软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(6):2604−2642 [doi: 10.13328/j.cnki.jos.007308] [CSTR: 32375.14.jos.007308]

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

软件供应链 SBOM 关键技术研究*

孙泽雨[1,2,3], 吴敬征[1,4,5], 凌 祥[1,4,5], 魏怡琳[1], 罗天悦[1], 武延军[1,4,5]

[1](中国科学院 软件研究所 智能软件研究中心, 北京 100190)

[2](国科大杭州高等研究院 智能科学与技术学院, 浙江 杭州 310024)

[3](中国科学院大学, 北京 100049)

[4](基础软件与系统重点实验室 (中国科学院 软件研究所), 北京 100190)

[5](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

通信作者: 吴敬征, E-mail: jingzheng08@iscas.ac.cn

摘 要: 供应链级别的开源软件及组件复用是当前软件开发的主流模式. 该模式避免了重复开发, 降低了研发成本, 提高了开发效率, 但是也不可避免地存在组件的来源未知, 成分不清, 漏洞不明, 许可证违规等问题. 为解决上述问题, 研究人员提出了软件物料清单 (software bill of material, SBOM). SBOM 详细列出了构成软件的组件及组件之间的关系, 揭示了潜在的和已知的威胁, 使软件透明化. 自提出以来, 国内外研究人员针对 SBOM 的研究主要聚焦在 SBOM 的现状、应用和工具上, 缺少理论化、体系化的研究. 综述 SBOM 的背景、基本概念、生成技术、工具及性能分析、应用、挑战与趋势, 并提出融合细粒度安全漏洞感知, 许可证冲突检测的 SBOM+, 以期从概念、技术、工具、应用和发展等方面为 SBOM、软件开发、供应链安全等研究人员提供支撑.

关键词: 软件供应链; 开源软件; 软件成分; SBOM

中图法分类号: TP311

中文引用格式: 孙泽雨, 吴敬征, 凌祥, 魏怡琳, 罗天悦, 武延军. 软件供应链 SBOM 关键技术研究. 软件学报, 2025, 36(6): 2604–2642. http://www.jos.org.cn/1000-9825/7308.htm

英文引用格式: Sun ZY, Wu JZ, Ling X, Wei YL, Luo TY, Wu YJ. Research on Key Technologies of SBOM in Software Supply Chain. Ruan Jian Xue Bao/Journal of Software, 2025, 36(6): 2604–2642 (in Chinese). http://www.jos.org.cn/1000-9825/7308.htm

Research on Key Technologies of SBOM in Software Supply Chain

SUN Ze-Yu[1,2,3], WU Jing-Zheng[1,4,5], LING Xiang[1,4,5], WEI Yi-Lin[1], LUO Tian-Yue[1], WU Yan-Jun[1,4,5]

[1](Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](School of Intelligent Science and Technology, Hangzhou Institute for Advanced Study, UCAS, Hangzhou 310024, China)

[3](University of Chinese Academy of Sciences, Beijing 100049, China)

[4](Key Laboratory of System Software (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[5](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: The current mainstream mode of software development is the supply chain-level reuse of open-source software and components. It avoids repetitive development, reduces research and development costs, and enhances development efficiency. However, it inevitably brings about issues such as unknown component sources, unclear component compositions, unidentified component vulnerabilities, and license violations. To address these issues, researchers propose software bill of materials (SBOM). SBOM provides a detailed list of software components and their relationships, reveals potential and known threats, and makes software transparent. Since its proposal, research on SBOM by researchers both at home and abroad mainly focus on its current status, applications, and tools, lacking theoretical

* 基金项目: 国家自然科学基金 (62202457); 中国科学院战略性先导科技专项 (XDA0320401); 2023 年开源社区软件物料清单 SBOM 平台项目 (E3GX310201)
收稿时间: 2024-04-30; 修改时间: 2024-06-17, 2024-08-19; 采用时间: 2024-10-14; jos 在线出版时间: 2025-03-19
CNKI 网络首发时间: 2025-03-19

and systematic research. This study presents a comprehensive review of the background, basic concepts, generation techniques, tools and performance analysis, applications, challenges, and trends of SBOM. It also proposes the new concept of SBOM+, which integrates fine-grained security vulnerability perception and license conflict detection. The aim is to provide support for researchers engaged in SBOM, software development, and supply chain security from the perspectives of concepts, technologies, tools, applications, and development.

Key words: software supply chain; open source software; software component; software bill of material (SBOM)

1 引 言

Michael Porter 于 1985 年提出价值链模型[1]. 该模型描述企业通过一系列活动将原材料转化为最终产品并交付给用户的过程. 供应链的理念源自该模型. 最初, 供应链被视为物流企业的运作模式, 其结构单一且内部联系紧密. 随着认知的深化, 供应链逐渐拓展至企业外部, 向上延伸至供应商, 向下延伸至消费者. 供应链由最初的单一链接逐步演化为复杂的网状链接.

软件开发与供应链存在逻辑相似性, 可以将供应链的概念应用到软件开发. Holdsworth[2]在 1995 年首次提出软件供应链的概念, 他认为根据具体标准设计和编写软件的整个过程是软件供应链. 何熙巽等人[3]于 2020 年提出软件供应链的概念, 将软件供应链视为一个系统, 包括设计阶段, 开发阶段和交付阶段. 开发人员在设计阶段和开发阶段完成软件的编写, 最终在交付阶段将软件交付给用户. Liang 等人[4]于 2020 年提出开源软件供应链的概念, 认为开源软件供应链的核心是由开源软件间的依赖关系形成的对最终产品的供应关系. Ji 等人[5]于 2023 年提出开源软件供应链的概念, 认为开源软件供应链是一个供应关系网络, 记录了开源软件使用的所有开源组件之间的供应关系. 软件供应链概念的演进, 逐步清晰刻画了软件之间的链接特征.

随着软件开发模式的优化, 软件供应链的概念应用日益凸显. 传统的软件开发模式下, 主要由单一供应商独立开发整个软件, 而当前主流软件开发模式则强调复用已有软件和组件, 辅助软件开发. 新的模式避免了重复工作, 降低了研发成本, 提高了开发效率. Sonatype 于 2023 年发布的软件供应链调查报告[6]显示, 开源项目的数量持续增长. 从 2022 年到 2023 年, 开源项目的数量平均增长了 29%. 开源软件的下载量在 2023 年增长了 33%. 这一趋势证实了软件复用对软件开发的重要性.

尽管当前以软件复用为主的开发模式具有诸多优势, 但由于无法确定被复用软件的来源、成分、漏洞和合规性, 导致该开发模式存在许多安全隐患. “源图”[7]将软件供应链的风险分为 3 类: ①可维护性风险; ②安全性风险; ③合规性风险. 其中, 可维护性风险是由软件供给中断或停服带来的威胁, 安全性风险是由漏洞引发的危害, 合规性风险是由许可证冲突导致的问题. 例如, 恶意的 npm[8]开源代码维护者故意损坏自己的代码, 给复用该代码的其他开发者带来可维护性风险. 攻击者在 SolarWinds[9,10]软件的更新过程中植入后门, 非法窃取数据, 给数千名用户带来安全性风险. 黑客利用 Apache Log4j[11]的漏洞在易受攻击的系统上执行恶意代码, 给数万名复用 Apache Log4j 的开发者带来安全性风险. Katzer[12]复用了 Jacobsen 的软件却没有遵守许可要求, 给 Katzer 带来合规性风险. 最终法院裁定 Katzer 违反开源软件许可协议的要求, 构成著作权侵权. 针对软件供应链的攻击层出不穷, 各国逐渐重视软件供应链的安全.

美国率先提出应对软件供应链威胁的措施. 美国于 2021 年发布了关于改善国家网络安全的行政命令[13], 要求国家标准及技术研究所 (NIST) 发布指南, 确定增强软件供应链安全, 并公布软件物料清单 (software bill of material, SBOM) 的最低要素. 类似于食品配料表列出了构成食品的原材料, 揭示了食品的成分和潜在过敏源, SBOM 列出了构成软件的组件及组件之间的关系, 揭示了软件的构成要素和可能的缺陷, 使软件透明化, 同时也使软件供应链透明化, 协助组织应对风险.

SBOM 自提出以来, 逐渐展现其重要价值. SBOM 提供的软件结构和依赖关系的视图协助组织管理软件, 追踪潜在风险. 组织借助 SBOM 实现快速判断开发的软件, 选购的软件或正在使用的软件是否存在安全风险, 是否符合内部政策和法规的要求, 并及时采取应对措施, 降低风险带来的成本. 当前针对 SBOM 开展了广泛的研究, 但主要集中在 SBOM 的现状、应用和工具上, 缺乏理论化和体系化, 不利于 SBOM 的推广和发展. 本文从背景、基本概念、生成技术、工具及性能分析、应用、挑战与趋势等多个方面综述了 SBOM, 并提出了融合细粒度安全漏

洞感知和许可证冲突检测的 SBOM+, 旨在从概念、技术、工具、应用和发展等方面为 SBOM 及相关从业人员提供支撑.

本文第 2 节介绍了 SBOM 的背景. 第 3 节阐述了 SBOM 的基本概念. 第 4 节探讨了 SBOM 的生成技术. 第 5 节介绍了 SBOM 工具并分析了工具的性能. 第 6 节探讨了 SBOM 的应用. 第 7 节展望了 SBOM 的趋势和挑战. 最后对本文进行了总结.

2 背 景

为保障软件供应链的安全, 美国于 2021 年提出 SBOM, 然而实际上 SBOM 的提出可以追溯到更早的时期. 图 1 展示了 SBOM 的发展历程.

