

CentSCA: 基于中心性分析的 C/C++软件成分分析方法

胡雨涛^{1),2),4)} 董家骏^{1),2),4)} 孙铭远²⁾ 张云鹤²⁾ 吴月明^{1),2),3),4)} 邹德清^{1),2),3),4)}

¹⁾(分布式系统安全湖北省重点实验室 武汉 430074)

²⁾(华中科技大学 网络空间安全学院 武汉 430074)

³⁾(大数据技术与系统国家地方联合工程研究中心 服务计算技术与系统教育部重点实验室 武汉 430074)

⁴⁾(金银湖实验室 武汉 430074)

摘要 随着开源项目的广泛应用,开发人员对第三方库(Third Party Library, TPL)的依赖日益加深,这在加速软件开发的同时,也可能引入 TPL 中的漏洞,给软件安全带来隐患。因此,准确识别项目依赖的第三方库成为软件工程领域的重要研究课题。当前,面向 C/C++ 语言的软件成分分析(Software Composition Analysis, SCA)方法主要分为基于 TPL 全局特征和基于 TPL 关键特征两类。其中,基于全局特征的方法虽然信息覆盖全面,但由于受到噪声干扰,存在特征鲁棒性差且检测开销大等局限性;为了捕捉 TPL 的细节特征以增强检测鲁棒性,基于关键特征的方法筛选具有 TPL 语义代表性的核心函数作为关键特征,但是由于缺乏语义信息的考虑,难以准确定位核心函数。针对上述问题,提出了一种基于中心性分析的软件成分分析方法(CentSCA)。该方法基于代码属性分别过滤重复函数、简单函数和常见函数等非核心函数,然后利用中心性分析方法对函数调用图中关键节点(函数)的捕捉能力,定位具有 TPL 核心语义的关键函数,进而实现软件依赖检测。为了验证 CentSCA 的有效性,本文构建了一个包含 510 个开源项目和 1016 条依赖关系的真实数据集,并对其进行了详细实验。实验结果显示, CentSCA 取得了 0.72 的 F1 分数,相较于现有的 SCA 工具 CENTRIS 和 OSSFP,分别提升了 26% 和 5%。在真实项目的案例研究中, CentSCA 相较于 OSSFP 使供应链漏洞检测的误报数量减少了 35 个,进一步验证了其在实际场景下的实用性。

关键词 软件成分分析; 中心性分析; 代码语义分析; 开源软件; 软件安全

中图法分类号 TP311 DOI号 10.11897/SP.J.1016.2026.00679

CentSCA: A Software Composition Analysis Method for C/C++ Project Based on Centrality Analysis

HU Yu-Tao^{1),2),4)} DONG Jia-Jun^{1),2),4)} SUN Ming-Yuan²⁾ ZHANG Yun-He²⁾
WU Yue-Ming^{1),2),3),4)} ZOU De-Qing^{1),2),3),4)}

¹⁾(Hubei Key Laboratory of Distributed System Security, Wuhan 430074)

²⁾(School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074)

³⁾(National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Wuhan 430074)

⁴⁾(Jinyinhu Laboratory, Wuhan 430074)

Abstract With the widespread adoption of open-source projects, software developers have become increasingly reliant on third-party libraries (TPLs). This reliance greatly accelerates software

development, yet it also introduces potential vulnerabilities contained within these TPLs, thereby posing significant risks to software security. Consequently, accurately identifying the third-party libraries used in a software project has become a critical research topic in the field of software engineering. Current Software Composition Analysis (SCA) techniques for C and C++ programs can be broadly categorized into two types: global-feature-based and key-feature-based approaches. Global-feature-based methods typically extract and compare comprehensive representations of entire TPLs, which enables wide coverage but often suffers from noise interference. As a result, such methods tend to exhibit poor feature robustness and high computational overhead during detection. In contrast, key-feature-based methods aim to enhance detection robustness by selecting core functions that are semantically representative of a TPL. However, existing approaches often overlook the semantic structure of functions and fail to effectively distinguish truly representative core functions from non-essential ones, leading to inaccurate dependency identification. To address these limitations, this paper proposes CentSCA, a novel C/C++ SCA method based on centrality analysis. CentSCA leverages code-attribute filtering to eliminate non-core functions, including duplicate, trivial, and widely shared common functions. It then constructs a function call graph for each library and employs centrality analysis to identify key nodes that carry the core semantics of a TPL. These central nodes are treated as semantically meaningful key functions that best represent the library's unique characteristics. By matching these core functions with those in target projects, CentSCA achieves accurate dependency detection with improved robustness and scalability. To validate the effectiveness of CentSCA, we constructed a large-scale real-world dataset containing 510 open-source projects and 1,016 dependency relationships. Extensive experiments were conducted to evaluate the detection performance against representative state-of-the-art tools, including CENTRIS and OSSFP. The experimental results demonstrate that CentSCA achieves an F1 score of 0.72, outperforming CENTRIS and OSSFP by 26 percent and 5 percent, respectively. Furthermore, in a real-world case study of software supply chain vulnerability detection, CentSCA reduced the number of false positives by 35 compared with OSSFP, further confirming its practical utility in real-world security assessment scenarios. In summary, CentSCA provides an effective and interpretable framework for fine-grained third-party library identification in C and C++ software ecosystems. By integrating structural centrality analysis with semantic-aware function selection, it bridges the gap between coarse-grained and semantic-level SCA methods. This research contributes both a theoretical advancement in dependency detection methodology and a practical tool for improving the security and maintainability of software supply chains.

Keywords software composition analysis; centrality analysis; code semantics analysis; open source software; software security

1 引 言

在开源项目蓬勃发展的当下,越来越多的开发人员选择引用第三方库(Third party Library, TPL)来复用已有功能,这一方式有效避免了重复开发,极大地缩短了软件开发周期^[1]。然而,过度依赖 TPL 也带来了不容忽视的安全问题, TPL 中潜藏的漏洞可能会被引入项目,给软件安全埋下隐患^[2]。因此,

明确项目所依赖的 TPL 清单,对开发人员及时应对安全问题至关重要。软件成分分析(SCA)技术应运而生,旨在帮助开发人员检测项目依赖的 TPLs,提升系统的可维护性与安全性。

多数编程语言,如 JAVA、Python 和 JavaScript 等,都配备了官方的项目管理工具,像 Maven、PyPI 和 npm 等,这些工具能够生成软件物料清单(Software Bill of Materials, SBOM),清晰记录项目

依赖的第三方库。但 C/C++ 语言由于缺乏官方项目管理工具, 只能借助代码克隆检测来判断项目对 TPL 的依赖, 即通过判断项目是否复用了 TPL 的源代码来确定依赖关系。现有的面向 C/C++ 语言的 SCA 方法, 依据 TPL 特征的粒度可分为基于 TPL 全局特征和基于 TPL 关键特征两类, 它们均通过匹配 TPL 和待测项目的特征来进行软件依赖检测。

基于 TPL 全局特征的方法试图从各个角度获取 TPL 的全部信息, 虽然信息覆盖全面, 但容易受到大量噪声干扰, 导致检测鲁棒性差, 且检测所需的时间和空间开销较大。以 OSSPolice^[3]为例, 它从文件目录角度出发, 将项目的文件目录结构和文件名构建为 TPL 全局特征, 然而, 这些特征极易被更改, 鲁棒性欠佳, 导致检测漏报率较高。CENTRIS^[4]则从源代码角度入手, 将 TPL 全部原创函数作为全局特征, 通过比较项目间相同原创函数的占比与预设相似度阈值来检测依赖关系。CENTRIS 存在两个主要局限: 其一, 检测效果对相似度阈值的选择极为敏感, 鲁棒性不足, 阈值设置过高会导致漏报, 过低则易受非重要函数(即特征噪声)干扰产生误报; 其二, 需要对项目间的全部函数两两比较, 复杂度较高。尽管通过哈希值匹配时间开销有所降低, 但难以扩展处理复杂代码克隆(Type 2-4 克隆)。

基于 TPL 关键特征的 SCA 方法, 旨在通过捕捉 TPL 的核心细节特征提升检测鲁棒性。例如, OSSFP^[5]将具有项目代表性语义的函数视为核心函数(即 TPL 的关键特征), 一旦项目克隆引用了 TPL 的关键函数, 即判定其依赖于该 TPL。具体而言, OSSFP 先去除 TPL 库中的重复函数, 再通过代码复杂度筛选掉简单函数, 利用词频-逆文档频率(tf-idf)过滤常见函数, 最后将剩余函数认定为核心函数。可以看出, OSSFP 在筛选 TPL 关键函数的过程中, 没有任何对于代码语义信息的考量。核心函数作为实现 TPL 主要功能逻辑及算法的函数, 在缺乏语义信息分析的情况下, 无法保证定位的准确性。

为了验证这一观点, 本文在第 3.2 节中以现实世界的开源项目为例, 使用 OSSFP 检测其核心函数, 并对结果进行了深入分析。结果显示, OSSFP 所识别出的核心函数中, 包含了一些具备特定逻辑结构、不常见且由仓库作者原创的函数(如第 3.2 节中提到的测试函数)。然而, 从语义上看, 这些函数与 TPL 核心功能逻辑及算法无直接关联, 不应被视为具有 TPL 核心语义的关键函数。这一现象表明, 仅依据函数的原创性、逻辑结构和通用性等表层特

征, 难以准确识别真正的核心函数; 还需引入对代码语义信息的更深入理解与分析。

由此, 引出本文的研究挑战: 如何依据函数的语义特征精准定位能够代表 TPL 关键语义的核心函数?

针对上述挑战, 本文提出了一种基于中心性分析的软件成分分析方法(CentSCA)。该方法从语法和语义两个维度综合考量函数的重要性。具体而言, 在语法层面, CentSCA 通过量化 TPL 中函数的语法特征, 依次过滤掉重复函数、简单函数和常见函数, 得到待选核心函数集 V_1 。在语义层面, 方法构建 TPL 的函数调用图以建模函数间的语义关联, 并利用中心性分析计算每个函数在调用图中的中心性分数, 进一步通过设定阈值筛选得分较高的函数, 构成待选核心函数集 V_2 。最终, CentSCA 取 V_1 与 V_2 的交集作为 TPL 的核心函数集合。选择交集的原因将在第 4.4.2 节详细说明; 中心性分析中所采用的阈值则来源于第 5.2 节对照实验中得到的最优参数设置。

为验证 CentSCA 在 C/C++ 项目软件成分分析中的有效性, 本文构建了一个包含真实依赖标注的数据集, 并开展了系统性实验。数据集基于 GitHub 上星标数排名前 100 的仓库, 筛选出 56 个有效项目及其依赖的第三方库, 通过解析依赖声明构建了 510 个项目样本。随后, 我们对样本中的函数级依赖关系进行了专家交叉验证标注, 最终形成包含 1016 条依赖关系的高质量数据集。

实验结果表明: 1) 在经过参数优化后, CentSCA 的 F1 分数达到 0.72; 2) 消融实验证实了 CentSCA 各关键模块(如语法过滤、语义分析与中心性分析)的必要性; 3) 与当前代表性工具 CENTRIS 和 OSSFP 相比, CentSCA 的依赖检测性能分别提升了 26% 和 5%; 4) 在真实项目中, 基于 CentSCA 的依赖识别相比 OSSFP, 使供应链漏洞检测的误报数量减少了 35 个, 进一步验证了其在实际场景中的有效性。

在此基础上, 为进一步评估核心函数识别对软件成分分析下游任务的影响, 本文选取现实世界中的开源仓库 filament^①作为案例, 分别采用 OSSFP 和 CentSCA 方法进行依赖检测, 并结合 Snyk^②漏洞数据库对识别出的依赖关系进行供应链漏洞检测。

① filament, <https://github.com/google/filament>

② snyk, <https://github.com/snyk/cli>

实验发现, OSSFP 由于缺乏语义分析, 错误地将 TPL 中一些非核心函数识别为核心函数, 而这些非核心函数也在 filament 中出现, 导致 yaml-cpp^①, mruby^②, libgit2^③等存在漏洞的 TPL 被误判为依赖项, 进而在 Snyk 查询中造成了 35 个漏洞误报。相较而言, CentSCA 凭借对核心函数的准确识别, 有效避免了上述依赖误报, 从而显著减少供应链漏洞误报数量。

本文的主要贡献如下:

