of the 2010 ACM Int'l Conf. on Object-oriented Programming Systems Languages and Applications. Reno: ACM, 2010. 175–190. [doi: 10.1145/1869459.1869475]

[50] Pham NH, Nguyen TT, Nguyen HA, Nguyen TN. Detection of recurring software vulnerabilities. In: Proc. of the 25th IEEE/ACM Int'l Conf. on Automated Software Engineering. Antwerp: ACM, 2010. 447–456. [doi: 10.1145/1858996.1859089]

[51] Li YK, Xue YX, Chen HX, Wu XH, Zhang C, Xie XF, Wang HJ, Liu Y. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 533–544. [doi: 10.1145/3338906.3338975]

[52] Xu ZZ, Chen BH, Chandramohan M, Liu Y, Song F. SPAIN: Security patch analysis for binaries towards understanding the pain and pills. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Buenos Aires: IEEE, 2017. 462–472. [doi: 10.1109/ICSE.2017.49]

[53] Jang J, Agrawal A, Brumley D. ReDeBug: Finding unpatched code clones in entire OS distributions. In: Proc. of the 2012 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2012. 48–62. [doi: 10.1109/SP.2012.13]

[54] Na Y, Woo S, Lee J, Lee H. CNEPS: A precise approach for examining dependencies among third-party C/C++ open-source components. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 236. [doi: 10.1145/3597503.3639209]

[55] Ain QU, Butt WH, Anwar MW, Azam F, Maqbool B. A systematic review on code clone detection. IEEE Access, 2019, 7: 86121–86144. [doi: 10.1109/ACCESS.2019.2918202]

[56] Johnson JH. Substring matching for clone detection and change tracking. In: Proc. of the 1994 Int'l Conf. on Software Maintenance. Victoria: IEEE, 1994. 120–126. [doi: 10.1109/ICSM.1994.336783]

[57] Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. In: Proc. of the 1999 IEEE Int'l Conf. on Software Maintenance (ICSM 1999). Oxford: IEEE, 1999. 109–118. [doi: 10.1109/ICSM.1999.792593]

[58] Roy CK, Cordy JR. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: Proc. of the 16th IEEE Int'l Conf. on Program Comprehension. Amsterdam: IEEE, 2008. 172–181. [doi: 10.1109/ICPC.2008.41]

[59] Liu C, Chen C, Han JW, Yu PS. GPLAG: Detection of software plagiarism by program dependence graph analysis. In: Proc. of the 12th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. Philadelphia: ACM, 2006. 872–881. [doi: 10.1145/1150402.1150522]

[60] Jiang LX, Misherghi G, Su ZD, Glondu S. DECKARD: Scalable and accurate tree-based detection of code clones. In: Proc. of the 29th Int'l Conf. on Software Engineering (ICSE 2007). Minneapolis: IEEE, 2007. 96–105. [doi: 10.1109/ICSE.2007.30]

作者简介

徐美秋, 博士生, CCF 学生会员, 主要研究领域为开源软件供应链安全, 软件成分分析技术.

王舒婧, 硕士生, 主要研究领域为开源软件供应链安全, 软件成分分析技术.

王莹, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为智能软件开发技术, 开源软件供应链安全, 开源治理技术, 国产平台软件迁移适配技术.

于海, 博士, 副教授, 主要研究领域为软件测试, 软件重构及软件测试技术, 软件体系结构, 复杂网络理论, 混沌加密技术.

朱志良, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为混沌加密技术, 复杂网络理论, 软件测试技术.