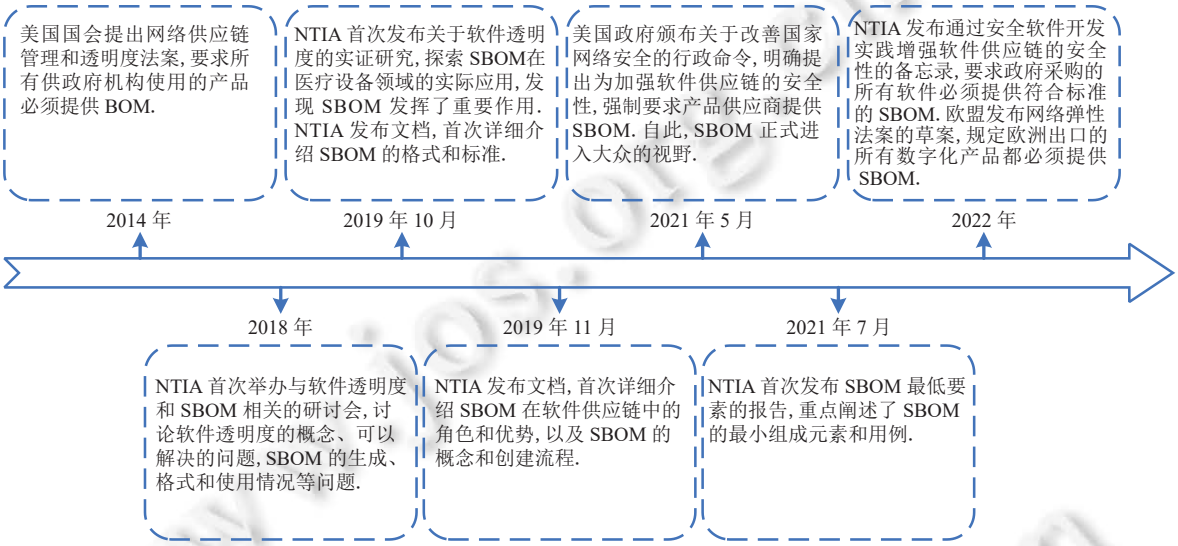

图 1 SBOM 的发展历程

2014 年, 美国国会提出了网络供应链管理和透明度法案[14], 要求所有供政府机构使用的产品必须提供物料清单 (bill of materials, BOM), 以便了解产品的构成, 及时响应安全威胁. 尽管法案未能通过, 但引起了相关从业人员对 BOM 和透明度的高度关注. 为促进软件透明度的发展, 美国国家电信和信息管理局 (NTIA) 于 2018 年 7 月首次举办与软件透明度和 SBOM 相关的研讨会[15]. 与会者讨论了软件透明度的概念, SBOM 的生成、格式等内容.

自 2018 年起, NTIA 持续举办与软件透明度和 SBOM 相关的研讨会, 并于 2019 年 10 月 1 日首次发布关于软件透明度的实证研究[16], 研究探索了 SBOM 在医疗设备领域的实际应用, 发现 SBOM 为医疗设备的采购、管理和风险评估提供了极大便利. 于是, NTIA 迅速于 2019 年 10 月 25 日发布相关文档[17], 首次详细介绍了 SBOM 的格式和标准, 并于同年 11 月发布相关文档[18,19], 首次详细介绍了 SBOM 在软件供应链中的角色和优势, 以及 SBOM 的概念和创建流程.

2021 年 5 月, 美国政府颁布关于改善国家网络安全的行政命令[13], 明确提出 SBOM, 并强制要求产品供应商提供 SBOM. 命令中还提到, 软件开发通常通过组合现有软件来构建新的软件, 被组合的软件作为组件成为新软件的一部分, 发挥着重要作用. 类似于食品的成分表, SBOM 详细列举了组成软件的组件和组件之间的关系. 开发者通过 SBOM 了解软件的构成, 及时响应漏洞; 采购者借助 SBOM 对所购软件进行漏洞检测和许可证合规性分析, 评估软件风险; 操作者利用 SBOM 快速确定软件是否存在安全隐患. 自此, SBOM 正式进入大众视野.

应行政命令的要求, NTIA 于 2021 年 7 月发布 SBOM 最低要素的报告[20], 首次重点阐述了 SBOM 的最小组成元素和用例. 后续, NTIA 于 2022 年发布通过安全软件开发实践增强软件供应链的安全性的备忘录[21], 要求政府采购的所有软件必须提供符合标准的 SBOM, 同月, 欧盟发布了网络弹性法案的草案[22], 所有数字化产品都必

须提供 SBOM, 违反规定的公司将面临巨额罚款.

表 1 列举了国内外发布的与软件供应链相关的法律法规和指南.

表 1 国内外发布的与软件供应链相关的法律法规和指南

国家/地区	时间	名称	概要
中国	2021/11	“十四五”软件和信息技术服务业发展规划[23]	该规划突出开源在软件产业创新和数字中国建设中的关键作用, 强调开源已成为全球软件创新的主导模式
	2021/12	“十四五”国家信息化规划[24]	该规划提出加快国内开源社区建设, 构筑完善的开源生态链
	2022/6	软件物料清单实践指南[25]	该指南由中国信息通信研究院协同多位专家学者发布, 分别从价值用例, 组成要素, 格式与工具等10个维度指导SBOM的落地实践
	2023/8	新产业标准化领航工程实施方案(2023–2035年)[26]	该方案提出制定开源标准, 包括术语, 许可证, 互联互通, 项目成熟度, 社区运营治理以及供应链管理等方面
	2024/4	GB/T 43698-2024 网络安全技术 软件供应链安全要求[27]	该标准明确了安全目标, 规定了风险管理, 组织管理和供应活动管理的要求
	2024/4	GB/T 43848-2024 网络安全技术 软件产品开源代码安全评价方法[28]	该标准为评估开源代码的安全提供了方法和指标, 指导开发者系统性地评估开源代码的安全性
美国	2012/1	全球供应链安全国家战略[29]	该战略将全球供应链提升为国家安全战略, 确立安全与高效为两大目标
	2014/12	网络供应链管理和透明度法案[14]	该法案旨在确保美国政府开发或购买的软件, 固件或产品的完整性, 要求提供包含所有第三方和开源组件的SBOM
	2021/5	关于改善国家网络安全的行政命令[13]	该命令呼吁联邦政府实施严格的软件评估机制, 提升软件供应链的安全性和完整性
	2022/5	网络供应链风险管理计划[30]	该计划提供了企业在供应链各阶段识别, 评估和应对风险的主要措施和做法, 分享了供应链攻击趋势和最佳实践
	2023/3	开源软件供应链安全相关立法[31]	该立法确立网络安全和基础设施安全局局长在开源软件安全方面的职责, 以促进开源软件的稳定与可靠
	2023/9	CISA开源软件安全路线图[32]	该路线图要求加强开源生态系统的联动协作, 构建完善的开源软件安全保障工作
	2023/11	保障软件供应链安全: SBOM推荐实践指南[33]	该指南协助软件开发人员和供应商管理开源软件和软件物料清单, 维护并了解软件安全
欧盟	2020/10	开源软件战略 (2020–2023)[34]	该战略旨在推动并充分利用开源代码带来的变革, 创新和协作, 促进软件解决方案, 知识和专业技能的共享与再利用
	2021/7	供应链攻击威胁全景图[35]	该全景图描述并研究了供应链攻击的活动, 强调应特别关注供应链攻击, 并将供应商纳入防护和安全验证机制中
	2022/9	网络弹性法案[22]	该法案规定所有在欧盟市场销售的设备和软件必须符合网络安全标准
加拿大	2023/2	保护您的组织免受软件供应链威胁[36]	该文件强调了降低软件供应链风险的重要性, 并建议审查供应商, 持续监控风险, 将供应链风险管理纳入安全计划

随着 SBOM 的提出, 许多研究人员开始关注到 SBOM. 虽然 NTIA 提供的资料从 SBOM 的基本概念、创建、实施、应用案例等方面进行了详细阐述, 但为进一步了解 SBOM 的发展, 本文在 IEEE Xplore 和 ACM Digital Library 中围绕 SBOM 这一领域进行检索, 筛选了 30 篇紧密相关的文献.

在研究 SBOM 的应用方面, Jaatun 等人[37]指出 SBOM 有助于识别和管理潜在的隐患, 减少针对软件供应链的威胁. Nocera 等人[38]发现 SBOM 在开源项目中的采用率不高但呈上升趋势, 并且 SBOM 工具的可用性和功能仍存在不足. 文献 [39–42] 发现, 在采购软件或设备时集成 SBOM 增强了软件或设备的透明度, 提供了软件或设备的安全评估. 文献 [43–45] 发现, 在容器中集成 SBOM 有助于管理容器安全. Foster 等人[46]研究了如何将 SBOM 与网络威胁分析集成, 增强分析和防御攻击的能力. Hyeon 等人[47]指出在区块链技术中集成 SBOM 能有效防止恶意软件注入等安全风险. Wu 等人[48]认为在系统中集成 SBOM 能检测并定位存在漏洞的组件. Crawford 等人[49]发现借助 Linux 内核强制采用 SBOM 有助于确保软件的完整性和安全性. Sun 等人[50]实现了一种集成 SBOM 追踪漏洞相关组件的方法. O’Donoghue 等人[51]使用开源工具 Trivy 和 Grype 分析 SBOM 探索软件漏洞的存在情况. Xia 等人[52]扩展了 SBOM 的应用范围, 提出 AI 材料清单的概念, 以期更好地管理 AI 系统. Kishimoto 等人[53]实现了一

个集成 SBOM 管理软件的工具 Osmy, 自动评估安全风险和文件的完整性.

在 SBOM 的现状和发展方面, Kloeg 等人[54]发现集成商和软件供应商采用 SBOM 的意向较高, 而客户和个体开发者的采用 SBOM 的意愿较低. Xia 等人[55]探究了从业者对 SBOM 的看法. Chaora 等人[56]指出将 SBOM 纳入法规的重要性. Zahan 等人[57]探究了采用 SBOM 的好处以及面对挑战. BI 等人[58]和 Stalnaker 等人[59]探讨了采用 SBOM 的过程中遇到的实际挑战, 并提出了多种解决方案.