(1) 提出了一种基于中心性分析的关键函数定位方法, 通过加强代码语义的考虑, 利用函数调用图的代码语义表征能力和中心性分析方法对图结构中关键节点(函数)的捕捉能力, 精准定位具有 TPL 语义代表性的关键函数;

(2) 实现并部署了基于中心性分析的软件成分分析框架 CentSCA, 并将其应用于包含 1016 条软件依赖关系的真实 C/C++ 项目依赖数据集;

(3) 经详细实验评估, CentSCA 能够正确检出数据集中的 715 条依赖关系, 准确率达到 0.74, 召回率为 0.70, F1 分数为 0.72。相较于最先进的 C++ 语言 SCA 工具(CENTRIS 和 OSSFP), F1 分数分别提升 26% 和 5%。

本文第 2 节介绍软件成分分析的相关工作及研究进展; 第 3 节介绍本文的研究背景, 包括相关函数概念的定义和研究动机; 第 4 节介绍基于中心性分析的软件成分分析方法的具体方法设计和实现; 第 5 节分别通过参数实验、消融实验和对比实验来评估 CentSCA 在软件成分分析方面的有效性; 最后总结全文, 并展望未来工作。

2 相关工作

在软件开发过程中, 第三方库的使用愈发普遍。Lopos 等人^[1]的研究表明, 为提高开发效率、避免重复开发, 开发人员常将 TPL 的源代码直接克隆到项目仓库中。Gharehyazie 等人^[6]对 Github 上代码克隆现象的分析发现, 仓库间克隆代码的比例在 10%~30% 之间。此外, Abdalkareem 等人^[7]的研究指出, 在 22 个流行的开源安卓应用中, 平均有 1.06% 的代码直接克隆自 Stack Overflow。

然而, 复用 TPL 源代码存在安全风险, 即使是广泛使用且备受认可的 TPL, 也不可避免地存在漏洞, 这些漏洞可能会被引入到使用该 TPL 的仓库中^[8]。例如, Log4j2 漏洞通过 TPL 依赖传播, 对 Java 生态中大量仓库及版本产生了影响^[9]。此外, 多数包

和依赖项的漏洞都有相应补丁, 明确软件依赖项能有效应对大部分供应链安全风险^[10]。但在 Python 生态系统中, 漏洞从产生到修复的周期较长, 这一问题随着软件供应链的发展愈发突出^[11]。因此, 开发者迫切需要借助软件成分分析工具管理 TPL 依赖, 及时检测并解决安全风险。

现有的 SCA 学术工具主要从二进制文件和源代码两个方面进行检测。在二进制 SCA 方面, 早期研究工作中, Wukong^[12]通过轻量级的静态语义特征进行粗粒度检测, 快速筛选出可疑应用, 然后对这些可疑应用进行细粒度检测; LibRadar^[13]通过分析大量应用生成第三方库的稳定代码特征, 并利用哈希技术实现高效匹配; Libd^[14]利用 Android 应用程序的内部代码依赖性来检测和分类候选库; AtvHunter^[15]通过匹配 Java 库的控制流图对安卓应用进行 TPL 检测; LibScout^[16]通过利用代码层次结构信息提取库配置文件, 使其能够在混淆的情况下检测安卓的 APK 包; LibPecker^[17]引入自适应相似性阈值来计算与目标库之间的相似性, 进一步提高检测精度。在面向 C/C++ 语言的二进制 SCA 方面, B2SFinder^[18]使用不变量将二进制文件映射到源代码来检测 TPL; ModX^[19]引入了模块化技术, 将库分成更小粒度, 以便提供更细粒度的 TPL 匹配; Binaryai^[20]使用 Transformer 模型来提取 C/C++ 二进制文件的语法特征, 并通过局部性和其他语义特征来精确匹配源函数; Haq 和 Caballero^[21]分析了 61 种二进制代码相似性判断方法, 并提供了这些方法的特征、实现和范围。

在面向 Java 语言的 TPL 检测方面, Java 仓库通常采用官方项目管理工具 Maven 进行 TPL 管理。具体而言, Maven 会生成项目的 SBOM 用于记录其依赖的 TPLs。现有面向 Java 仓库的商业 SCA 工具, 如 Eclipse steady^④、Dependency-check^⑤等均通过解析 pom.xml 等依赖文件来检测由于软件依赖而引入的漏洞。Dann 等人^[22]和 Imtiaz 等人^[23]对面向 Java 的 SCA 工具进行了准确性评估; Zhao 等人^[24]调查了影响 Java SCA 工具准确性的因素。与 Java 相似的是, Python、Ruby 和 Javascript 等编程语言也具有相应的官方项目管理工具, 如 Python 的 PyPI、Ruby 的 RubyGems 和 Javascript 的 npm。此外, 商业工具

① yaml-cpp, <https://github.com/jbeder/yaml-cpp>

② mruby, <https://github.com/mruby/mruby>

③ libgit2, <https://github.com/libgit2/libgit2>

④ Eclipse Steady, <https://github.com/eclipse/steady>

⑤ OWASP, <https://owasp.org/www-project-dependency-check/>

Ochrona^①通过解析依赖文件来分析 python 仓库依赖的 TPL。

在面向 C/C++ 仓库的 SCA 方面, 由于 C/C++ 不具有官方仓库管理工具, 因此无法通过解析 SBOM 文件识别仓库依赖的 TPLs。目前, 仅有少量的商用工具和学术研究成果提出了面向 C/C++ 语言的 SCA 方法, 即基于代码克隆检测技术^[25-27]识别软件中的复用代码, 并依此构建 C/C++ 仓库之间的依赖关系。OSSPolice^[3]使用轻量级的特征(即目录结构和文件名)作为 TPL 的签名来提高匹配效率, 一旦待测项目的目录结构或文件名称被使用者更改, 将产生大量漏报; SourcererCC^[28]是一种基于代码克隆检测的 SCA 方法, 使用基于令牌的索引作为项目特征, 提高了检测的鲁棒性, 但难以检测 TPL 之间代码克隆导致的嵌套克隆问题; CENTRIS^[4]仅保留重复函数中最早编写的函数, 以保证函数唯一性, 以此消除 TPL 间的嵌套克隆。然而, 由于需要计算仓库中全部函数与 TPL 全部函数的相似度, 导致时间复杂度高, 应用场景受限。目前最先进的 C/C++ 语言 SCA 工具 OSSFP^[5]将 TPL 的全部函数分类为重复函数、简单函数、常见函数和核心函数, 然后依次过滤前三类函数, 并将剩余函数视作 TPL 的核心函数, 最后对 TPL 核心函数和待测仓库的函数进行克隆检测, 以判断待测仓库和 TPL 之间的依赖关系。然而, 由于 OSSFP 的过滤方法缺乏对函数语义的考虑, 并未对核心函数进行精确定位, 难以保证剩余函数为核心函数, 导致检测效果不够理想。

图神经网络(Graph Neural Networks, GNN)近年来在软件工程领域展现出强大的表达能力, 已在代码漏洞检测、程序分析等任务中取得了显著成果。然而, 尽管 GNN 在通用代码分析中表现优异, 将其应用于软件成分分析任务仍面临诸多限制。一方面, 训练高质量的 GNN 模型通常依赖大规模标注数据集, 且模型训练过程本身也需大量计算资源与时间开销, 难以满足 SCA 场景下对高效性与可扩展性的要求; 另一方面, 尽管 GNN 所生成的向量特征在捕捉深层语义方面具有一定能力, 但其生成过程的可解释性较弱, 且函数指纹的稳定性和准确性仍存在争议。目前, 尚缺乏在 SCA 场景中被广泛采用且成熟稳定的 GNN 方法。

为了提供“软件供应链安全管理体系化”解决方案, 助力企业提升软件供应链安全管理能力, 国内涌现了大批 SCA 商业工具。例如, 鸿渐科技^②基于片段级算法技术及自主研发的软件成分分析数

据库, 支持 Type 1-3 克隆分析, 并实现 3 行代码的依赖匹配范围; 奇科厚德^③同时支持代码片段扫描、代码依赖关系分析和二进制代码分析等多种技术, 为用户提供项目 SBOM。国外安全公司也逐步加强了对于软件供应链的关注, 开发了一批代表性的 SCA 商业工具, 如 Black Duck^④, 以帮助客户建立软件物料清单, 提升软件供应链的安全性和可控性。

3 研究背景

3.1 函数定义

3.1.1 重复函数

重复函数指并非由项目作者自行编写, 而是从其他开源项目中复制导入的函数。以项目 pcsx2^⑤为例, 文件目录“/3rdparty/glad”下的所有函数均为重复函数。上述函数并非由项目 pcsx2 原创, 而是通过克隆项目 glad^⑥的函数导入 pcsx2。

3.1.2 简单函数

简单函数指的是代码逻辑较为简单的函数, 一般具有代码复杂度较低的特点。此类函数不包含任何关键语义, 在仓库中仅起到语义衔接的作用。图 1 展示了分别来自 function_types^⑦项目和 curl^⑧项目的两个代码逻辑简单的实例。从图中可以看出, 这类函数语义匮乏, 不具有仓库语义的代表性, 因此不符合仓库核心函数的选择标准。

```

int id() const                                CURLcode win32_init(void)
{
    return val_id;                             {
                                                }
}                                              }
(a) 简单函数示例 1                          (b) 简单函数示例 2

```

图 1 简单函数示例

3.1.3 常见函数

常见函数是指具有一定代码复杂度, 但代码逻辑和功能并不专属于某个项目的函数。这类函数通常实现通用的代码功能, 如文件读写、编码解码以及网络协议处理等。因此, 排除这类函数有助于更准确地提取仓库的核心函数。例如, 图 2 所示的函数, 它通过减少偏移量来更新 CMatchFinder 结构体

① Ochrona. <https://ochrona.dev>

② 鸿渐科技 <https://www.redrocket.cn>

③ 奇科厚德 <https://checode.cn>

④ Black Duck. <https://www.synopsys.com>

⑤ Pcsx2 <https://github.com/pcsx2/pcsx2>

⑥ glad <https://github.com/Dav1dde/glad>

⑦ function_types https://github.com/boostorg/function_types

⑧ curl <https://github.com/aseprite/curl>

```
void MatchFinder_ReduceOffsets(CMatchFinder *p, UInt32 subValue)
{
    p->posLimit -= subValue;
    p->pos -= subValue;
    p->streamPos -= subValue;
}
```

图2 常见函数示例

的位置和限制,常用于处理流数据时调整当前位置,以反映已读取或处理的数据量。该函数在 libchtr^①和 OPENCTM^②仓库中均有出现,是一个功能较为通用的函数。

3.1.4 核心函数

核心函数是项目中实现核心算法和关键代码逻辑的原创函数。以项目 minimp3^③为例,其 player/player.cpp 中的 close() 函数就是其核心函数。minimp3 是一个轻量级、快速且准确的 MP3 解码器,可用于嵌入式设备和系统实现 MP3 音频文件的解码。close() 函数用于关闭 MP3 程序,在 minimp3 项目中具有重要的代码逻辑,实现了核心功能。从图3中可以看出,该核心函数有大量调用其他函数的操作,由此可推断,此类函数在项目函数调用图中可能处于较为核心的位置。

```
static void close()
{
    nk_sdl_shutdown();
    SDL_GL_MakeCurrent(_mainwindow, _mainContext);
    for (auto it = _previews.begin(); it != _previews.end(); it++)
    {
        GLuint tex = it->second;
        glDeleteTextures(1, &tex);
    }
    SDL_GL_DeleteContext(_mainContext);
    SDL_DestroyWindow(_mainwindow);
    SDL_Quit();
}
```

图3 核心函数示例

3.2 研究动机

为了更清晰地说明现有方法 OSSFP 的局限性,并为改进方向提供思路,以一个真实的开源项目 QCodeEditor^④为例进行分析。QCodeEditor 是一个基于 Qt 框架的代码编辑器组件,常被用于具有语法高亮、自动完成等编程语言编辑功能的项目中。该项目共有 73 个函数,采用 OSSFP 文章中报告的最佳参数对其进行过滤。具体来说,依次经过 OSSFP 的重复函数、简单函数和常见函数过滤后(过滤详细情况如表1所示),剩余的 24 个函数被 OSSFP 认定为 QCodeEditor 的核心函数。