在评估 SBOM 工具的质量方面, Torres-Arias 等人[60]分析了 SBOM 的质量问题, 指出不同的 SBOM 生成工具的质量参差不齐. Balliu 等人[61]探讨了 6 个 SBOM 生成工具的局限性, 发现 SBOM 生成工具之间存在显著差异, 当前 SBOM 的生成仍面临技术和标准上的挑战. Bifolco 等人[62]指出 GitHub 依赖图的不精确性可能会影响基于依赖图生成的 SBOM 的有效性. Rabbi 等人[63]探讨了 4 个 SBOM 生成工具的表现, 发现 CNN 较为稳定高效. Mirakhorli 等人[64]探讨了 SBOM 相关开源和专有工具, 指出 SBOM 生成工具最多, 大多数工具使用特定的 SBOM 格式. Halbritter 等人[65]探究了 8 个 SBOM 工具的准确性和可靠性, 发现工具均不准确可靠. Dalia 等人[66]对生成 SBOM 的现有工具进行了比较分析, 发现现有技术尚未成熟, 且未完全自动化.

3 SBOM 的基本概念

NTIA 建立的信息模型 SBOM 详细记录了软件的组件信息和组件关系信息. 其中, 信息涵盖了组件和组件关系的信息, 即 SBOM 的基本元素, 模型则涉及呈现数据的结构和规范, 即 SBOM 的格式.

3.1 SBOM 关键元素定义

NTIA 在发布的文档[19,20,67]中定义了一系列基本元素, 描述了软件 SBOM 的元信息和软件组件的元信息. 表 2 整理了 NTIA 定义的一系列基本元素.

表 2 SBOM 的基本元素

类别	元素		描述	说明
SBOM的元信息	作者		创建软件SBOM的实体	创建SBOM的作者可以是软件供应商或其他个人或组织
	时间戳		生成SBOM的时间或最新的更新时间	时间戳应采用国际通用的格式
组件的元信息	供应商名称		创建组件的个人或组织	需考虑供应商存在别名的情况
	组件名称		组件的名称	需考虑组件存在多个别名的情况
	版本号		组件的版本	如果组件供应商未提供组件版本信息, 则应在该部分创建一个版本信息
	其他唯一标识符	组件哈希	组件的加密散列	加密哈希是最准确的唯一标识组件的标识符
		唯一标识符	帮助唯一标识组件的附加信息	可以使用全局唯一标识符来创建唯一标识符
	关系		组件之间的关系	默认为包含关系, 表示下游组件对上游组件的包含或依赖

NTIA 将 SBOM 的基本元素分为描述 SBOM 元信息的元素和描述组件元信息的元素. 其中, SBOM 的元信息指的是描述和标识整个 SBOM 文档的基本信息, 该信息同时也标识了被描述的软件. 组件的元信息指的是描述和标识软件组件的详细信息.

SBOM 的元信息包括作者和时间戳. 作者指的是创建软件 SBOM 的实体名称, 可以是个人或组织, 标识是谁创建了这个 SBOM. 当作者不是软件供应商时, 说明软件的 SBOM 是由其他个人或组织创建的. 时间戳指的是生成 SBOM 的时间或最新的更新时间. 时间戳应采用国际通用格式, 确保跨时区的一致性.

组件的元信息包括供应商名称、组件名称、版本号、组件哈希、唯一标识符和关系. 供应商名称指的是创建组件的个人或组织, 需考虑供应商存在多个别名的情况. 组件名称描述了组件的名称, 同样需考虑组件存在多个别名的情况, 可以使用“命名空间:名称”的格式来表示组件名称, 其中, “命名空间”指代供应商名称, “名称”指代“组件名称”. 版本号描述了组件的版本信息. 如果组件供应商未提供版本信息, 则应在该部分创建一个版本信息, 以便标识该组件.

为提供多层次的标识机制, SBOM 还通过组件哈希和唯一标识符来标识组件. 组件哈希提供了组件的加密散列信息. 虽然还可以使用加密签名来代替加密哈希, 但会增加额外的开销. 组件的哈希值可用于验证组件的完整性, 防止组件在传播过程中被恶意篡改. 此外, 还可以在不同层级上生成哈希值, 例如为组件的源码和编译后的二进制文件分别生成哈希值, 在多个层级上验证组件完整性. 同时也可以为软件的组件集合生成一个总体哈希值. 尽管组件哈希值能唯一标识每个组件, 但它仅作为 SBOM 中的可选项存在. 唯一标识符是帮助唯一标识组件的附加信息, 可以使用全局唯一标识符[19]为组件创建唯一标识符. 全局唯一标识符在全局范围内是唯一的, 但可能出现不同的 SBOM 作者对同一组件使用不同的标识符, 或者不同的 SBOM 作者对不同的组件使用相同的标识符的情况, 理想情况下, 唯一标识符在全球是唯一的, 这也正是 SBOM 的发展愿景[68]. 常见的全局唯一标识符包括 CPE (common platform enumeration), PURL (package URL) 和 SWHID (software heritage ID), UUID (universal unique identifier).

关系描述了组件之间的关系, 默认为包含关系, 表示下游组件包含或依赖上游组件, 也可反向描述为被包含关系, 表示上游组件被下游组件包含或依赖. 选择何种描述方式都是允许的, 重点在于全局需使用相同的描述方式. 现有的 SBOM 格式支持多种关系类型, 但都是对包含关系的进一步细化, 主要涵盖: ① 对未更改的二进制组件的直接引用; ② 对未更改的源码组件的链接或编译; ③ 对派生的源码组件的链接或编译.

理想情况下, 软件的 SBOM 信息, 软件的组件以及这些组件的 SBOM 信息, 应该由各自的供应商提供. 然而, 实际上并不能保证每个供应商都会主动提供相关信息. 此时, 需要其他个人或组织根据能获得的信息创建 SBOM, 但无法确保信息是完整且准确的. 因此, NTIA 建议将缺失的属性值设置为“无断言”或“无值”. 此外, NTIA 还建议在 SBOM 的关系元素中增加关系断言. 关系断言主要涵盖 4 种情况: ① 未知, 默认情况, 表示无法确定组件是否依赖于其他组件; ② 没有依赖的组件, 表示该组件不依赖任何其他组件; ③ 列出部分依赖组件, 表示该组件依赖于一个或多个组件, 但只列出了部分已知的依赖组件; ④ 列出所有依赖组件, 表示该组件依赖于一个或多个组件, 且已列出所有依赖的组件.

为支持不同的用例, 除了基本元素之外, NTIA 建议增加额外的属性. 这些属性包括但不限于: ① 组件的终止生命周期或终止支持日期; ② 组件的技术范围; ③ 组件的分组. 考虑到 SBOM 信息的加密和验证, NTIA 还建议设置数字签名和公钥相关的属性.

3.2 SBOM 标准化格式

NTIA 发布的文档[17,20,69]中概述了 3 种 SBOM 格式, 包括 SPDX (software package data exchange), CycloneDX 和 SWID (software identification) tags.

3.2.1 SPDX 格式

SPDX[70]由 Linux 基金会提出, 于 2021 年被批准为 ISO/IEC 5962:2021 标准. 目前, SPDX 的最新版本为 2.3 版本[71]. 表 3 展示了此版本包含的元素及相应的解释说明.

表 3 SPDX 包含的元素

元素	描述	说明
创建信息	向前和向后兼容所需的信息	每个SPDX文档都要提供, 必要内容
包信息	软件包的元信息	包括但不限于: 产品, 容器, 组件等可以由一个或多个文件组成, 还可以包含子包
文件信息	软件包中每个文件的元信息	包括许可证和版权信息. 特有字段
片段信息	文件中从其他来源复制的内容	复制的片段可能存在安全漏洞或违反许可证合规性. 特有字段
其他许可信息	未包含在SPDX许可证列表中的许可证信息	若许可证不在SPDX许可证列表中, 则必须使用该字段. 特有字段
关系信息	软件包, 文件和片段之间的关系	支持45种关系, 可以根据具体情况进行扩展
注释信息	关于软件包, 文件和片段的注释信息	对软件包, 文件和片段的进一步解释
审核信息	不再建议使用该字段, 建议将内容整合至注释信息中	在SPDX 1.2中兼容, 但自SPDX 2.0起被弃用

表 3 列出了 SPDX 2.3 版本包含的字段信息. 其中, 创建信息提供了版本兼容所需的信息, 是每个 SPDX 文档都必须提供的, 其他字段可根据创建者的特定需求选择使用. 包信息和关系信息是 NTIA 规定的对组件和组件关系的描述. 本文认为不在 NTIA 要求的基本元素范围内且其他 SBOM 格式均未提供的字段为格式的特有字段. 对 SPDX 来说, 文件信息, 片段信息和其他许可信息是 SPDX 的特有字段. 虽然注释信息和审核信息 NTIA 并未要求, 但 3 种 SBOM 格式均提供了与注释相关的字段, 所以并不将该字段视为特有字段.

假设一个具体实例场景, 由个人 Ana 开发的软件 A 1.1 版本依赖于由个人 Bob 开发的软件 B 1.1 版本, 即软件 B 1.1 版本是软件 A 1.1 版本的组件. 表 4 展示了软件 A 的 SPDX 格式的 SBOM.

表 4　SPDX 格式的 SBOM 示例

1.	SPDXVersion: SPDX-2.3
2.	DataLicense: CC0-1.0
3.	DocumentNamespace: http://www.spdx.org/spdxdocs/8f141b09-1138-4fc5-aefb-fc10d9ac1eed
4.	DocumentName: SBOM example
5.	SPDXID: SPDXRef-DOCUMENT
6.	Creator: Person: Ana
7.	Created: 2024-06-30T11:40:49Z
8.	Relationship: SPDXRef-DOCUMENT DESCRIBES SPDXRef-A-v1.1
9.	
10.	PackageName: A
11.	SPDXID: SPDXRef-A-v1.1
12.	PackageVersion: 1.1
13.	PackageSupplier: Person: Ana
14.	PackageDownloadLocation: NOASSERTION
15.	FilesAnalyzed: false
16.	PackageChecksum: SHA1: 75068c26abbed3ad3980685bae21d7202d288317
17.	PackageLicenseConcluded: Apache-2.0
18.	PackageLicenseDeclared: NOASSERTION
19.	PackageCopyrightText: Copyright 2024 Ana
20.	Relationship: SPDXRef-A-v1.1 CONTAINS SPDXRef-B-v1.1
21.	
22.	PackageName: B
23.	SPDXID: SPDXRef-B-v1.1
24.	PackageVersion: 1.1
25.	PackageSupplier: Person: Bob
26.	PackageDownloadLocation: NOASSERTION
27.	FilesAnalyzed: false
28.	PackageChecksum: SHA1: 94568c26abbed3ad3980685deaf1d7202d268314
29.	PackageLicenseConcluded: Apache-2.0
30.	PackageLicenseDeclared: NOASSERTION
31.	PackageCopyrightText: Copyright 2024 Bob
32.	Relationship: SPDXRef-B-v2.1 CONTAINS NOASSERTION

表 4 中, 第 1–8 行为创建信息, 其中, SPDXVersion 指示 SPDX 的版本, DataLicense 指示数据许可. Document Namespace 指示了当前 SBOM 采用的命名空间. DocumentName 指示了 SBOM 文档的名称. SPDXID 指示了 SBOM 文档的 SPDX 标识符. Creator 指示了 SBOM 的创建者. Created 指示了 SBOM 的创建时间. Relationship 指示了文档描述的关系, 表示本文档描述了软件 A 的 1.1 版本. 第 10–20 行以及第 22–32 行是对各个软件的描述, 即包信息, 其中, PackageName 指示了软件的名称. SPDXID 指示了软件的 SPDX 标识符. PackageVersion 指示了软件的版本号. PackageSupplier 指示了软件的供应商. PackageDownloadLocation 指示了软件的下载位置, 表示当前未提供下载位置. FilesAnalyzed 指示是否分析了软件中的文件. PackageChecksum 指示了软件的校验和. PackageLicenseConcluded 指示了软件确实使用的许可证, 标记经审查后确实使用的许可证. PackageLicense Declared 指示

软件是否声明了许可证. PackageCopyrightText 指示了软件的版权. Relationship 指示了软件的依赖关系, 表示软件 A 依赖软件 B, 但对于软件 B 来说, 当前不明确软件 B 的依赖关系.

3.2.2 CycloneDX 格式

CycloneDX[72]由 OWASP 组织支持和维护, 是一个轻量级的 SBOM 格式. 目前, CycloneDX 的最新版本为 1.5 版本[73]. 表 5 展示了此版本包含的元素及相应的解释说明.

表 5 CycloneDX 的内容

元素	描述	说明
BOM元数据	BOM的元信息	实体的供应商, 制造商和实体名, 以及生成BOM的工具和BOM文档使用的许可证. 其中BOM的格式和版本是必要内容
组件	组件的重要元信息	实体使用的第一方和第三方组件的详细信息, 包括组件的来源, 许可证等信息
服务	实体可能调用的外部API	实体可能调用的外部API, 以及实体与其调用的外部API之间的数据流. 特有字段
依赖	组件之间的依赖关系	描述组件与组件, 组件与服务以及服务与服务之间的依赖关系, 包括直接依赖关系和间接依赖关系
组合	依赖关系的完整性	组件依赖关系的完整性, 包括完整的, 不完整的, 不完整的第一方组件, 不完整的第三方组件和未知
漏洞	公开披露的漏洞和漏洞的可利用性	还可描述未公开披露但对实体产生影响的漏洞. 特有字段
配方	制造和部署的过程	描述实体或实体中某元素的制造流程以及操作等信息. 特有字段
注释	为被注释对象提供额外信息	可对注释使用数字签名, 以便对注释内容进行验证
扩展	新功能	可在该部分对新功能进行快速的原型设计. 特有字段

表 5 展示了 1.5 版本的 CycloneDX 要求的字段信息. BOM 元数据描述了被创建的 SBOM, 其中的 BOM 格式和版本是必须提供的内容. 组件, 依赖和组合字段提供了 NTIA 要求的基本元素, 而服务, 漏洞, 配方和扩展字段是 CycloneDX 的特有字段.

同样以第 3.2.1 节中的假设为例, 表 6 展示了软件 A 的 CycloneDX 格式的 SBOM.

表 6 CycloneDX 格式的 SBOM 示例

```
<?xml version="1.0" encoding="utf-8"?>
<bom xmlns="http://cyclonedx.org/schema/bom/1.5"
    serialNumber="urn:uuid:3e671687-395b-41f5-a30f-a58921a69b71" version="1"
    <metadata>
        <authors><author><name>Ana</name></author></authors>
        <component type="application">
            <name>A</name>
            <version>1.1</version>
            <hashes><hash alg="SHA-1">75068c26abbed3ad3980685bae21d7202d288317</hash></hashes>
            <cpe>cpe:2.3:a:ana:a:1.1:*:*:*:*:*:*:*</cpe>
            <externalReferences/>
            <components/>
        </component>
        <manufacture><name>Ana</name></manufacture>
        <supplier><name>Ana</name></supplier>
    </metadata>
    <components>
        <component type="library" bom-ref="pkg:Bob/B@1.1">
            <publisher>Bob</publisher>
            <group>Bob</group>
            <name>B</name>
            <version>1.1</version>
            <hashes><hash alg="SHA-1">94568c26abbed3ad3980685deaf1d7202d268314</hash></hashes>
```

表 6 CycloneDX 格式的 SBOM 示例 (续)

```
24.             <cpe>cpe:2.3:a:bob:b:1.1:*:*:*:*:*:*:*</cpe>
25.             <purl>pkg:Bob/B@1.1</purl>
26.           </component>
27.         </components>
28.         <dependencies>
29.           <dependency ref="pkg:Bob/B@1.1"/>
30.         </dependencies>
31.         <compositions>
32.           <composition>
33.             <aggregate>unknown</aggregate>
34.             <assemblies><assembly ref="pkg:Bob/B@1.1"/></assemblies>
35.           </composition>
36.         </compositions>
37.       </bom>
```

表 6 中, 第 1–16 行为 BOM 的元数据, 其中, <?xml version=1.0 encoding=utf-8?>声明了文档遵循 XML 1.0 标准, 使用 UTF-8 编码格式. <bom xmlns=http://cyclonedx.org/schema/bom/1.5 定义了 XML 文档使用的结构, 其中的 1.5 声明了采用的 CycloneDX 格式为 1.5 版本. serialNumber 指示了 SBOM 的唯一标识符, version 指示了 SBOM 的版本. 第 4–16 行指示了 SBOM 的作者, 被描述的软件 A 的名称, 版本, 哈希值, CPE 值, 供应商等信息. 第 17–27 行描述了软件依赖的组件, 即软件 B 的元信息, 包括名称, 版本, 哈希值, CPE 值, PURL 等信息. 第 28–30 行描述了软件 A 的依赖信息, 即软件 A 依赖于软件 B. 第 31–36 行提供了关系断言, 表示软件 B 的依赖关系是未知的.

3.2.3 SWID 格式

SWID[74]由 ISO 和 IEC 发布, 被批准为 ISO/IEC19770-2:2015 标准. 目前, SWID 格式的最新版为 2016 年版[75]. 表 7 展示了该版本中用于描述软件的标签.

表 7 SWID 的内容

元素	描述	说明
主标签	标识和描述成功安装的软件	必须提供软件的名称, 标签的全局唯一标识符和创建者. 标签的全局唯一标识符与已安装的软件一一对应, 当更新或删除软件时, 软件的主标签也要随之一同被更新或删除
补丁标签	标识和描述软件的补丁	补丁标签描述修复软件的信息, 补丁与补丁, 补丁与软件之间的关系, 包括: ① 适用于被修补的软件; ② 补丁的补丁, 在应用补丁之前需要先安装另一个补丁; ③ 替换另一个补丁, 即替换旧补丁. 特有字段
语料库标签	描述处于预安装状态的软件	语料库标签主要用于在软件安装前验证软件的完整性和软件发布者的身份, 确定是否拥有该软件的有效许可证, 以便在不合适情况下及时终止软件的安装. 特有字段
补充标签	补充软件安装后的信息	任何实体都可以在安装主标签的同时或之后的任意时刻安装任意数量的补充标签. 补充标签可以对任何标签进行补充. 当补充标签与其他标签发生冲突时, 应以主标签中的内容为准, 并报告冲突异常

SWID 主标签中的软件名称, 标签的全局唯一标识符和创建者是必须提供的信息. 此外, SWID 通过在标签中使用<Link>字段描述标签之间的关系. 考虑到软件可能作为组件供其他软件使用, SWID 使用<Link>链接软件的主标签来描述软件之间的关系. 此外, SWID 还使用<Link>链接补丁标签与补丁标签, 补丁标签与软件的主标签, 以及补充标签和被补充的标签来描述标签之间的关系. 其中, 补丁标签和语料库标签是 SWID 的特有字段.

同样以第 3.2.1 节中的假设为例, 表 8 展示了软件 A 的 SWID 格式的 SBOM.

表 8 中, 第 1–10 行描述了软件 A 的主标签, 其中<SoftwareIdentity>指示了 SWID 标签的开始. xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"指示了 XML 的命名空间, 表示关联到 ISO/IEC 19770-2:2015 标准, xmlns:sha512="http://www.w3.org/2001/04/xmlenc#sha512"声明使用的是 SHA-512 哈希算法. name 指示了软件的名称, tagId 指示了软件的唯一标识符. version 指示了软件的版本信息. <Entity>指示了描述的实体, 这里表示

Ana 是标签的创建者和软件的创建者. <Link>指示了链接之间的关系, 表示软件 A 依赖于标识为 Bob/B@1.1 的软件. <Payload>指示了软件的具体信息. <File>列出了文件名和哈希值.