表1 OSSFP 对 QCodeEditor 进行函数过滤的详细情况

步骤	TPL 全部函数	重复函数过滤	简单函数过滤	常见函数过滤
剩余函数数量	73	56	26	24
相比前一步骤的过滤数量	—	17	30	2

如第 3.1.4 节所述,核心函数需要体现项目的核心算法和逻辑。按照这一标准,对 OSSFP 针对 QCodeEditor 检出的核心函数进行人工检查。人工分析结果显示,在 OSSFP 测出的 24 个核心函数中,有 13 个函数实现的算法逻辑相对薄弱,且不具有项目代表性语义。例如,在文件夹 example/resources/code.samples/cxx.cpp 中存在一个函数(如图4所示),该函数实现了一个简单的打印求和功能。可以看出,该函数具有一定的逻辑结构,代码复杂度较高,因此未被简单函数过滤步骤剔除;同时,它是开发者编写用于 QCodeEditor 项目测试的函数,不具有通用功能,所以也未被常见函数过滤步骤过滤掉。在这种情况下,OSSFP 将其识别为核心函数。显然,该函数与 QCodeEditor 核心语义并无关联,属于 OSSFP 错误识别的核心函数。

考虑到 OSSFP 在筛选核心函数时,只是单独分析每个函数的代码属性,然后整体进行非核心函数过滤,并未考虑 TPL 整体语义以及函数之间的关联性。为了找出真实核心函数与 OSSFP 错误识别的核心函数之间的差异,构建 TPL 的函数调用图。函数调用图通过展示功能模块、函数间的调用关系以及数据流动路径来表达程序语义,能够直观地反映每个函数与其他函数之间的关联,进而间接揭示每个函数在整个 TPL 语义结构中的地位。

```
int main()
{
    int n, sum = 0;

    std::cout << "Enter a positive integer: ";
    std::cin >> n;

    for (int i = 1; i <= n; ++i)
    {
        sum += i;
    }

    std::cout << "sum = " << sum;
    return 0;
}
```

图4 OSSFP 错误识别的 QCodeEditor 核心函数示例

图5为 QcodeEditor 项目的函数调用图,其中每个节点表示项目中的一个函数。蓝色节点表示被 OSSFP 过滤掉的函数,灰色节点表示 OSSFP 识别出的核心函数,而带红色五角星的灰色节点则为人工确认的真实核心函数。

图5显示,真实核心函数(灰色红星节点)通常位于函数调用图的中心区域,而 OSSFP 错误识别

① libchtr <https://github.com/libchtr/libchtr>

② OPENCTM <https://github.com/Danny02/OpenCTM>

③ minimp3 <https://github.com/lieff/minimp3>

④ QCodeEditor <https://github.com/zeux/QcodeEditor>

的核心函数(灰色节点)大多分布在图的边缘。为量化这一现象,本文采用中心性分析中最基础的度中心性(Degree Centrality)方法,对各节点的中心性进行评估。该方法通过计算节点的入度与出度之和,反映其在调用图中的连接程度。中心性分数越高,表明函数在图中的连接越密集,作用越关键;反之,得分较低的函数多处于边缘位置,重要性较低。

表 2 展示了真实核心函数与 OSSFP 错误识别的核心函数在度中心性上的平均分数。结果显示,真实核心函数的平均分数(0.6769)远高于 OSSFP 错误识别函数的平均分数(0.1538),与图 5 的视觉

印象一致,进一步验证了 OSSFP 所误识别的函数在调用图中具有明显的边缘化特征。因此,通过构建 TPL 的函数调用图并计算函数的中心性分数,筛选出中心性较高的函数,是一种有效引入代码语义信息的方式,能够显著降低 OSSFP 将边缘化的非核心函数误识为核心函数的风险。

表 2 度中心性分数平均值

函数类型	度中心性分数平均值
真实核心函数	0.676923
OSSFP 错误识别的核心函数	0.153846

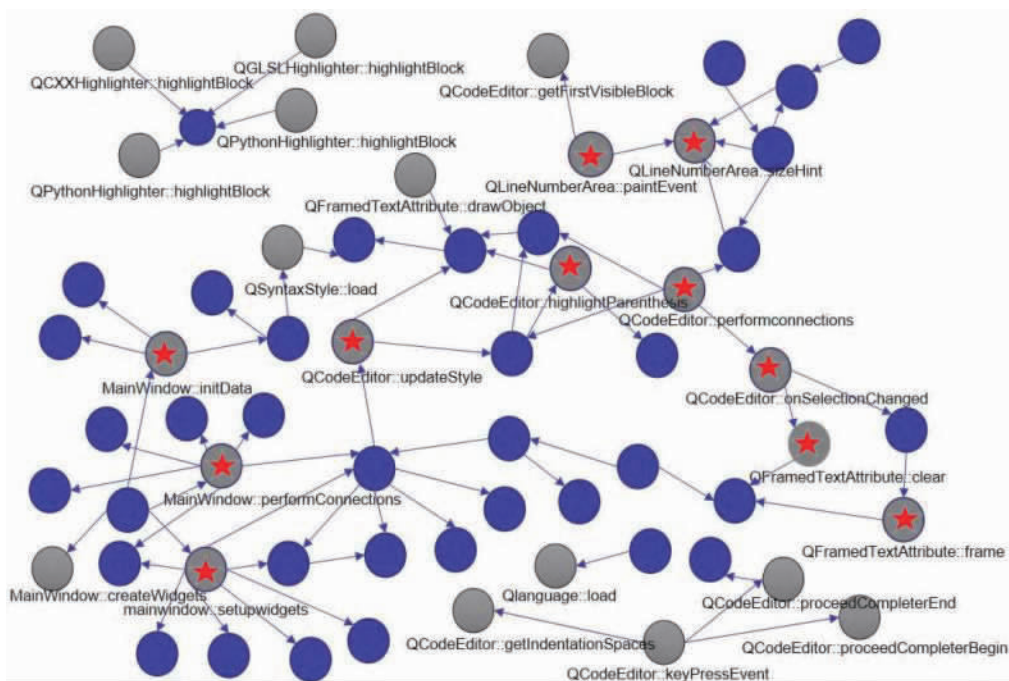


图 5 项目 QCodeEditor 的函数调用图

受上述分析的启发,本文提出一种基于中心性分析的软件成分分析方法。该方法在 OSSFP 仅通过函数代码属性进行非核心函数过滤的基础上,结合函数在调用图中的地位,充分考虑程序的整体语义,从而实现对 TPL 核心函数的精准定位。

4 方法

本文提出的基于中心性分析的软件成分分析方法(CentSCA)旨在更精准地识别 TPL 中的核心函数,从而提高 C/C++ 语言 SCA 的检测精确性。与以往方法(如 OSSFP)不同,CentSCA 不仅关注函数自身的语法特征,还引入了函数之间的结构联系,填补了传统方法在语义层面分析上的不足,如图 6 所示。

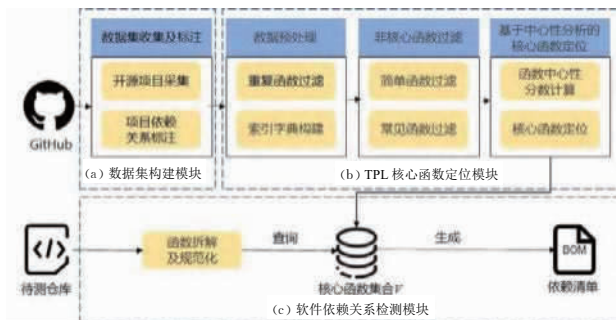


图 6 基于中心性分析的软件成分分析方法整体框架

以往工作将分析重点放在函数本身携带的静态属性上,仅基于语法层面的特征(如是否重复、是否过于简单)对函数进行过滤,试图绕开函数间的联系。例如, OSSFP 将函数依语法特征划分为不同类别并依次过滤,但未考虑这些函数在 TPL 结构中

的实际角色,忽略了其所体现的程序语义,从而导致核心函数的误判。

为此, CentSCA 引入函数调用图与中心性分析进行核心函数定位:调用图用于建模函数之间的调用关系,提供结构化的语义表达;中心性分析则对调用图中的每个函数节点进行重要性度量,量化其在整体结构中的地位与影响力,进而辅助识别真正具备核心语义的函数。

CentSCA 主要包括两个关键阶段:一是依据函数代码属性过滤掉重复函数、简单函数和常见函数;二是构建 TPL 的函数调用图,通过计算函数在图中的中心性分数,进一步筛选出真正的核心函数。通过结合函数代码属性和函数调用图的中心性分析, CentSCA 能够更全面地考虑程序的整体语义和函数间的关联性,从而提高核心函数识别的准确性。

4.1 数据集收集及标注

Roy 的研究^[29]表明, C/C++ 语言编写的开源项目,没有类似 Java 仓库的 SBOM 文件来提供项目依赖的 TPL 信息,如项目版本和 TPL 来源等。优质的 C/C++ 项目常在特定的文件目录下(例如, “/3rdparty”, “/deps” 和 “/third party” 等)引用 TPL 的源代码,或者在“Readme.md”和“Copyright”等文件中记录项目所依赖的 TPL 清单。根据上述特点, CentSCA 收集优质 C/C++ 项目并标注其所依赖的 TPLs 作为本文数据集。下面介绍数据集构造方法。

4.1.1 开源项目采集

Github 平台发布的开源项目质量良莠不齐,考虑到 Github 为项目设置的星标数(Star 数)能直观反映开源仓库的流行度,以此间接反映仓库质量。因此, CentSCA 筛选去除 Github 中星标数量排名前 100 的开源仓库中的无效项目(如只有头文件的库),最终保留 56 个有效开源仓库及其依赖的 TPLs 作为采集目标。最终采集 TPLs 中 C 语言项目 145 个, C++ 语言项目 365 个,共计 510 个。

对于项目的采集过程, CentSCA 通过 git clone 命令将目标项目下载到本地,并保留项目的创建信息和版本信息。为了便于管理,爬取过程只关注发布 tag 的项目版本,其他分支版本不再考虑。需要说明的是,如果项目没有发布 tag 版本,则只获取其 master 分支代码。

4.1.2 项目依赖关系标注

针对采集的目标仓库,采用人工分析的方式对其依赖的 TPLs 进行标注。根据开发者的编程习惯,优质 C/C++ 项目的 TPL 依赖项声明常见于三处,分别是:以文本方式记录在 Readme 和 Copyright 文件;以版权声明的方式记录在 commit 和函数头部的注释中;以文件夹的方式存放在 “/3rdparty”, “/deps” 和 “/third party” 等第三方文件夹目录下。

具体的项目依赖关系标注流程如下:首先,针对仓库 Readme 和 Copyright 文件,记录其文本信息中涉及的依赖 TPL 名称和版本;其次,针对 commit 以及函数头部注释,记录其版权信息中涉及的 TPL 名称和版本;然后,在仓库中寻找第三方文件夹(“/3rdparty”, “/deps” 和 “/third party” 等),根据文件夹内项目名称,记录其依赖的 TPLs:如果以链接引用的形式存放,则查看引用版本的签名提交时间,并查取最接近的 tag 版本记录;如果是文件的形式引用,则结合上述两处获取到的版本信息进行补充;最后,在 Github 搜索引擎中搜索 TPL 名称,验证是否存在对应版本的 TPL。如果存在,则将 TPL 名称和版本以二元组的形式储存。为了保证仓库依赖关系标注的准确性,五位专家进行交叉检验,最终保留数据集中五名专家均认可的依赖关系条目。具体而言,每位专家查看 TPL 的函数是否被真实地复用在项目中,检查标准是判断项目与 TPL 是否存在相同或相似的函数。一旦存在,则认为此 TPL 真实被目标项目依赖,并记录 TPL 函数被目标仓库使用的具体位置以及 TPL 版本。最终,构成 CentSCA 的基准真相(Ground Truth, GT),并按照表 3 所示的方式存储。

表 3 CentSCA 部分数据集详情

项目地址	项目版本	TPL 被引入的具体位置	TPL 地址	TPL 版本
https://github.com/fogleman/Craft	master	/deps/glfw/deps/tinythread.c	https://github.com/tinythread/tinythread	v1.1
https://github.com/godotengine/godot	master	/thirdparty/misc/fastlz	https://github.com/ariya/FastLZ	0.5.0
https://github.com/winlinvip/srs	master	/trunk/3rdparty/st-srs	https://github.com/ossrs/state-threads	v1.9.1
https://github.com/arangodb/arangodb	v3.11.3	/3rdParty/zlib/zlib-1.2.13	https://github.com/madler/zlib	v1.2.13