表 8 SWID 格式的 SBOM 示例

```
<SoftwareIdentity
    xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
    xmlns:sha512="http://www.w3.org/2001/04/xmlenc#sha512"
    name="A"
    tagId="Ana/A@1.1"
    version="1.1">
  <Entity name="Ana" role="tagCreator softwareCreator"/>
  <Link href="swid:Bob/B@1.1" rel="component"/>
  <Payload><File name="Ana-A-1.1.exe" sha512:hash="BC55DEF84538898754536AE47CC907387B8F61D9ACD7D3FB8B
  8A624199682C8FBE 6D1631088AE6A322CDDC4252D3564655CB234D3818962B0B75C35504D55689"/>
  </Payload>
</SoftwareIdentity>

<SoftwareIdentity
    xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"
    xmlns:sha512="http://www.w3.org/2001/04/xmlenc#sha512"
    name="B"
    tagId="Bob/B@1.1"
    version="1.1">
    <Entity name="Bob" role="tagCreator softwareCreator"/>
    <Payload><File name="bob-browser-2.1.exe"
    sha512:hash="FF4893471E763B94165CC277A9FB01D7ED66256FDDD6467D9
    1E35AFF8F445C6312 832FD97DE1FD517606019BDC5F46E9E4E4814601E1FCB1010E90C2EBE54820"/></Payload>
</SoftwareIdentity>
```

3.2.4 SBOM 格式比较

由于 SWID 更像是软件标识符而非 SBOM, 所以 SWID 更常作为软件标识符使用[64], 当前较为常用的 SBOM 格式是 SPDX 和 CycloneDX 格式. Stalnaker 等人[59]采访的 50 位被试者中只有 5 位使用了 SWID, 大多数被试者使用的是 SPDX 或者同时使用 SPDX 和 CycloneDX. Xia 等人[55]也在访谈和调查中发现 SPDX 和 CycloneDX 格式最为常用. 除了使用的广泛程度之外, 表 9 还从侧重点, 适用场景和优缺点这几个方面比较了这 3 种 SBOM 格式.

表 9 SBOM 格式比较

格式	侧重点	适用场景	优点	缺点
SPDX	侧重于描述软件的许可证信息	适用于专注记录软件许可信息, 审查合规性的场景	有助于对软件进行合规性审查, 记录信息的粒度更细, 到片段级	未重点关注到安全性问题. 由于分析的粒度较细, 所以需要的信息资源较多
CycloneDX	侧重于软件的安全性分析和漏洞管理	适用于专注分析安全性和管理漏洞的场景	灵活且轻量级, 易于生成和解析	侧重于安全性分析, 未重点关注到合规性问题
SWID	侧重于识别和跟踪软件的管理和部署	适用于专注识别和跟踪软件的场景	标准化的格式标签, 便于组织管理	侧重于软件的管理, 在合规性分析和安全性分析方面的支持较弱

SPDX, CycloneDX 和 SWID 这 3 种 SBOM 格式各有侧重和优缺点. SPDX 重点关注软件的许可证和版权信息, 适用于需要详细审查合规性的场景. 由于 SPDX 提供信息的粒度为片段级, 所以生成和维护 SPDX 所需的信息资源较多. 由于 SPDX 未设置相关字段记录漏洞信息, 所以不适合重点分析安全性的场景. CycloneDX 侧重于分析软件的安全性和管理漏洞, 适用于重点分析安全和管理风险的场景. CycloneDX 提供信息的粒度为组件级, 并且 CycloneDX 集成了 SPDX 的许可证信息, 无需记录大量的许可证和合规性信息, 所以 CycloneDX 更加灵活, 轻量

级, 易于生成和解析, 但对许可证和合规性信息的支持不如 SPDX. SWID 适用于管理和跟踪软件的场景, 在合规性和安全性分析方面的支持较弱. 选择何种的 SBOM 格式需根据具体的需求和使用场景决定, 也可同时使用多种 SBOM 格式以更好地满足组织的合规性, 安全性和管理需求.

4 SBOM 生成技术

由于 SBOM 相关文献聚焦于 SBOM 的应用, 工具以及现状, 为深入研究 SBOM 的生成技术, 本文从 NTIA 提供的与 SBOM 生成相关的资料[17,69,76−78]中梳理出 SBOM 生成涉及的领域, 在 IEEE Xplore 和 ACM Digital Library 中进行检索, 筛选了 61 篇紧密相关的文献.

根据面向的对象, 分析的粒度, 本文对 SBOM 的生成技术进行了分类. 图 2 展示了本文对 SBOM 生成技术的分类. 根据面向的对象分类, SBOM 生成技术细分为源码级 SBOM 生成技术和二进制级 SBOM 生成技术. 源码级 SBOM 生成技术面向的对象是以源码形式发布的软件, 而二进制级 SBOM 生成技术面向的对象是以二进制形式发布的软件. 根据分析的粒度分类, 源码级和二进制级 SBOM 生成技术均细分为面向组件依赖的 SBOM 生成技术和面向开源片段引用依赖的 SBOM 生成技术. 面向组件依赖的 SBOM 生成技术分析的是软件复用了哪些软件作为组件, 而面向开源片段引用依赖的 SBOM 生成技术分析的是软件复用了哪些代码片段, 比面向组件的技术分析的粒度更细. 对于面向组件依赖的 SBOM 生成技术又可进一步分类. 其中, 源码级软件面向组件依赖的 SBOM 生成技术根据分析的对象分为基于组件依赖清单文件的 SBOM 生成技术和基于代码文件依赖关系分析的 SBOM 生成技术. 二进制级软件面向组件依赖的 SBOM 生成技术根据分析采用的方法分为基于静态分析依赖关系的 SBOM 生成技术和基于动态分析依赖关系的 SBOM 生成技术.

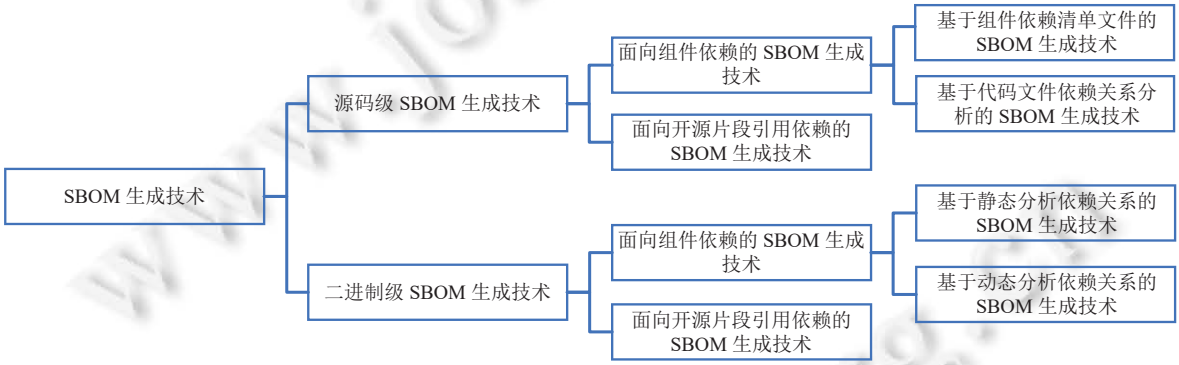

图 2 SBOM 生成技术分类

NTIA 将 SBOM 的生成分为 4 个步骤[78]: ①识别软件包含的组件; ②获取组件的数据; ③将组件数据规范化为 SBOM 格式; ④检查 SBOM 格式是否有效, 是否包括基本属性. 特别地, NTIA 强调 SBOM 格式检查的目的是检查必须字段是否存在, 格式是否结构化, 而不是检查 SBOM 内容的准确性[78]. 此外, 被复用的现有软件除了被称为组件之外, 还被称为第三方包、 第三方库或三方库. 后续将基于 SBOM 的生成步骤分析如何利用各个技术生成 SBOM.

4.1 源码级 SBOM 生成技术

源码级 SBOM 生成技术面向的对象是以源码形式发布的软件. 以源码形式发布的软件保留了软件具体的实现方式和内部逻辑, 因此为源码级软件生成 SBOM 较为容易. 根据分析粒度进一步分类, 将源码级 SBOM 生成技术划分为: ① 面向组件依赖的 SBOM 生成技术; ② 面向开源片段引用依赖的 SBOM 生成技术.

4.1.1 面向组件依赖的 SBOM 生成技术

面向组件依赖的 SBOM 生成技术分析的粒度为组件级. 根据被分析的文件进一步分类, 面向组件依赖的 SBOM 生成技术可划分为: ① 基于组件依赖清单文件的 SBOM 生成技术; ② 基于代码文件依赖关系分析的 SBOM 生成技术.

(1) 基于组件依赖清单文件的 SBOM 生成技术

清单文件声明了软件的元信息, 例如软件的名称, 版本和依赖关系等, 在软件开发中扮演了关键角色. 开发者完成软件开发后, 将软件上传到相应的包存储库, 便于其他开发者复用. 包管理器负责管理包存储库中的软件, 充当开发者和包存储库之间的桥梁. 为方便其他开发者对软件的理解和包管理器对软件的管理, 开发者提供软件的清单文件, 描述软件的元信息. IEEE 发布的 2024 年编程语言排名表[79]中列举了多种编程语言, 表 10 展示了不同编程语言常用的包存储库, 包管理器和清单文件.

表 10 不同编程语言常用的包存储库, 包管理器和清单文件

语言	包存储库	包管理器	清单文件
Python	PyPI	pip	requirements.txt
Java	Maven	maven	pom.xml
C/C++	Conan	conan	conaninfo.txt
JavaScript/TypeScript	Npm	npm	package.json
C#	NuGet	nuget	packages.config
Go	Go	go mod	go.mod
PHP	Packagist	composer	composer.json
Ruby	RubyGems	gems	gemfile
Swift	Swift Package Index	Swift Package Manager	package.swift
Dart	Dart Pub	Pub	pubspec.yaml
Rust	crates.io	Cargo	cargo.toml
Perl	CPAN	CPAN	META.yml, META.json
Erlang/Elixir	Hex.pm	Rebar	rebar.config
Haskell	Hackage	Cabal	*.cabal
DLang	Dub package registry	Dub	dub.json, dub.sdl
Elm	Elm Packages	elm-package	elm.json

虽然表 10 列出了不同语言常用的清单文件, 但开发者可能会灵活使用不同方式声明清单文件. 例如, pandas[80]的清单文件名为 requirements-dev.txt; numpy[81]单独设置了 requirements 文件夹, 其中存放了 build_requirements.txt, test_requirements.txt 等清单文件, 说明了不同情况下软件的依赖信息. 开发者还可能用表达需求的其他英文命名清单文件. 除此之外, 一种语言的包存储库可能会有多个包管理器, 而不同的包管理器会分析不同的清单文件. 以 Python 为例, 除了官方推荐的 pip 之外, 还有 poetry 和 pipenv, 而 poetry 分析的清单文件是 pyproject.toml, pipenv 分析的清单文件是 Pipfile 和 Pipfile.lock, 开发者会根据包管理器的不同提供相应的清单文件. 总之, 考虑的情况越丰富, 得到的依赖信息越充足, 生成的 SBOM 也就越完善.

由于清单文件提供了软件的元信息, 所以可以直接借助包管理器解析软件的清单文件生成软件的 SBOM. 图 3 展示了基于组件依赖清单文件的 SBOM 生成技术框架. 首先, 分析软件的清单文件, 得到软件依赖的组件, 并通过包管理器查找组件的元数据, 然后按需要的格式, 例如 SPDX 或 CycloneDX, 对得到的信息进行整理, 得到软件的 SBOM, 并检查 SBOM 格式是否有效.

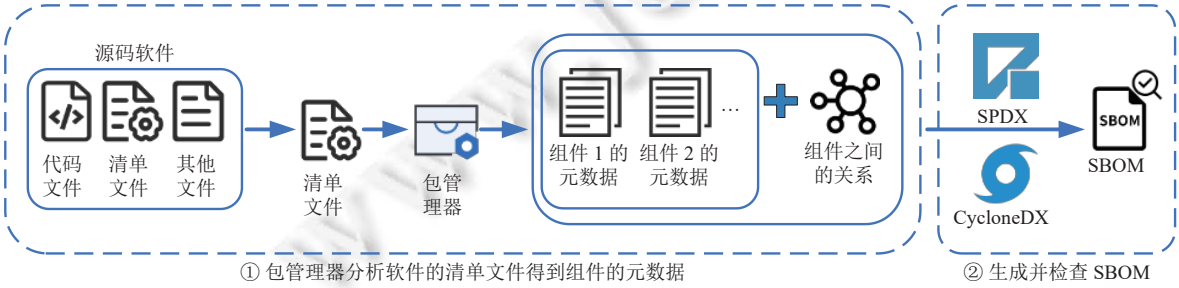

图 3 基于组件依赖清单文件的 SBOM 生成技术框架

每种编程语言的清单文件都有其独特的声明软件依赖组件的方式. 例如, 基于 Python 实现的软件使用 requirements.txt 作为清单文件, 其中记录了软件依赖的组件名称和版本. 基于 Java 实现的软件使用 pom.xml 作为清单文件, 其中不仅记录了软件的依赖信息, 还记录了其他与软件相关的信息. 该文件中的<groupId>标签记录了软件所属的组织名称, <artifactId>标签记录了软件在所属组织中的 ID, <version>标签记录了软件的版本, <dependencies>标签记录了软件的依赖信息. 基于 JavaScript 实现的软件使用 package.json 作为清单文件, 其中的 name 标签记录了软件的名称, version 标签记录了软件的版本, dependencies 标签记录了软件运行时的依赖信息, devDependencies 标签记录了软件开发时的依赖信息.

从清单文件获得组件名称和版本后, 可以借助包管理器获取指定版本组件的其他信息. 例如, Python 实现的软件可以通过包管理器 pip 提供的命令行指令 pip show package_name 查询组件的详细信息, 包括作者, 许可证, 依赖关系等, 其中 package_name 指代软件名. Java 的包管理器 maven 通过命令行指令 mvn dependency:tree 得到软件的依赖树. JavaScript 实现的软件通过包管理器 npm 提供的命令行指令 npm view <package_name>@<version_number>查询组件的详细信息, 其中 version_number 指代软件版本.

但上述操作适合表 10 中像 Python, Java 和 JavaScript 这种有成熟的包管理器的语言, 而 C/C++等语言并不依赖于包管理器. Tang 等人[82]研究总结了 C/C++项目的依赖模式, 发现目前 C/C++生态系统没有使用包管理器的惯例, 而且现有的包管理器无法有效地管理 C/C++项目的依赖关系. C/C++开发者还是更倾向于通过代码克隆实现对依赖项的引用. 代码克隆是指开发人员直接复用代码实现特定功能或解决特定问题的行为. 这种做法无法保障被复用代码的安全性还增加了维护成本.

清单文件和包管理器的组合使用为分析软件的组件提供了便利, 大多数生成 SBOM 的工具都优先选择这种方式获取软件的依赖信息. 特别地, NTIA 指出软件组件分析 (software composition analysis, SCA) 可用于识别软件的组件[76,78], 所以当前许多 SCA 工具也被称为 SBOM 工具. Zhao 等人[83]分析了 4 个开源 SCA 工具和 2 个商业工具 T1 和 T2, 发现这些工具都高度依赖清单文件和包管理器. Sharma 等人[84]也分析了 5 个 SCA 工具, 同样发现这些工具也借助包管理器和清单文件获取组件信息. Imtiaz 等人[85]筛选了 7 个开源 SCA 工具, 和 2 个商业 SCA 工具 Commercial A 和 Commercial B, 发现虽然可以通过分析清单文件, 源代码和二进制文件检测组件, 但工具更倾向于借助清单文件和包管理器分析项目的依赖关系, 9 个 SCA 工具中只有 Commercial B 通过分析软件的测试信息获取依赖信息. 此外, Halbritter 等人[65]筛选了 8 个 SBOM 工具, 发现 4 个工具声明组合使用清单文件和包管理器获取依赖信息, 而其他未声明的工具, 经实验验证, 同样使用包管理器分析清单文件得到软件的依赖信息.

清单文件和包管理器带来的便利让众多 SBOM 工具严重依赖这种方法. Mirakhorli 等人[64]对当前 84 个 SBOM 工具进行分析, 发现 SBOM 生成工具高度依赖清单文件和包管理器, 即使考虑分析代码文件的工具也同样会参考清单文件. 大多数 SBOM 生成工具在没有清单文件和包管理器的情况下, 无法生成详细的 SBOM, 并且这些工具通常全面接受清单文件中的内容, 而不追究准确性. 因此, 分析代码文件对 SBOM 的生成显得尤为重要.

(2) 基于代码文件依赖关系分析的 SBOM 生成技术

编程语言一般会在代码头部标明依赖的组件. 例如, Python 在代码头部以“import…”“from…import…”“import…as…”和“from…import…as…”的形式标明依赖的组件. JavaScript 通过“import…from…”引入依赖组件. Java 通过“import…”引入依赖组件. 虽然可以通过分析代码文件的导入信息获得软件依赖的组件, 但无法由此获得组件的版本信息. 即使可以将最新版视为组件的版本, 但组件的不同版本之间难免存在差异, 并且各个组件之间可能存在因版本导致的冲突. 许多研究人员对该问题进行研究, 发现可以通过构建和查询知识库获得软件组件的版本信息.

图 4 展示了基于代码文件依赖关系分析的 SBOM 生成技术的框架. 首先, 从包存储仓中获得不同软件各个版本的数据, 例如软件依赖的组件, 并由此构建一个知识库, 知识库中记录了精确到版本的软件间的依赖关系. 然后, 分析软件的代码文件获得软件的依赖信息. 接着, 根据依赖信息查询知识库, 得到详细的依赖关系. 最后, 按需要的格式整理得到的信息, 生成并检查 SBOM.

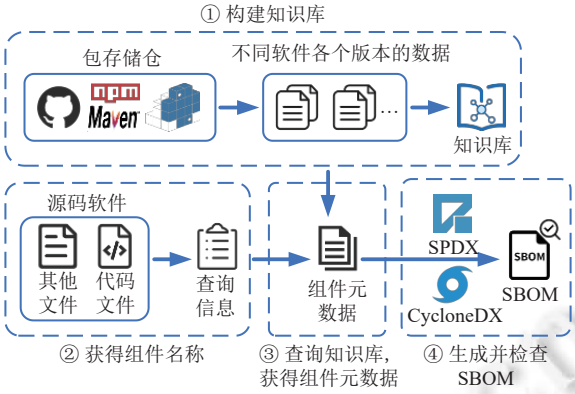

图 4 基于代码文件依赖关系分析的 SBOM 生成技术框架

Horton 等人[86]提出了 DockerizeMe. DockerizeMe 采用动静结合的技术构建知识库, 静态解析提取软件提供的模块, 依赖的组件; 动态安装并导入软件的模块, 记录可能缺少的组件, 由此构建知识图谱. DockerizeMe 为被分析代码构建抽象语法树 (abstract syntax code, AST), 提取代码导入的资源, 查询知识图谱, 将导入的资源映射到对应的软件, 进而识别代码依赖的组件及其版本, 以及组件间的关系. 后续, Horton 等人[87]还提出 V2. V2 在 DockerizeMe 的基础上进行了扩展, 通过与 DockerizeMe 提供的环境进行交互, 根据反馈调整依赖项的版本.

Woo 等人[88]提出了 CENTRIS. CENTRIS 收集精确到版本的 C/C++库, 提取库提供的函数, 存储为库签名, 这些签名构成了知识库. CENTRIS 进一步分析知识库中库之间的关系, 减少由库间复用带来的误报. CENTRIS 计算被分析软件中的函数与库签名之间的相似度, 超过阈值的库被视为软件的组件. 特别地, CENTRIS 辅助使用代码相似性检测技术得到组件级依赖, 而并非追溯代码片段的来源.

Ye 等人[89]提出了 DockerGen. DockerGen 抓取大量的 Dockerfile, 解析 Dockerfile, 提取 Docker 镜像, 操作系统, 软件包以及依赖关系, 构建知识图谱. 给定目标软件, DockerGen 在知识图谱中查找包含该软件的 Docker 镜像以及与该软件相关联的软件包, 并确定软件包的安装顺序, 为软件生成 Dockerfile.

Wang 等人[90]提出了 SnifferDog. SnifferDog 收集大量 Python 库及其不同版本, 从库中提取应用程序接口 (application programming interface, API), 构建库 API 映射的知识库. SnifferDog 为被分析的代码构建 AST, 提取代码使用的 API, 在库 API 映射知识库中查询提供这些 API 的库及其版本. 最终得到代码依赖的组件.

Ye 等人[91]提出了 PyEGo. PyEGo 分析 Python 库及其不同版本的清单文件, 获得库依赖的组件, Python 解释器以及系统库, 构建知识图谱 PyKG. 然后, PyEGo 为被分析的代码构建 AST, 提取代码导入的库及其模块, 查询 PyKG, 将这些模块映射到对应的库, 获取代码依赖的库及其版本, 以及库之间的关系.