4.2 数据预处理

为了支撑后续 TPL 检测, 本节将仓库源代码拆分为函数粒度, 并对函数进行代码规范化后去除 TPL 库中的重复函数, 然后对函数源码及函数元数据进行结构化梳理以生成索引字典。

4.2.1 重复函数过滤

代码中存在大量与代码语义无关的字符, 如注释、变量名和空白符等, 为了减少上述信息干扰, 本节对于源代码进行规范化处理。此外, 提取函数的创建时间和唯一性标识作为函数的元数据, 并按照函数编写时间, 保留相同函数中的原创函数。

复用 TPL 源代码不可避免地会根据项目自身需求对 TPL 代码进行适应性微调。例如, 调整 TPL 代码中的变量名和函数名以适应本项目的语义环境。因此, 为了避免变量名等语义无关信息对于 TPL 检测效果的影响, CentSCA 对提取的函数进行以下两个方面的代码规范化处理: (1) 去除与代码功能无关的内容, 如代码中的所有注释信息以及多余空白字符; (2) 将函数名和变量名映射为统一名称, 如 void FUNC1, int VAR1。

对于 SCA 任务而言, 源函数具有两个重要信息, 分别是函数创建时间和代码唯一性。

针对创建时间, 相同(或相似)的函数需要按照编写时间的先后顺序来判断依赖关系, 最先创建编写的函数即为原创函数, 而较晚编写的函数可以被视为参考了较早编写的相同(或相似)函数。为了保证公平性, CentSCA 采用函数所属文件的最后一次修改的时间戳信息, 来表征函数创建时间。鉴于数据集面向 Github 中的开源项目, 使用 Git 指令查询代码提交日志来确定文件最后一次修改的时间戳信息, 并将其作为函数创建的时间戳。

针对代码唯一性信息, CentSCA 利用信息摘要算法(Message-Digest Algorithm, MD5) 计算规范化后函数代码文本的哈希值, 以此值作为源函数的唯一性标识, 从而快速检索数据库内相同的函数文本, 并实现对于重复函数的去重。

4.2.2 索引字典构建

为支持后续的高效函数查询与重复函数识别, 本小节构建一个索引字典(Index Dictionary), 如图 7 所示。该字典以函数的源代码哈希值作为键, 以其对应的函数元信息作为值, 元信息包括: 函数所在的仓库名、版本号、函数名、出现在不同项目中的次数, 以及最早出现的时间戳等。



图 7 数据库哈希索引结构

索引字典的构建过程如下:

(1) 函数元信息提取与映射构建: 遍历每个项目的所有版本, 提取其中所有函数, 并计算其源代码哈希值, 然后将哈希值作为键, 函数元信息作为值, 构成候选键值对;

(2) 索引插入与更新: 在处理每一个函数时, 首先检查其哈希值是否已经存在于索引字典中: 如果尚未出现, 则将其添加到字典中, 并初始化其出现次数为 1, 同时记录其创建时间作为当前最早出现时间; 如果已存在, 则从索引中取出已有记录, 与当前函数的创建时间进行比较。若当前函数的创建时间更早, 则用当前函数的信息更新索引; 否则, 保持原有记录不变。同时, 无论是否更新创建时间, 都会将该函数的出现次数加一。需要注意的是, CentSCA 采用“最早出现”原则来识别原始函数, 即将首次出现在开源生态中的函数视为被复用对象, 后续出现的相同哈希值函数视为重复函数。

在应用该字典识别重复函数时, 若某个 TPL 中的函数哈希值在索引字典中已存在, 且对应的元信息并不来自当前 TPL, 则将其判定为重复函数, 予以过滤。此外, 为防止某个函数在同一 TPL 内部由于版本演化等原因多次出现造成统计偏差, 在处理函数出现次数时, 仅保留当前 TPL 内创建时间最早的实例进入索引。这样可以确保函数的出现次数准确反映其在不同 TPL 项目中的分布频次, 而非在单一项目内的重复统计。该策略也为后续常见函数的识别与过滤提供了数据支撑。

4.3 非核心函数过滤

为了精准地定位具有仓库代表性语义的函数, 本小节旨在过滤仓库中明显的非核心函数, 以减少其对于仓库核心函数定位的干扰。经分析, 明显的非核心函数有两种类型, 分别是代码逻辑和语义过于简单的函数(即简单函数)和实现通用功能的函数(即常见函数)。

4.3.1 简单函数过滤

本步对简单函数进行过滤, 减少简单函数对于

仓库核心函数定位的影响。由于简单函数具有代码复杂度低的特点, CentSCA 选择被广泛使用的代码复杂度指标—— MI (Maintainability Index, MI)^[30] 作为衡量标准。 MI 可以通过式(1)计算, 其中 HV 代表 Halstead 体积, CC 代表圈复杂度, LOC 代表代码行数。

$$MI = 171 - 5.2 \times \ln(HV) - 0.23 \times (CC) - 16.2 \times \ln(LOC) \quad (1)$$

其中, 由于函数经过规范化已去除空行, 因此 LOC 可以通过正则匹配换行符获得。 CC 是评估程序稳定性和可信度的度量方法, CC 较低的函数潜在地具有较简单的逻辑。 CC 可以通过式(2)计算。其中 P 表示控制流图中的条件节点数, 为了保证效率, CentSCA 通过计算条件分支的语句数量来代替控制流图中的条件节点数。针对 HV , 计算公式如公式(3)所示。其中, N_1 和 N_2 分别是操作符和操作数的总数; n_1 和 n_2 分别是操作符和操作数的数量, 均通过正则匹配获得。需要说明的是, 在源代码分析中, 变量、常数、数字和字符串被视为操作数, 而其他类型的词法标记均被视为操作符。

$$CC = P + 1 \quad (2)$$

$$HV = (N_1 + N_2) \times \log_2(n_1 + n_2) \quad (3)$$

根据 MI 的计算公式可以推断, 逻辑较为简单的函数会得到较高 MI 的分数, CentSCA 通过对 MI 设定阈值 θ_0 来过滤逻辑较为简单的函数。OSSFP 实验发现为 θ_0 的取值为 0.5 时, 能够取得最佳检测性能, 因此 CentSCA 参考 OSSFP 的结论, 将简单函数过滤比例 θ_0 的最佳取值选为 0.5。即将 TPL 库中 MI 分数排名靠前的 50% 函数视作简单函数, 并将其过滤, 剩余函数保留至下一阶段(常见函数过滤)。

4.3.2 常见函数过滤

本步对第 3.1.3 节中提到的常见函数进行过滤。尽管上一步骤已经过滤掉项目中的简单函数, 但仍存在代码复杂度较高, 而代码逻辑通用且常见的函数, 难以通过上一步骤实现过滤。

经调研发现, 常见函数具有在 TPL 库中出现频率远超其他函数的特点, 如 `zstd`^① 仓库中 `lib/common/xxhash.c` 的函数 `XXH32_digest_endian()` 用于计算 32 位的哈希值。经统计, 该函数在 CentSCA 数据集中的 5 个不同项目中反复出现, 说明其是一个实现通用功能的函数。该函数的功能并不能归属于某个项目特有, 不具有一个项目代表性的语义。函数内包含 26 行代码, MI 分数为 77.96, 而如图 3

所示的仅有 3 行代码的简单函数, MI 分数为 146.791469。在经过简单函数过滤之后, 该函数仍然保留在函数库中, 说明简单函数过滤步骤难以有效过滤常见函数。

为了过滤此类代码复杂度高且经常出现的函数, CentSCA 引入了文本挖掘和信息检索中广泛使用的词频概念^[31], 词频反映了一个词在单个文档中的出现频率。借鉴该思想, CentSCA 将函数视为词语, 整个数据集视为文档, 通过词频数识别衡量频繁出现的函数(即, 常见函数)。如上文所述的 `XXH32_digest_endian()` 函数, 在数据集的不同 TPL 中出现过 5 次, 因此其词频为 5。而在 TPL 仓库中仅出现过一次的函数, 词频为 1。可以看出, 词频能够有效区分数据集中的常见函数和非常见函数, 满足 CentSCA 对于常见函数过滤的需求。

CentSCA 采用阈值 θ_1 对常见函数进行过滤, 当函数的词频数大于 θ_1 时, 则被认定为常见函数并被过滤。为了选择 θ_1 的最佳取值, 实验预设 θ_1 的取值范围为 $[2, 4, 6, \dots, 20]$, 步长为 2。例如, 当 θ_1 为 2 时, CentSCA 将词频数(在数据集中出现次数)大于 2 的函数视为常见函数, 并将其过滤; 反之, 将词频数小于等于 2 的函数保留。

4.4 基于中心性分析的核心函数定位

基于代码克隆的 SCA 通过定位 TPL 的核心函数进行检测, 具备双重优势: (1) 聚焦关键函数能够显著减少相似性比较次数, 提升大规模分析效率; (2) 规避非核心函数依赖干扰, 降低误报风险。为弥补现有方法在非核心函数过滤中未充分考虑语义特征的问题, 本文结合函数调用图的语义建模能力和中心性分析对关键节点的识别能力, 实现对具备核心语义代表性的函数的有效筛选。

4.4.1 中心性分数定位

核心函数是仓库中表征其核心代码语义的函数, 经过人工对多个仓库源码的剖析, 发现核心函数具有以下三类结构性特征:

(1) 高函数影响力: 核心函数实现项目的关键功能, 通常被多个其他函数调用, 或通过调用整合其他函数的能力, 对整体功能实现具有关键影响;

(2) 高控制流和数据流流动性: 核心函数往往是不同功能模块或场景共享的关键语义节点, 处理的数据与控制逻辑贯穿多个路径, 是信息流通的中枢;

① `zstd` <https://github.com/facebook/zstd>

(3) 关键结构作用: 在函数调用链中, 核心函数处于连接多个子模块的枢纽位置, 是仓库核心功能实现流程中的结构性保障。

与此同时, 中心性分析方法最初用于衡量网络中节点的重要性, 广泛应用于社交网络、交通网络、信息传播等领域^[32-33]。该类方法基于邻接关系度量节点的“中心程度”, 从而识别在网络中具有关键影响的节点^[34]。具有高中心性的节点通常具备以下三方面属性:

(1) 高影响力: 对其他节点具有较强的依赖性, 一旦变化将对整个网络产生显著影响;

(2) 高信息流动性: 作为网络中信息传递的核心通道, 在流通效率上占据关键地位;

(3) 高结构稳定性: 维系网络连通性, 是保持系统稳定与完整的关键节点。

根据上述分析, 核心函数在调用图中的结构性特征与高中心性节点的网络属性高度一致。因此, 本文在函数调用图的基础上引入中心性分析方法, 通过计算函数节点的中心性分数, 筛选出在结构上处于核心位置、语义上发挥关键作用的函数, 从而实现核心函数的精准识别。具体而言, CentSCA 首先提取 TPL 的函数调用图, 随后采用中心性分析方法计算每个函数的中心性分数, 并据此筛选出中心性较高的函数作为候选核心函数。

下面详细介绍 CentSCA 使用到的四种中心性分析方法。

度中心性(Degree Centrality)采用最大可能度数对每个节点度数进行归一化, 以最直观的方式展示出一个节点(函数)在网络(函数调用图)中的重要性, 计算方法如式(4)所示, 其中 N 表示总节点数, 最大可能度数为 $N-1$, $\text{deg}(v)$ 表示节点 v 的度数。

$$C_d = \frac{\text{deg}(v)}{N-1} \quad (4)$$

介数中心性(Betweenness Centrality)计算经过一个节点的最短路径数量来确定。经过该节点的最短路径越多, 说明该节点(函数)在网络(函数调用图)中越为重要, 表明该函数(节点)在整个网络图中具有重要桥梁作用。介数中心性的计算方法如式(5)所示, 其中 δ_{st} 为节点 s 到节点 t 最短的路径数量, $\delta_{st}(v)$ 为节点 s 到节点 t 经过 v 节点的最短路径数量。

$$C_b(v) = \sum_{s \neq v \neq t} \frac{\delta_{st}(v)}{\delta_{st}} \quad (5)$$

紧密中心性(Closeness Centrality)计算节点与其他所有点的距离和的倒数, 函数距离和越小说明该节点(函数)距离各个节点(函数)越近, 可以近似看作其所处于网络(函数调用图)的中心位置。计算方法如式(6)所示, 其中 $\text{dis}(i, j)$ 为节点 i 到节点 j 的距离, G 为所有节点数。

$$C_c(i) = \frac{1}{\sum_{j \in G} \text{dis}(i, j)} \quad (6)$$

调和中心性(harmonic Centrality)的计算方式和紧密中心性的计算方式类似, 通过非连通图的紧密性获得。然而, 紧密性中心性只能用于连接图的中心性计算, 不适用于非连通图。调和中心性先取倒数再相加的计算方式, 可以将距离取到无穷, 因此适用于非连通图的中心性计算, 如式(7)所示。

$$C_h(i) = \frac{\sum_{j \in G} \frac{1}{\text{dis}(i, j)}}{N-1} \quad (7)$$

针对 TPL 的函数调用图, CentSCA 分别采用上述 4 种中心性分析方法计算每一个节点(函数)的中心性分数。根据式(4)~(7), 各个中心性的分数越高, 说明该函数在仓库中的语义贡献度越高, 即该函数是仓库核心函数的可能性越高。

CentSCA 采用阈值 δ 对中心性分数进行过滤, 同时由于中心性分析比例 δ 和中心性分析方法类型息息相关, 分别对四种中心性分析方法(介数中心性、调和中心性、度中心性和紧密中心性)在比例 δ 下进行实验, 处理对象为 TPL 的全部函数。具体而言, 分别测试不同中心性分析方法在 δ 预设比例为 [10%, 30%, 50%, 70%, 90%] 时, CentSCA 的检测表现。例如, 当中心性分析方法为度中心性, δ 取值为 30% 时, 则将 TPL 中度中心性分数较高的 30% 的函数视为 TPL 的候选核心函数集合。

4.4.2 核心函数定位

使用中心性分数定位核心函数和通过过滤非核心函数定位核心函数有两种结合方法, 下面对两种方式的优劣进行分析:

(1) 在非核心函数过滤基础上再使用中心性分数进行核心函数定位。由于先过滤掉非核心函数会破坏函数调用图的完整性, 使得函数调用图成为非连通图, 在非连通调用图中计算得到的函数中心性分数无法代表函数在项目实际运行的情况, 因此影响核心函数定位的精确性。

(2) 同时进行非核心函数过滤和中心性分数定位核心函数, 再将非核心函数过滤结果和中心性分数定位结果取交集, 该交集即为 TPL 最终的核心函数。此方法结合了非核心函数过滤和核心函数定位的优势, 且彼此的筛选过程互不干扰, 以此保证定位结果的完整性和有效性。

4.5 软件依赖关系检测

将中心性分数定位得到的候选函数集合, 与非核心函数过滤中得到的集合取交集, 得到最终核心函数结果, 进而用于软件依赖关系检测。

具体而言, 针对目标 TPL, 定义经过非核心函数步骤的过滤后, 剩余函数集合为 V_1 ; 中心性分数定位核心函数步骤筛选出的候选核心函数集合为 V_2 , 二者取交集作为 TPL 最终的核心函数集合 V 如式(8)所示。

$$V = V_1 \cap V_2 \quad (8)$$

CentSCA 把核心函数集合内的函数视作 TPL

的指纹函数, 它们具有仓库代表性的代码语义。因此, 一旦有仓库引用核心函数, 说明仓库依赖于该 TPL。此外, 为了提高 SCA 分析的效率, CentSCA 存储 TPL 核心函数的哈希值。

在获得 TPL 的核心函数集合 V 后, CentSCA 对待测仓库进行软件成分分析, 如图 8 所示。本部分的输入是待测项目和 TPL 的核心函数集合 V , 输出是该项目依赖的第三方组件清单, 即 SBOM 文件。

软件依赖关系检测部分可以分为三个步骤: 首先, 将仓库源代码按照函数粒度进行拆解和代码规范化(同第 4.2 节所述), 仓库被拆分为图 8 所示的 n 个函数; 其次, 计算每个规范化函数的文本哈希值, 并将每个函数哈希值与 TPL 的核心函数集合 V 进行匹配。如果匹配到相同哈希值, 则说明待测仓库引用了该 TPL 的核心函数, 即依赖于该 TPL; 最后, 列出与待测仓库具有相同哈希值的 TPL 清单, 以此获得待测仓库的第三方组件清单。

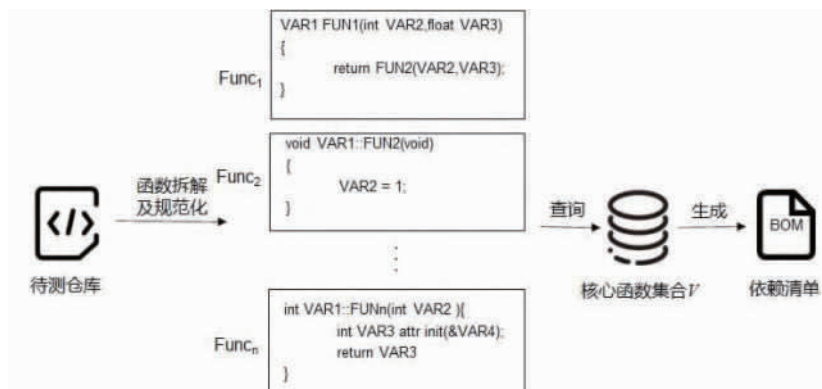


图 8 软件依赖关系构建流程

5 实验分析

本章对实验环节进行详细说明。首先, 阐述实验准备工作, 涵盖实验数据集、实验实施方法和实验评估指标。其次, 提出四个研究问题以验证本方法的有效性。研究问题 1 (Research Question1, RQ1) 通过参数实验为 CentSCA 选取最佳的参数组合; RQ2 通过消融实验探讨在定位项目核心函数过程中, 每个子步骤(包括非核心函数过滤和核心函数定位)的必要性; RQ3 通过对比实验评估 CentSCA 的准确性, 实验将 CentSCA 与最先进的两个面向 C/C++ 项目的 SCA 方法(OSSFP 和 CENTRIS)进行对比, 评估其检测效果, 并深入分析实验结果产生假正例与假负例的原因。RQ4 通过案例研究, 进

一步验证核心函数识别在实际下游任务中的价值, 实验选取一个真实开源仓库, 分别使用 CentSCA 与 OSSFP 进行 SCA 分析, 结合漏洞数据库 Snyk 的检测结果进行对比, 评估精准识别 TPL 核心函数对提升供应链漏洞检测准确性的实际效果。具体研究问题如下:

(1)RQ1: CentSCA 的最佳参数怎样选择?

(2)RQ2: 每个非核心函数过滤(及核心函数定位)步骤对 CentSCA 的检测性能有何贡献?

(3)RQ3: 与最先进 SCA 方法相比, CentSCA 检测软件依赖的效果如何?

(4)RQ4: CentSCA 在真实项目中的应用是否能够有效提高供应链漏洞检测的准确性?

5.1 实验准备

第 5.1 节从实验数据集、具体实施和评估指标三方面介绍实验准备。

5.1.1 实验数据集

本文通过构建的基准真相(Ground Truth, GT)数据集进行实验, 该数据集由五位专家进行交叉检验, 以保证数据集真实有效。如表 4 所示, 数据集最终覆盖了 510 个项目, 项目间共有 1016 条软件依赖关系。其中, 依赖 TPL 最多的项目共依赖 146 个 TPLs, 最少的项目依赖于 1 个 TPL, 项目平均有 11.55 项依赖, 项目依赖 TPLs 的中位数为 6 个。

表 4 实验数据集的详细信息

总项 目数	依赖关系 总数	项目依赖 数的最大值	项目依赖数 的最小值	项目依赖数量 的平均数	项目依赖数 的中位数
510	1016	146	1	11.55	6

5.1.2 具体实施

实验设备为内存大小 128GB, 配有 8 核英特尔 Xeon 处理器的标准服务器。实验环境采用 Ubuntu20.04 的操作系统, 系统各个阶段采用 hashlib^①、Networkx^②、Doxxygen^③和 Pydot^④等工具实现。具体而言, 在函数数据预处理阶段, 利用 re 进行正则表达式匹配将文件拆解为函数粒度, 并对函数进行代码规范化; 在索引构建阶段, 使用 Hashlib 为规范化后的函数生成文本哈希; 在核心函数定位阶段, 首先使用 Doxygen 为每个项目以 HTML 文档的形式生成函数调用图, 并将函数作为节点转为 Dot 格式, 再使用 Pydot 读取 Dot 文件, 最后使用 Networkx 计算各函数的中心性分数。

5.1.3 评估指标

本文采用常见的准确性评估指标对软件成分分析方法进行评估:

(1) 真正例(True Positive, TP): 正确识别为第三方依赖的依赖关系;

(2) 假正例(False Positive, FP): 错误识别为第三方依赖的依赖关系;

(3) 假负例(False Negative, FN): 未能成功识别的第三方依赖的依赖关系;

$$(4) \text{召回率(Recall)} = TP / (TP + FN)$$

$$(5) \text{准确率(Precision)} = TP / (TP + FP)$$

$$(6) F1 = 2 * \text{Precision} * \text{Recall} / (\text{Precision} + \text{Recall})$$

当检测到的依赖关系在 GT 数据集中存在时, 则认为工具正确识别了待测软件所依赖的 TPL, 记录为 TP; 反之, 则认为错误识别, 记录为 FP; GT 中存在的未能够检测到的依赖关系记录为 FN。识别结果的准确率为正确识别的依赖关系数量与检出依赖关系数量的比值。

5.2 RQ1: CentSCA 的最佳参数怎样选择?

5.2.1 设置

CentSCA 包含四个主要参数, 分别是简单函数过滤比例 θ_0 、常见函数过滤阈值 θ_1 、中心性分析比例 δ 和中心性分析方法的类型。其中 θ_0 如第 4.3.2 节所说明取得 0.5; 对于其他参数, 本节将通过一系列对照实验来为其选择最佳取值。

针对常见函数过滤阈值 θ_1 , 其过滤对象是经过简单函数过滤后保留的函数集合, 当函数词频数大于 θ_1 时, 则被认定为常见函数并进一步过滤。为了选择 θ_1 的最佳取值, 实验预设 θ_1 的取值范围为 [2, 4, 6, ..., 20], 步长为 2。例如, 当 θ_1 为 2 时, CentSCA 将词频数(在数据集中出现次数)大于 2 的函数视为常见函数, 并将其过滤。需要注意的是, TPL 的全部函数经过上述两个过滤步骤后, 保留的函数集合即为 V_1 , 如第 4.4.2 节所述。

由于中心性分析比例 δ 和中心性分析方法类型息息相关, 分别对四种中心性分析方法(介数中心性、调和中心性、度中心性和紧密中心性)在比例 δ 下进行实验, 处理对象为 TPL 的全部函数。具体而言, 分别测试不同中心性分析方法在 δ 预设比例为 [10%, 30%, 50%, 70%, 90%] 时, CentSCA 的检测表现。例如, 当中心性分析方法为度中心性, δ 取值为 30% 时, 则将 TPL 度中心性分数较高 30% 的函数视为 TPL 候选核心函数集合(即 V_2)。最终, TPL 的核心函数集合为 V (即 $V_1 \cap V_2$), 并对 V 生成哈希库, CentSCA 将待测项目的函数哈希值依次与 TPL 核心函数哈希库匹配, 以进行软件依赖检测。

5.2.2 结果

θ_1 的取值选取。图 9 展示了 θ_1 在不同取值时对 CentSCA 检测效果的影响。具体而言, 先将 θ_0 取值为 0.5 进行简单函数过滤; 然后, 针对剩余函数, 在