Cheng 等人[92]提出了 PyCRE. PyCRE 从 PyPI 中收集 Python 包及其不同版本, 安装包, 并记录安装状态. 对于成功安装的包, 提取包提供的模块, 依赖关系, 由此构建知识图谱. PyCRE 构建 AST 提取被分析代码导入的包及其模块. 通过查询知识图谱, 将调用的模块映射到对应的包, 获得代码的依赖信息. 后续, Cheng 等人[93]又提出了 ReadPyE. ReadPyE 扩充了 PyCRE 的知识图谱, 增加了与包相关的信息. 特别地, ReadPyE 通过命名相似度解决无法找到对应包的问题, 此外 PyCRE 设置规则推断适合的组件版本, 而 ReadPyE 在推断的环境中迭代验证和调整, 直至找到无环境异常的组件版本.

Jiang 等人[94]提出了 TPLite. TPLite 借助工具 SourcererCC 检测函数可能来源的第三方库, 而后分析第三方库的头文件, 许可证并借助工具 CCScanner 确定函数真实的第三方库来源. 接着, TPLite 分析第三方库间的函数相似性构建第三方库间的依赖关系, 以确保依赖关系的准确性.

Na 等人[95]提出了 CNEPS. CNEPS 根据代码文件间的相关性将被分析软件的分为多个模块. 然后利用 CENTRIS 识别每个模块中使用的第三方库, 进而构建模块与第三方库之间的关系. 通过分析代码中的#include 识别模块间的关系. 结合模块与第三方库之间的关系和模块间的关系得到软件的依赖信息.

Wu 等人[96]提出了 OSSFP. OSSFP 收集 C/C++开源库, 为方法级代码片段生成哈希特征, 根据特征信息移除不同库之间和库的不同版本之间重复的方法, 保留核心函数作为库的特征信息, 得到特征信息数据库. 最后, 通过匹配被分析软件的方法级代码片段的特征和特征信息数据库中的特征, 识别出软件依赖的组件.

表 11 对比总结了上述方法. 上述方法依靠现有知识构建供查询的知识库, 从待分析项目中挖掘与依赖相关的信息, 查询知识库获得待分析项目更详细的依赖信息. 然而, 结果的准确性受到知识库的完备性和正确性以及方法挖掘信息能力的限制, 导致在查询知识库获得更详细的依赖信息后, 还要耗费更多的时间推敲依赖项的版本, 避免依赖项之间的冲突.

表 11 基于代码文件依赖关系分析的 SBOM 生成技术方法对比

方法	处理内容	优点和贡献	缺点和局限
DockerizeMe	利用元数据构建知识图谱, 使用代码声明的组件名进行搜索	分析第三方库的动态依赖, 知识库更全面, 考虑了依赖项的安装顺序	受知识库完整性的限制, 仅考虑第三方库的最新版本
V2		多次动态调整依赖项版本	无法处理依赖问题被其他问题掩盖的情况, 分析过程耗时
CENTRIS	利用函数构建知识库, 使用代码中的函数进行搜索	消除了库的不同版本间冗余的函数, 分析库之间的关系减少了误报	受知识库完整性的限制
DockerGen	利用Dockerfile构建知识图谱, 使用软件名进行搜索	简化创建Dockerfile的过程, 应用范围广, 不局限于特定语言	受知识库完整性的限制
SnifferDog	利用API构建知识库, 使用代码使用的API进行搜索	创建库API映射知识库	受知识库完整性的限制
PyEGo	利用元数据构建知识图谱, 使用代码声明的组件名进行搜索	丰富了知识库, 能够处理复杂的依赖关系	受知识库完整性的限制
PyCRE	利用元数据构建知识图谱, 使用代码声明的组件名进行搜索	丰富了知识库, 降低分析的复杂性	倾向于选择库的最新版本
ReadPyE	利用元数据构建知识图谱, 使用代码声明的组件名进行搜索	丰富了知识库, 通过命名相似度解决无法找到库的情况	受知识库完整性的限制
TPLite	利用函数构建知识库, 使用代码中的函数进行搜索	丰富了库信息, 有效减少误报和漏报	受限于工具SourcererCC CCScanner
CNEPS	利用函数构建知识库, 使用代码中的函数进行搜索	全面地识别代码依赖的第三方库以及代码内的依赖	受限于工具CENTRIS
OSSFP	利用函数构建知识库, 使用代码中的函数进行搜索	消除了库的不同版本间和库之间的冗余函数, 有效减少误报和漏报	受知识库完整性的限制

无论是面向清单文件还是面向代码文件, 依赖信息的粒度都是组件级. 为了进行细粒度的分析以及考虑代码克隆的情况, 进一步考虑通过分析源码的代码片段生成 SBOM.

4.1.2 面向开源片段引用依赖的 SBOM 生成技术

除了直接复用软件之外, 开发者还会直接复用代码片段协助软件开发. 虽然复用代码片段带来了便捷, 但无法避免的要面临多种风险, 包括版权侵权和许可证不兼容等法律风险, 以及涉及恶意代码和漏洞的安全风险. 为了规避这些风险, 细粒度分析软件复用的代码片段是必要的.

图 5 展示了面向开源片段引用依赖的 SBOM 生成技术的框架. 首先, 按照不同的粒度, 例如方法级或类级, 提取代码片段, 考虑到去除噪声信息的干扰, 进一步将代码片段转化成中间表示. 然后, 根据中间表示的形式, 使用不同的方法将中间表示转化为特征向量, 通过分析特征向量, 推断出代码片段间的相似度, 得到被分析软件代码片段的来源. 最后, 根据被克隆代码片段的来源信息得到软件细粒度的依赖信息, 生成并检查 SBOM.

研究人员对代码相似性检测技术进行了广泛的研究. Cheng 等人[97]总结了 2018 年之前检测源代码相似性的文献采用的技术, 按提取的代码表征进行细分, 未考虑到二进制代码的相似性. Sun 等人[98]综述了 2022 年之前检测代码相似性的文献采用的技术, 按软件是否公开源代码将检测技术细分为源代码相似性检测和二进制代码相似性检测, 只考虑了源码与源码间、 二进制与二进制间的相似性, 未考虑二进制与源代码的相似性分析.

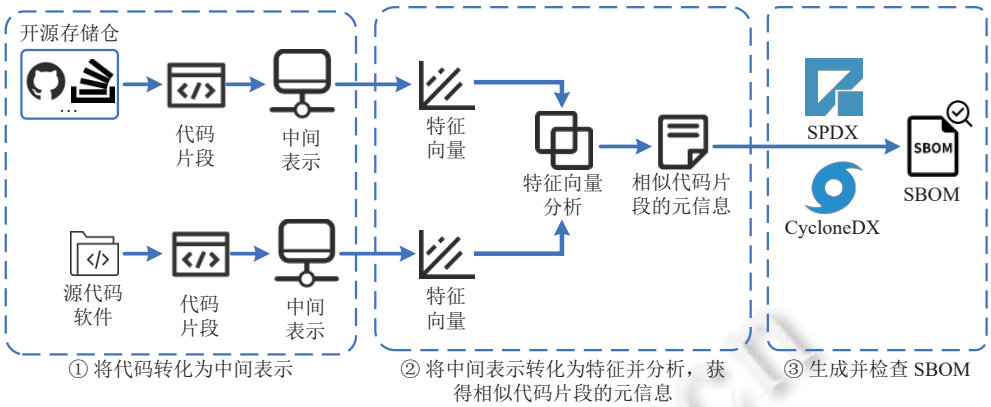

图 5 源码级面向开源片段引用依赖的 SBOM 生成技术框架

为使分类更加全面, 本文按待分析片段的类型对代码相似性检测技术进行细分, 具体分为源码级面向开源片段引用依赖的 SBOM 生成技术和二进制级面向开源片段引用依赖的 SBOM 生成技术. 前者面向的待分析片段的类型是源代码, 目的是为待分析的源代码片段溯源, 而后者面向的类型是二进制, 目的是为待分析的二进制代码片段溯源. 为避免重复分析, 本文重点关注 2023 年至今的代码相似性检测技术.

Wu 等人[99]提出了 Amain. Amain 为代码片段生成 AST, 将 AST 转换为基于马尔可夫链的状态矩阵作为代码片段的中间表示. 这种转换保留了详细信息, 降低了复杂性. 通过计算两个矩阵之间的距离, 得到特征向量, 分析特征向量, 判断代码对是否为克隆. 对于克隆代码对, 根据原代码片段的来源推出被分析代码片段的来源.

Hu 等人[100]提出了 Code2Img. Code2Img 基于代码的 AST, 计算代码片段的 N 行哈希值, 以 N 行哈希值作为键进行倒排索引定位可疑克隆. 对于可疑克隆, Code2Img 基于 AST 生成状态概率图, 将复杂的树结构转换为一维向量, 同时保留了 AST 的结构特征, 计算向量间的相似度来检测克隆.

Hu 等人[101]提出了 Tamer. Tamer 基于代码片段的 AST, 计算代码片段的 N-grams, 以 N-grams 的哈希值作为键进行倒排索引查找候选克隆对, 并使用过滤得分筛选候选克隆对. 对于筛选后的候选克隆对, 基于 AST 生成节点序列, 计算两个节点序列的相似度, 超过设定阈值的候选克隆对被认为是真正的克隆对.

Zou 等人[102]提出了 Tritor. Tritor 为代码片段生成 AST, 加入数据流和控制流信息增强 AST. Tritor 将增强后的 AST 划分为不同类型的三元组, 计算两个代码片段的三元组的相似性, 通过比较三元组的相似度判断代码片段的相似程度.

Mehrotra 等人[103]提出了 RUBHUS. RUBHUS 将代码片段解析成 AST, 加入控制流和数据流信息增强 AST. RUBHUS 利用图神经网络学习增强的 AST 的特征表示, 将特征输入分类器, 判断代码片段的相似性, 从而检测出代码克隆对.