① hashlib <https://docs.python.org/3/library/hashlib.html>

② Networkx <https://networkx.org/>

③ Doxygen <https://www.doxygen.nl>

④ Pydot <https://github.com/pydot/pydot>

不同 θ_1 下进行常见函数过滤；最后，将两步的剩余函数视为 TPL 的核心函数集合 V ，并对其执行软件依赖分析。

从图 9 的结果可以看出，随着 θ_1 降低，CentSCA 的准确率持续上升，召回率基本保持平稳后有略微下降。当 θ_1 大于 8 时，CentSCA 的召回率保持稳定，而精确率略有上升（不超过 5%）。其原因在于在 CentSCA 构建的数据集中，出现次数大于 8 次的函数数量极少，因此对于常见函数的过滤效果并不明显，导致精确率上升缓慢。当 θ_1 小于等于 8 且大于等于 2 时，CentSCA 准确率显著提升，召回率有略微降低（不超过 2%）。可以推断，数据集中出现 2~8 次的函数的数量较多，因此常见函数过滤效果明显。值得注意的是，当 θ_1 为 2 时，虽然 CentSCA 的准确率最高，但召回率下降明显。原因在于数据集中出现 2 次的函数数量较多，其可能在不同 TPL 中都提供了较为关键的语义，因此将出现 2 次的函数视为常见函数并将其过滤，很大可能导致误报。根据上述分析，CentSCA 将常见函数过滤阈值 θ_1 的最佳取值选定为 4，即当函数词频大于 θ_1 时，则认定为常见函数并将其过滤。

中心性分析方法和比例 δ 的选取。下面为 CentSCA 选择最佳的中心性分析方法及对应的中心性分析比例 δ 。具体而言，先将 θ_0 取值为 0.5，进行简单函数过滤；然后，将 θ_1 设置为 4，进行常见函数过滤，过滤后获得剩余函数集合 V_1 ；接着，分别采用不同的中心性分析方法和不同比例 δ 执行核心函数定位，得到候选核心函数集合 V_2 ；最后将 $V_1 \cap V_2$ 视为 TPL 的核心函数集合 V ，并对其执行

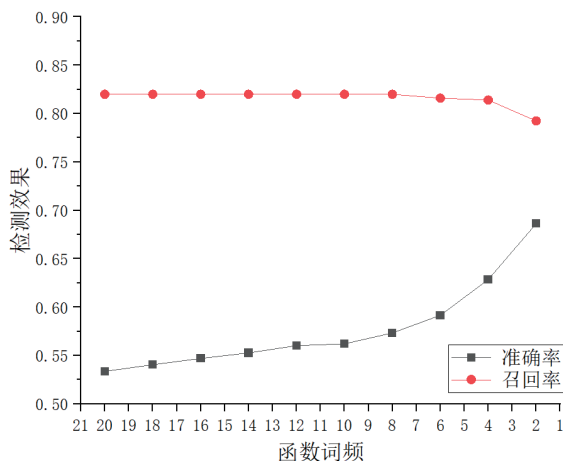


图 9 常见函数过滤阈值 θ_1 对 CentSCA 检测效果的影响

软件依赖分析。

由图 10 可知，CentSCA 在使用紧密中心性分析和调和中心性分析方法时表现不佳。具体而言，当 δ 为 10% 和 30%，使用紧密中心性和调和中心性的 CentSCA 的召回率都仅有 0.65 以下。其背后的原因在于，上述两种中心性分析方法都是通过度量一个节点到图中所有其他节点的平均最短路径长度来衡量节点重要性（区别在于调和中心性分析考虑了非连通图的情况）。其认为中心性高的函数，通常可以更快地与其他所有函数建立联系。换句话说，它们倾向于反映一个函数在函数调用图中作为“信息传递中心”的潜力。然而，在核心函数筛选任务中，核心函数并非一定会起到关键的信息传递作用，因此可能会错误定位出仅负责上下文语义衔接（不具有具体代码语义）的函数。总而言之，紧密中心性和调和中心性分析方法与核心函数定位的场景需求不符，导致核心函数定位不准确。

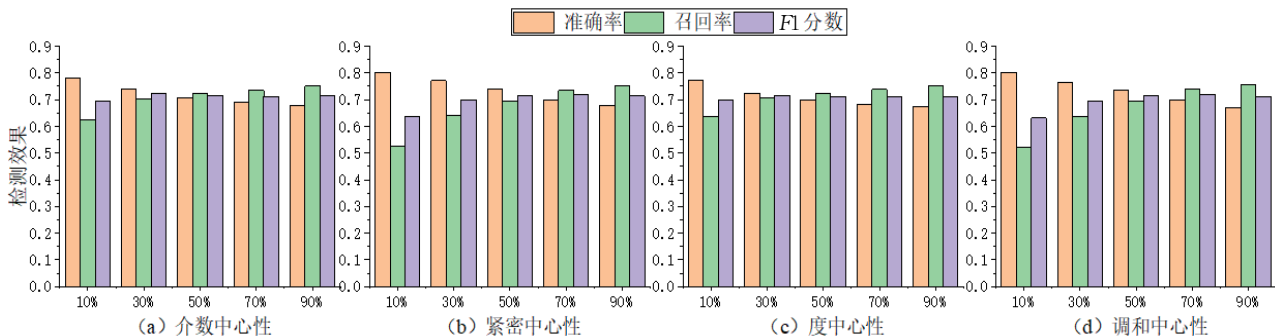


图 10 中心性分析方法及其比例 δ 对 CentSCA 检测效果的影响

与此同时，从图 10 可以看出度中心性和介数中心性分析方法表现较好。针对度中心性，中心性高的节点可以被看作是“本地中心”，在其直接连接的邻居中占据重要地位。在函数调用图中，如果一个

函数被频繁调用或多次调用其他函数，说明其可能实现了项目中的核心功能或关键业务逻辑。例如，处理数据、计算结果、执行重要的算法或实现了主要的应用逻辑，因此被其他模块或组件频繁调用，

以便在不同场景下复用其核心功能。针对介数中心性,介数中心性高的节点往往是其他节点之间信息传播的必经之路,对信息流具有很大的控制力和影响力,因为移除这些节点会显著增加网络中节点间的路径长度。换句话说,其能够识别网络中起到关键“控制”作用的节点,与核心函数在函数调用图中的作用相符。随着中心性分析比例 δ 增长,使用介数中心性和度中心性的 CentSCA,检测效果基本呈同样的变化趋势。综合考虑准确率和召回率,选择两者中 F1 得分略高的介数中心性分数作为 CentSCA 的中心性分析方法,并根据最高的 F1 分数,选择其对应的中心性分析比例 δ 为 30%。

回答 RQ1: 本节通过实验为 CentSCA 选择最佳参数。首先,本节通过分析常见函数过滤阈值 θ_1 对于 CentSCA 的准确率与召回率的影响,确定将出现次数大于 4 次的函数视为常见函数并过滤时, CentSCA 的检测效果最佳;随后,在 θ_1 为 4 条件下,分析四种中心性分析方法及其比例 δ 对于 CentSCA 检测效果的影响,并简要分析了四种中心性分析方法的特点,最终确定使用介数中心性,并将介数中心性分数最高的 30% 函数定为 TPL、候选核心函数。

综上所述,在 θ_0 为 0.5、 θ_1 为 4、 δ 为 30%且采用介数中心性的情况下, CentSCA 取得最佳性能,其准确率为 0.74,召回率为 0.70, F1 分数为 0.72。

5.3 RQ2: 每个非核心函数过滤(及中心性分数定位核心函数)步骤对 CentSCA 的检测性能有何贡献?

5.3.1 设置

本节通过消融实验来评估每个非核心函数过滤步骤及核心函数定位步骤对 CentSCA 性能的影响。具体而言, CentSCA 依次过滤重复函数、简单函数和常用函数,然后结合中心性分析定位核心函数的结果,生成 TPL 的最终核心函数集合 V 。为了评估每个步骤的有效性,实验首先对 TPL 中的全部函数进行软件依赖分析。然后,在此基础上依次叠加重复函数过滤、简单函数过滤、常用函数过滤和中心性分析定位核心函数。需要说明的是,实验参数选择均遵循第 5.2 节的最佳参数,即简单函数过滤比例 θ_0 设置为 0.5,常见函数过滤阈值 θ_1 为 4,采用介数中心性分析方法,且中心性分析比例 δ 为 30%。

5.3.2 结果

图 11 展示了采用不同非核心函数过滤和核心函数定位步骤时, CentSCA 的检测效果变化。其中,纵轴表示检测效果,横轴依次表示在前一步骤的基础上叠加的执行步骤。可以看出,随着每个步骤的叠加应用, CentSCA 的准确率提升明显,说明每个步骤都为核心函数集合 V 的筛选做出了显著贡献,即每个步骤均有助于筛选出准确的核心函数。

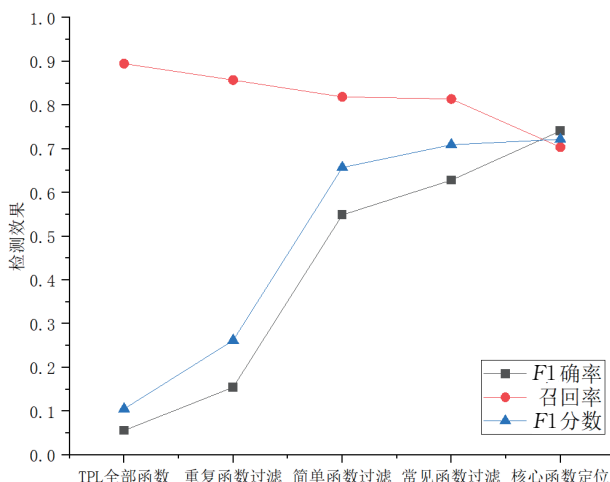


图 11 叠加过滤及定位步骤对 CentSCA 检测效果的影响

首先,分析直接针对 TPL 全部函数进行软件依赖分析的情况。如表 5 所示,本步骤面向数据集的全部函数,函数数量超过 2662 万。具体而言,直接将 TPL 中的全部函数视作 CentSCA 的核心函数集合 V ,然后针对核心函数集合 V 执行软件依赖分析。由于缺乏对函数的筛选,受嵌套 TPL 和其他无关紧要的非核心函数影响,此时的 CentSCA 产生了大量错误且混乱的依赖分析结果。例如,待测项目 A 和 TPL_C都依赖于 TPL_B的同一函数,对 A 进行软件依赖分析就得到 A 同时依赖于 TPL_B和 TPL_C的错误关系,产生假正例,因此此时准确率低于 0.1,如图 11 所示。

表 5 各步骤核心函数 V 集合中的函数数量

步骤	TPL 全部函数	重复函数过滤	简单函数过滤	常见函数过滤	核心函数定位
函数数量	26 620 937	799 668	407 018	406 829	97 054
过滤数量	-	258 821 269	392 650	189	309 775

其次,分析对 TPL 执行重复函数过滤后,进行软件依赖分析的情况。具体而言,仅保留相同函数中编写时间最早的一个,并将这些唯一函数构建

CentSCA 的核心函数集合 V , 当前核心函数集合 V 中的函数数量接近 80 万个(如表 5 所示)。在对重复函数进行过滤后, 每个函数只会归属于一个 TPL, 使得 TPL 之间克隆导致的误报数量大量减少, 因此准确率有所上升。然而, 由于核心函数集合 V 中仍然存在简单函数和常见函数, 因此当项目复用了 TP 中的此类函数时, 也会被认定依赖于该 TPL。例如, 即使项目存在一个函数与 TPL 中函数体内仅包含一行“RETURN VAR1”的函数相同, 也会被测出项目依赖于该 TPL。上述问题导致该阶段的检测准确率仍处于较低水平(不超过 0.15), 如图 11 所示。

然后, 分析简单函数过滤对软件依赖分析的影响。在重复函数过滤后, 过滤 MI 值大小前 50% 函数, 将剩余 40 万余个函数作为核心函数集合 V 。由于简单函数普遍存在, 且执行代码规范化后简单函数重复概率大大提升, 叠加简单函数过滤使 CentSCA 准确率提升近 40%(图 11), 这表明简单函数是影响 SCA 准确性的关键因素。然而, 由于常见函数与其他非核心语义代表的函数存在, 检测效果仍不够理想。

之后, 分析常见函数过滤对软件依赖分析的影响。在重复/简单函数过滤基础上, 过滤数据集中出现频次 >4 的常见函数(第 5.2 节参数), 将剩余函数作为核心函数集合 V 。需要说明的是, 由表 5 可以看出, 常见函数过滤步骤仅过滤了 189 个函数(受数据集规模限制, 部分常见函数的出现次数可能少于过滤的阈值), 但在真实场景中编解码等常见函数的数量不容小觑。在叠加常见函数过滤之后, 准确率有所提升(图 11), 然而, 已有的三种过滤手段均未考虑任何语义和语法结构信息, 因此难以保证剩余函数即为具有项目代表语义的核心函数。

最后, 分析叠加中心性分析方法进行核心函数进行定位后的情况进行核心函数定位。表 5 显示, 中心性分析有效地过滤了 30 万余个非核心函数, 剩余的核心函数集合仅有 9 万余个。由于介数中心性分析方法能够筛选出在函数调用图中起到关键“控制”作用的函数, 且这些函数在多个模块中被反复调用, 承担着关键的代码逻辑, 与核心函数的定义高度契合, 通过介数中心性分数能够有效定位出 TPL 核心函数, 检测准确率相比之前再次上升(图 11)。值得注意的是, 虽然此时召回率有所下降, 但核心函数数量从 40 万减少到 9 万, 大幅减少了后续依赖分析的比较次数, 提高了检测效率。同时,

由于函数调用图生成工具 Doxygen 的准确性限制, 部分假负例未能被完全排除(详见第 5.4 节)。总体而言, 从图 11 中的 F1 分数可以看出, 基于中心性分析的核心函数定位方法有助于 CentSCA 提升整体检测效果。

回答 RQ2: 本节进行消融实验, 拆分每个非核心函数过滤(以及通过中心性分数定位核心函数)步骤, 以分析每个步骤对 CentSCA 性能提高的贡献。实验表明, 不同类型的非核心函数(重复函数、简单函数和常见函数)对软件依赖分析的准确性有明显的负向影响, 且针对性设计的每个过滤步骤都能够有效排除对应的非核心函数的影响。此外, 中心性分析方法由于考虑了代码结构, 有助于精准定位 TPL 的核心函数, 以此进一步提升检测性能。

5.4 RQ3: 与最先进 SCA 方法相比, CentSCA 检测软件依赖的效果如何?

5.4.1 设置

第 5.4 节通过对比实验来评估 CentSCA 对于 C/C++ 软件成分分析的检测性能实验采用本文构建的 GT 数据集, 共有 1016 条软件依赖关系, 分别记录 CentSCA 和对比工具应用于该数据集时的 TP、FP、FN、准确率、召回率和 F1 分数。实验选取最先进的两个 C/C++ SCA 方法作为对比方法(CENTRIS 和 OSSFP)。

需要说明的是, 对比工具的参数选择均遵从其文章建议的最佳参数。具体而言, CENTRIS 对于项目相似性的阈值选择为 0.1, 即认定存在 10% 及以上相同函数的项目之间存在依赖关系; OSSFP 采用两组参数进行实验, 其中 OSSFP 将简单函数过滤比例 θ_0 设置为 0.5, 针对常见函数的过滤为 0.2, 即将 tf-idf 分数排名在前 20% 的函数视为常见函数并过滤。

5.4.2 结果

与 CENTRIS 的比较。根据表 6 的结果可以看出, 相比于其他两个方法, CENTRIS 的真正例数量明显较少, 且假负例的数量较高。这是由于 CENTRIS 在排除相同函数后, 侧重于考虑两个项目之间整体的相似度(具有相同函数的比例), 且需要预设项目的相似度阈值来判断项目之间是否存在依赖于预设的相似度阈值, 将会导致以下问题: (1) 当阈值设置较低时, 则无法规避项目中的简单函数和常用函数的影响; 假设两个项目的编码风格都倾向于封装简单函数(如, 主体仅有一行 return 语

句的函数)辅助功能衔接,那么由于这两个项目共享多个相同函数,极有可能造成 CENTRIS 的误报;(2)当阈值设置较高时,虽然能在一定程度上避免误报,但会牺牲召回率。如表 6 报告的数据,CENTRIS 的表现虽然取得了可以接受的假正例数量,但是其假负例数量远超其他工具,同时真正例的数量也处于最低水平,因此只取得了 0.403 的召回率。此外,CENTRIS 计算相同函数在项目中的占比需要一一对比两个项目之间的全部函数,因此在时间开销方面严重受限,难以应用到真实环境。

CentSCA 的方法并非从项目之间整体的相似度考虑,而是通过关注具有项目代表性的核心函数是否被引用,来判断项目之间的依赖关系。由此,可以有效避免项目大量非核心函数对于项目相似性的影响,从而提升检测的准确性。因此,CentSCA 在准确率和召回率方面都具有明显优势。

与 OSSFP 的比较。表 6 中记录了 OSSFP 在两

表 6 CentSCA 与 CENTRIS 和 OSSFP 的对比

SCA 方法	GT	TP	FP	FN	准确率	召回率	F1 分数
CENTRIS	1016	410	302	607	0.537 35	0.403 54	0.460 93
OSSFP ¹	1016	536	91	429	0.854	0.55	0.673
OSSFP ²	1016	827	490	189	0.628	0.8139	0.709 26
CentSCA	1016	715	250	301	0.740 93	0.703 74	0.721 86

注:(1)GT: ground truth 下正确的依赖关系的个数。TP: 真正例。FP: 假正例。FN: 假负例;(2)OSSFP¹: 采用 OSSFP 工具原文中选取的最佳参数;(3)OSSFP²: 采用第 5.2 节中筛选的最佳参数。

另一方面,较于 OSSFP²,由于 CentSCA 加强了对代码结构的考虑,对于 TPL 核心函数的定位更加准确,并由此取得了更高的准确率;针对召回率,由于基于中心性分析的核心函数定位方法是在非核心函数过滤(即, OSSFP 的方法)的基础上加强了筛选力度,且受限于函数调用图生成工具 Doxygen 的误差(在假负例分析中详细提到),因此遗漏了一些中心性分数不高的核心函数,导致假负例的数量增多。

假正例分析。下面分析 CentSCA 测出的假正例情况,主要有以下原因:

首先,CentSCA、OSSFP 和 CENTRIS 存在两个共有的误报原因:(1)受到数据集范围的制约,数据集无法覆盖开源仓库的全部项目。因此,当两个项目同时依赖于一个未纳入数据集的 TPL 时,SCA 方法即会认为上述两个项目间存在依赖关系,由此报告出假正例;(2)受限于部分项目版本管理不规范问题,导致函数创建时间不准确。因此,SCA 方

种参数条件下的结果,通过 F1 分数可以看出,CentSCA 整体上比 OSSFP 取得了更好的检测结果。具体而言,针对召回率, OSSFP¹ 对于常见函数过滤的 tf-idf 分数阈值设置为 0.2,通过分析数据集全部函数的 tf-idf 值分布发现,大部分函数的出现次数为 1 次(词频为 1 的函数在整个数据集中的占比约为 88%)。因此,若只保留 tf-idf 分数前 20% 的函数作为核心函数,将错误地将仅出现一次或两次的函数作为常见函数过滤,导致大量核心函数遗漏,导致检出 TP 数较少,召回率仅有 0.55。针对准确率,由于参数选择, OSSFP 在常见函数阶段过滤函数的数量远大于 CentSCA,多过滤的部分可能覆盖真实的常见函数和核心函数。因此, OSSFP 的 TP 数量和 FP 数量都远低于 CentSCA,根据准确率的计算公式, OSSFP 的准确率优于 CentSCA,但同时牺牲了召回率。

法在重复函数过滤时会错误识别最早编写的相同函数,导致依赖检测产生假正例。通过人工抽样检查 30 个 CentSCA 检出的假正例,发现其中 7 个由数据集不充分造成,8 个由仓库版本管理不规范导致。

此外, CentSCA 对常见函数的过滤有所遗漏。由于数据集不够全面,出现次数较少的函数无法被 CentSCA 识别为常见函数。例如,项目 Libexpat^① 和项目 expat^② 中存在相同函数,经人工分析,该函数为实现解析 XML 文档功能的常见函数,并非上述两个项目专属的功能,因此事实上两项目之间并不存在依赖关系。然而,数据集中这个函数出现的次数为 2,小于 CentSCA 对于常见函数阈值(出现次数 4 次以上),因此无法被有效过滤,进而产生假正例。根据前文分析, OSSFP 设置 tdf-idf 分数最高的 20% 的函数为常见函数(出现次数大于 1 次),虽然

① Libexpat <https://github.com/libexpat/libexpat>

② expat <https://github.com/expat/expat>

可以成功过滤上述实例,但是 OSSFP 对于常见函数的过度过滤,无可避免地降低了召回率。通过人工抽样检查 30 个 CentSCA 检出的假正例,发现 15 个由常见函数过滤遗漏导致。

假负例分析。下面分析 CentSCA 测出的假负例情况,主要有以下原因:

首先,三个工具共同存在由于部分项目版本管理不佳(与上文假正例分析中原因类似)导致的假负例情况。因为版本控制不当,SCA 工具记录的重复函数创建时间早于原创函数,因此测出完全相反的依赖关系,即原仓库依赖于引用仓库,进而产生假负例。此外,原创函数所属项目记录错误,导致真正原创函数的项目的被依赖关系无法被检测到,造成漏报。

其次,CentSCA 受限于函数调用图生成工具 Doxygen 的不准确性,导致检出假负例。具体而言,CentSCA 使用 Doxygen 来解析项目并生成以函数为节点的函数调用图,然后通过 Networkx 包以调用图为输入,计算项目中各函数的中心性分数。然而,经过人工检查,Doxygen 工具有以下几个方面不足:(1)Doxygen 工具无法解析预处理指令(例如, #if、#ifdef 和 #ifndef 等),因此在预处理指令下被定义的函数无法被 Doxygen 识别,导致生成的函数调用图中缺失此类函数。由于上述函数的缺失,部分项目的函数调用图成为不连通图,因此在基于中心性分析的核心函数定位步骤中,不仅无法计算此类函数(节点)的中心性分数,也会进一步影响其他到函数(节点)的中心性分数计算;(2)Doxygen 在解析部分仓库的 *.h 与 *.hpp 文件时,无法成功解析其中定义的函数。无法被成功解析的函数将会被默认丢弃,因此造成部分依赖关系的丢失,从而产生假负例。随机抽取的 30 个 CentSCA 测出的假负例中,有 3 个与上述原因有关。

回答 RQ3: 本节设计对比实验,在同一数据集下将 CentSCA 与现有工具 CENTRIS 与 OSSFP 进行对比。实验结果表明,CentSCA 取得了最高的 F1 分数,分别相较于采用源参数的 CENTRIS 和 OSSFP 提升了 26% 和 5%。

5.5 RQ4:CentSCA 在供应链漏洞检测中的应用效果

5.5.1 设置

供应链漏洞检测作为软件成分分析中的关键

下游任务,其高误报率一直是业界关注的难点问题。准确的依赖关系识别是漏洞检测可靠性的基础,而精确识别 TPL 的核心函数则是提高依赖识别准确率的关键一环。

CentSCA 通过结合语法与语义特征,准确识别 TPL 中的核心函数,从而有效减少因依赖识别错误引发的漏洞误报,提升漏洞检测的整体准确性。为直观展示核心函数识别对实际 SCA 场景中供应链漏洞检测效果的影响,本节以现实世界中的开源项目 filament^①(谷歌旗下流行开源项目)为案例,在第 5.2 节确定的最佳参数设置下,分别使用 CentSCA 与 OSSFP 识别 filament 所依赖的 TPLs。随后,借鉴 V1SCAN 方法^[35]中对于供应链漏洞检测能力的评估流程,利用知名漏洞检测工具 Snyk^②的开源漏洞数据库,查询所识别的 TPL 是否存在已知漏洞,并将其作为 filament 的供应链漏洞报告进行对比分析。

5.5.2 结果

表 7 展示了 CentSCA 与 OSSFP 在 filament 项目上的 SCA 结果及对应的供应链漏洞检测效果。结果显示,CentSCA 相较于 OSSFP 将错误识别的依赖关系(SCA FP)数量从 12 降至 5,漏洞误报数由 35 降为 0,显著提升了漏洞检测的准确性。

表 7 CentSCA 与 OSSFP 对 filament 仓库的 SCA 结果

方法	SCA TP	SCA FP	供应链漏洞误报数量	供应链漏洞漏报数量
CentSCA	16	5	0	4
OSSFP	21	12	35	2

OSSFP 仅基于函数的语法特征识别核心函数,判断标准较为宽松,缺乏对代码语义的充分考虑,导致将部分实际并不承担核心功能的函数误判为核心函数,从而产生了 12 个错误依赖。这些错误检测的 TPL 中存在 35 个已知漏洞,最终导致 Snyk 报告中出现 35 个漏洞误报。

相比之下,CentSCA 采用语法与语义的双重筛选机制,显著提升了核心函数识别的准确性,从源头上减少了错误依赖,降低了漏洞检测中的误报风险,证明了其在供应链安全检测任务中的实际应用价值。

回答 RQ4: 本节通过将 CentSCA 与 OSSFP 应

① filament, <https://github.com/google/filament>

② snyk, <https://github.com/snyk/cli>

用于一个现实世界中的开源项目 filament 进行供应链漏洞检测,以证明 CentSCA 在供应链漏洞检测中的应用效果。实验结果表明, CentSCA 相较于 OSSFP 将错误识别的依赖关系数量从 12 降至 5,漏洞误报数由 35 降为 0,显著提升了漏洞检测的准确性。

6 总结与展望

针对已有 C/C++ 语言 SCA 工具由于缺乏对于代码语义的考虑,难以精确定位 TPL 核心函数的问题,本文提出了一种基于中心性分析的软件成分分析方法, CentSCA。该方法首先过滤重复函数、简单函数和常见函数,然后利用中心性分析方法对函数调用图中的关键节点进行分析,从而准确定位具有 TPL 语义代表性的核心函数,并进行软件依赖检测。为了验证 CentSCA 的有效性,构建了一个包含 510 个开源项目和 1016 条依赖关系的真实数据集,并在该数据集上分别开展了参数实验、消融实验和对比实验。实验结果表明,在最优参数下, CentSCA 能够取得 0.72 的 F1 分数,并且 CentSCA 的每一个函数过滤(或核心函数定位)步骤均有助于提高核心函数定位的准确性。相较于现有最先进 SCA 工具 CENTRIS 和 OSSFP,检测效果分别提升了 26% 和 5%,证明了 CentSCA 在实际应用方面的有效性。

目前工作仍存在一定局限性。首先,根据第 5.4 节对于 CentSCA 测出的假正例和假负例分析,数据集不全面的问题对于工具测试的影响较大,因此未来工作会继续补充完善数据集。具体而言,进一步补充项目和依赖条目数量,并在现有依赖标注的基础上,加入人工对于 TPL 核心函数的筛选,然后人工检查项目是否引用了 TPL 的核心函数,从而提升数据集依赖关系的标注质量;其次,现阶段 CentSCA 通过代码规范化后,对 TPL 的核心函数和项目函数进行代码哈希值匹配,因此仅能检测由于 Type-1 和 Type-2 类型克隆导致的软件依赖关系。未来工作考虑加入复杂类型克隆检测算法,以支持 CentSCA 检测项目在更改更多代码的情况下对 TPL 依赖的情况。最后,未来工作考虑结合代码大模型,在核心函数定位时加入更多代码语义信息。例如,利用代码大模型的函数摘要能力生成代码摘要。然后与 TPL 说明文档进行语义关联匹配,从而更精确地定位项目的核心函数。

参 考 文 献

- [1] Lopes C V, Maj P, Martins P, et al. Déjàvu: A map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 2017, 1(OOPSLA): 1-28.
- [2] Ladisa P, Plate H, Martinez M, et al. Sok: Taxonomy of attacks on open-source software supply chains//*Proceedings of the 2023 IEEE Symposium on Security and Privacy*. San Francisco, USA, 2023: 1509-1526
- [3] Duan R, Bijlani A, Xu M, et al. Identifying open-source license violation and 1-day security risk at large scale//*Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. Dallas, USA, 2017: 2169-2185
- [4] Woo S, Park S, Kim S, et al. Centris: A precise and scalable approach for identifying modified open-source software reuse//*Proceedings of the 43rd International Conference on Software Engineering*. Madrid, Spain, 2021: 860-872
- [5] Wu J, Xu Z, Tang W, et al. Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions//*Proceedings of the 45th International Conference on Software Engineering*. Melbourne, Australia, 2023: 270-282
- [6] Gharehyazie M, Ray B, Keshani M, et al. Cross-project code clones in github. *Empirical Software Engineering*, 2019, 24(3): 1538-1573
- [7] Abdalkareem R, Shihab E, Rilling J. On code reuse from stack overflow: An exploratory study on android apps. *Information and Software Technology*, 2017, 88: 148-158
- [8] Reid D, Jahanshahi M, Mockus A. The extent of orphan vulnerabilities from code reuse in open source software//*Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh, USA, 2022: 2104-2115
- [9] Mao T Y, Wang X Y, Chang R, et al. Software supply chain security analysis technology for java language ecosystem. *Chinese Journal of Software*, 2023, 34(6): 2628-2640(in Chinese)
(毛天宇, 王星宇, 常瑞, 申文博, 任奎. 面向Java语言生态的软件供应链安全分析技术. *软件学报*, 2023, 34(6): 2628-2640)
- [10] Zerouali A, Mens T, Decan A, et al. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Software Engineering*, 2022, 27(5): 107
- [11] Alfadel M, Costa D E, Shihab E. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering*, 2023, 28(3): 59
- [12] Wang H, Guo Y, Ma Z, et al. Wukong: A scalable and accurate two-phase approach to android app clone detection//*Proceedings of the 2015 International Symposium on Software Testing and Analysis*. Baltimore, USA, 2015: 71-82
- [13] Ma Z, Wang H, Guo Y, et al. Libradar: Fast and accurate detection of third-party libraries in android apps//*Proceedings of the 38th International Conference on Software Engineering Companion*. Austin, USA, 2016: 653-656
- [14] Li M, Wang W, Wang P, et al. Libd: Scalable and precise third-party

- library detection in android markets//Proceedings of the 39th International Conference on Software Engineering. Buenos Aires, Argentina, 2017: 335-346
- [15] Zhan X, Fan L, Chen S, et al. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications//Proceedings of the 43rd International Conference on Software Engineering. Madrid, Spain, 2021: 1695-1707
- [16] Backes M, Bugiel S, Derr E. Reliable third-party library detection in android and its security applications//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria, 2016: 356-367
- [17] Zhang Y, Dai J, Zhang X, et al. Detecting third-party libraries in android applications with high precision and recall//Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering. Lecce, Italy, 2018: 141-152
- [18] Yuan Z, Feng M, Li F, et al. B2sfinder: Detecting open-source software reuse in cots software//Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. San Diego, USA, 2019: 1038-1049
- [19] Yang C, Xu Z, Chen H, et al. ModX: Binary level partially imported third-party library detection via program modularization and semantic matching//Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, USA, 2022: 1393-1405
- [20] Jiang L, An J, Huang H, et al. BinaryAI: Binary software composition analysis via intelligent binary source code matching//Proceedings of the 46th International Conference on Software Engineering. Lisbon, Portugal, 2024: 1-13
- [21] Haq I U, Caballero J. A survey of binary code similarity. ACM Computing Surveys, 2021, 54(3): 1-38
- [22] Dann A, Plate H, Hermann B, et al. Identifying challenges for oss vulnerability scanners A study & test suite. IEEE Transactions on Software Engineering, 2021, 48(9): 3613-3625
- [23] Intiaz N, Thorn S, Williams L. A comparative study of vulnerability reporting by software composition analysis tools//Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. Bari, Italy, 2021: 1-11
- [24] Zhao L, Chen S, Xu Z, et al. Software composition analysis for vulnerability detection: An empirical study on java projects//Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. San Francisco, USA, 2023: 960-972
- [25] Sheneamer A, Kalita J. A survey of software clone detection techniques. International Journal of Computer Applications, 2016, 137(10): 1-21
- [26] Roy C K, Cordy J R. A survey on software clone detection research. Queen's School of computing TR, 2007, 541(115): 64-68
- [27] Koschke R. Survey of research on software clones. Dagstuhl Seminar Proceedings, 2007: 1-24
- [28] Sajnani H, Saini V, Svajlenko J, et al. Sourcerercc: Scaling code clone detection to big-code//Proceedings of the 38th International Conference on Software Engineering. Austin, USA, 2016: 1157-1168
- [29] Roy C K, Cordy J R, Koschke R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming, 2009, 74(7): 470-495
- [30] Oman P, Hagemester J. Metrics for assessing a software system's maintainability//Proceedings of the Proceedings Conference on Software Maintenance 1992. Orlando, USA, 1992: 337-344
- [31] Hiemstra D. A probabilistic justification for using tf-idf term weighting in information retrieval. International Journal on Digital Libraries, 2000, 3(2): 131-139
- [32] Freeman L C. A set of measures of centrality based on betweenness. Sociometry, 1977: 35-41
- [33] Freeman L C. Centrality in social networks conceptual clarification. Social Networks, 1978, 1(3): 215-239
- [34] Alvarez-Socorro A J, Herrera-Almarza G C, González-Díaz L A. Eigencentality based on dissimilarity measures reveals central nodes in complex networks. Scientific Reports, 2015, 5(1): 17095
- [35] Woo S, Choi E, Lee H, et al. V1scan: Discovering 1-day vulnerabilities in reused c/c++ open-source software components using code classification techniques//Proceedings of the 32nd USENIX Security Symposium. Anaheim, USA, 2023: 6541-6556



HU Yu-Tao, Ph.D., postdoc researcher. Her research interests include software supply chain security, vulnerability detection, and code clone analysis.

DONG Jia-Jun, M. S. candidate. His research interests focus on software supply-chain security and vulnerability detection.

Background

In the burgeoning landscape of open-source software, developers widely incorporate Third-Party Libraries (TPLs)

SUN Ming-Yuan, M.S. His research interest focuses on software supply-chain security.

ZHANG Yun-He, Ph. D., associate professor. His research interests include computer networks and network security.

WU Yue-Ming, Ph. D., associate professor. His research interests include open-source software security, artificial intelligence security, and mobile security.

ZOU De-Qing, Ph. D., professor. His research interests include software security, vulnerability detection, proactive defense, and software-defined security.

to reuse existing functionalities, greatly accelerating development and reducing redundant work. However, this

practice introduces significant security risks, as latent vulnerabilities within TPLs can compromise projects. Accurately identifying TPL dependencies is therefore essential for timely security responses. Software Component Analysis (SCA) technology serves this purpose, helping developers detect TPLs and enhance system security.

While many languages like Java, Python, and JavaScript benefit from official package managers (e.g., Maven, npm) that generate clear Software Bills of Materials (SBOM) for dependencies, the C/C++ language lacks such standard tools. Existing C/C++ SCA methods fall into two main categories: those based on global TPL features and those based on key TPL features. Both detect dependencies by matching features between the TPL and the project.

Methods using global features attempt comprehensive information capture but often suffer from sensitivity to noise and code changes (e.g., OSSProlice's reliance on directory structures), leading to poor robustness and high costs (e.g., CENTRIS's pairwise function comparisons struggling with complex clones). Methods based on key features, such as OSSFP, aim for improved robustness by focusing on representative functions. However, OSSFP's filtering process for identifying these "core functions" lacks crucial consideration of code semantics, hindering the accuracy of core function identification and limiting overall detection performance. The current state of C/C++ SCA is thus limited by insufficient robustness, scalability, and the inability to accurately capture the semantic essence of TPLs.

This background highlights a critical research challenge:

How to accurately locate TPL core functions that truly represent key semantics, specifically based on function semantic features?

To address this challenge, this paper proposes a Software Component Analysis method based on Centrality Analysis (CentSCA). CentSCA enhances key function identification by first applying standard filtering, then generating a Function Call Graph (FCG) to represent code semantics. Centrality analysis on the FCG identifies high-scoring functions, which are then intersected with filtered functions to precisely determine the TPL's core functions. Dependency detection is then performed by checking for clones of these semantically-derived core functions in the target project.

Evaluated on a real C/C++ dataset with 1016 dependency relationships, CentSCA achieved an F1 score of 0.72 after parameter optimization. Ablation studies confirmed the necessity of CentSCA's core components. Compared to the state-of-the-art C/C++ SCA tools, CENTRIS and OSSFP, CentSCA significantly improves detection effectiveness, increasing the F1 score by 26% and 5% respectively. This demonstrates that CentSCA's semantic-aware approach to key function identification successfully addresses the limitations of existing methods, advancing the state of the art in C/C++ software component analysis.

This paper was supported by the National Natural Science Foundation of China (No. U2336203). The Study aims to enhance the effectiveness of software supply chain analysis in C/C++ through centrality analysis methods.