Alhazami 等人[104]提出了 GoC. GoC 将代码片段解析为 AST, 以后序遍历的方式提取非终端节点, 识别节点之间的控制和数据依赖关系, 构建一个表示代码片段的有向图. 提取图的拓扑特征, 包括节点的加权度数, 特征向量等. 训练机器学习模型分析特征检测代码克隆.

Liu 等人[105]提出了 TAILOR. TAILOR 将源代码转换成结合了 AST, 控制流图 (control flow graph, CFG) 和数据流图 (data flow graph, DFG) 的代码属性图, 全面反映代码的语法和语义特征. TAILOR 训练图神经网络捕捉代码的图结构特征, 借助分类器分析图结构特征评估代码片段之间的相似性.

Wang 等人[106]提出了 TACC. TACC 将代码片段中相邻的代码行连接成 N 行片段, 并转换为词元序列. 使用两个过滤算法分别基于 N 行片段和词元序列的相似性过滤出候选克隆对. 接着提取候选克隆对的哈希树和特征向量, 匹配子树和分析特征向量, 综合分析结果报告检测到的克隆对.

Li 等人[107]提出了 ZC^3. ZC^3 训练大语言模型学习忽略语言之间的差异, 理解不同语言的代码片段之间的相似性. 通过这些步骤, 模型直接将不同语言的代码片段映射到同构表示空间, 并进行比较, 从而有效地检测出跨语言

的相似代码.

Shan 等人[108]提出了 Gitor. Gitor 对收集的代码片段进行词法分析, 提取关键信息, 构建全局样本图. 然后, Gitor 基于全局样本图应用节点嵌入技术将所有样本转化为对应的向量表示. 通过比较样本之间向量的相似性检测出可能的克隆对.

Xu 等人[109]提出了 DSFM. DSFM 将代码片段解析为 AST, 使用前序遍历将 AST 转换为子树序列. 对每个子树应用递归编码器和序列编码器, 得到子树的向量表示. 计算方法级和块级相似性, 结合方法级和块级的相似性, 获得代码的相似性.

Wu 等人[110]提出了 Goner. Goner 将代码转换为 AST, 并分割成多个子树. 将子树分类, 基于各种子树之间的相似度得到两个方法的相似性, 接着使用机器学习模型根据相似性判断代码对是否克隆.

Feng 等人[111]提出了 Toma. Toma 对代码进行词法分析, 提取标记序列. 通过计算相似性提取两个标记序列之间的相似性特征, 并使用训练好的机器学习模型根据相似性特征将代码对分类为克隆或非克隆.

Xu 等人[112]提出了 TCNAS. TCNAS 为代码生成 AST 和 DFG, 将代码结构转换为顺序数据以便模型处理. 基于优化的 Transformer 模型处理数据, 提取代码片段的特征向量. 通过比较代码片段的特征向量的相似度来检测代码克隆.

Li 等人[113]提出了 Prism 方法. Prism 将高层语言代码编译成多种架构的汇编代码, 并将汇编代码转换为嵌入表示. 同时, 利用 BERT 从高层语言代码中提取句法和语义特征, 将高层语言和低层语言的嵌入进行特征融合, 生成最终的代码表示, 并使用分类器对生成的代码表示进行分类, 检测代码对是否为克隆.

Zhang 等人[114]提出了 FSD-CLCD. FSD-CLCD 将代码解析为 AST, 添加控制流信息, 引入全局节点增强控制流之间的连接, 构建代码的图结构表示. 对图进行剪枝, 保留关键节点和全局节点的连接边, 并随机删除其余部分的边, 以减少图的规模和噪声的干扰. 在图学习阶段, 使具有相似语义的代码在特征空间中的分布更加一致, 从而实现跨语言代码克隆检测.

Du 等人[115]提出了 AdaCCD. AdaCCD 基于预训练的语言模型生成代码片段的嵌入表示, 实现捕捉代码的语义特征. 接着代码特征进行聚类, 发现潜在的语义相似和不相似的代码对.

Yuan 等人[116]提出的方法首先将 Java 源代码转换为中间代码, 并基于中间代码构建包含 AST, CFG 和 DFG 信息的中间代码图. 利用图嵌入技术提取中间代码图的语义信息, 同时应用 Code2Vec 技术提取源代码的语法信息, 将两种信息通过融合层合并为一个向量. 最后, 使用分类器检测代码克隆.

表 12 对比总结了上述方法. 上述方法的本质是从代码片段中提取表示代码的特征信息, 通过对比不同代码的特征信息, 判断代码是否相似. 不同方法之间的特性在于提取的代码特征信息不同. 通过提取不同种类的代码特征多角度捕捉代码的语法, 语义特征, 从而较好地检测出克隆代码对. 生成代码特征向量和度量相似度的方法逐渐从计算距离, 进行分类转换为多阶段多维度度量特征, 旨在降低资源消耗, 提高检测精度.

表 12　面向开源片段引用依赖的 SBOM 生成技术方法对比

方法	处理内容	优点和贡献	缺点和局限
Amain	基于AST生成马尔可夫链, 得到状态转移矩阵	将复杂的AST转换为简单的马尔可夫链模型, 提高了检测效率和速度	难以分析复杂的AST
Code2Img	基于AST的邻接矩阵生成邻接图, 然后编码为状态概率图	简化了AST的复杂结构, 同时保留了结构信息, 可扩展且适用于大规模克隆检测	难以分析复杂克隆类型, 资源消耗大
Tamer	基于AST的节点序列生成N-grams	将复杂的AST分割成相对简单的子树, 有效地减少检测成本	难以检测复杂的克隆类型, 检测时间较慢
Tritor	利用数据流和控制流增强AST, 将增强后的AST视为社会网络	增强了对代码结构和语义的捕捉能力, 准确性和可扩展性高	计算资源需求大
RUBHUS	利用控制流和数据流增强AST	结合了语法、语义和结构信息	图神经网络训练和部署难度大
GoC	基于AST提取控制和数据依赖关系, 构建有向图	提供了一种表示代码的方式, 在处理大规模输入时更具可扩展性且效率高	模型训练和推理的计算开销较大

表 12 面向开源片段引用依赖的 SBOM 生成技术方法对比 (续)

方法	处理内容	优点和贡献	缺点和局限
TAILOR	结合AST, CFG和DFG	有效整合多种代码表示	泛化能力差
TACC	N行片段的标记序列, AST和哈希树	两阶段组合分析, 提高了检测速度	参数设置复杂
ZC^3	映射代码片段到同构表示空间	零样本跨语言代码克隆检测	依赖于训练数据集
Gitor	全局样本图	结合关键词和边信息, 能捕捉代码复杂的关系和相似性	分析速度慢, 耗时
DSFM	基于AST生成子树序列	结合方法级和块级的相似性, 提高了检测的准确性	只在特定数据集上进行评估
Goner	基于AST生成子树序列, 统计各种子树出现的次数	简化了树结构的复杂性, 同时保留了丰富的语义信息	方法复杂且开销大
Toma	基于代码进行多分类	多角度保留代码特征	只在特定数据集上进行评估, 处理复杂情况的能力有限
TCNAS	基于AST和DFG转换得到顺序数据	有效捕捉了长期依赖关系, 集成了先进的神经架构搜索技术	方法设计复杂
Prism	将高层语言代码编译成多种架构的汇编代码	跨语言代码克隆检测, 融合高层次和低层次编程语言的语法和语义信息	依赖编译器, 需要较高的计算资源和时间成本
FSD-CLCD	利用关键控制流信息增强AST	有效捕捉了结构依赖和全局信息, 剪枝操作消除噪声干扰	重要边可能被剪枝, 导致信息损失
AdaCCD	语义保持的程序变换	仅使用源代码作为输入, 通用性高且易扩展	依赖聚类数量
文献[116]	基于中间代码构建中间代码图	基于中间代码的代码表示方法, 更好地捕捉源代码的语法和语义特征	依赖于编译器和Joern工具

4.2 二进制级 SBOM 生成技术

以二进制形式发布的软件可以直接在系统中运行, 虽然使用方便但将源码软件编译为二进制软件时会造成信息丢失. 因此, 为二进制软件生成 SBOM 比为源码软件生成 SBOM 更加复杂和困难. 根据分析粒度进一步分类, 二进制级软件的 SBOM 生成技术同样可划分为: ① 面向组件依赖的 SBOM 生成技术; ② 面向开源片段引用依赖的 SBOM 生成技术.

4.2.1 面向组件依赖的 SBOM 生成技术

根据分析方法进行分类, 面向组件的 SBOM 生成技术可进一步分为基于静态分析依赖关系的 SBOM 生成技术和基于动态分析依赖关系的 SBOM 生成技术.

4.2.1.1 基于静态分析依赖关系的 SBOM 生成技术

根据静态分析面向的对象不同, 基于静态分析依赖关系的 SBOM 生成技术可进一步划分为: ① 面向二进制应用软件的 SBOM 生成技术; ② 面向第三方库软件的 SBOM 生成技术. 二者的区别在于, 面向二进制应用软件的 SBOM 生成技术面向的对象是大量的二进制软件, 而面向第三方库软件的 SBOM 生成技术面向的对象是大量的第三方库.

图 6 展示了面向二进制应用软件的 SBOM 生成技术. 首先, 从应用存储仓中收集大量的二进制软件, 通过反编译工具得到软件的中间表示. 然后根据中间表示生成软件的特征, 对特征进行聚类, 由于多个软件可能复用了同一个组件, 所以属于同一个组件的特征会聚集在一起, 形成一个大型的集群, 因此聚类形成的大型集群被视为一个组件. 最后, 根据得到的组件信息生成并检查 SBOM.

F-Droid, Google Play, AppBrain, GitHub, SourceForge 等软件存储库提供了大量的二进制软件. Apktool, Baksmali, Androguard, Soot 等逆向工程工具实现软件的反编译, 根据反编译的结果, 生成软件的包结构, 图和类依赖关系等, 作为二进制软件的中间表示. 使用哈希算法处理软件的中间表示, 得到软件的特征向量, 进一步对特征向量进行聚类分析. 由于不同软件会复用同一个软件作为组件, 因此属于同一个软件的特征会聚集在一起形成一个大型集群. 分析聚类结果, 可获得待分析软件的组件信息.

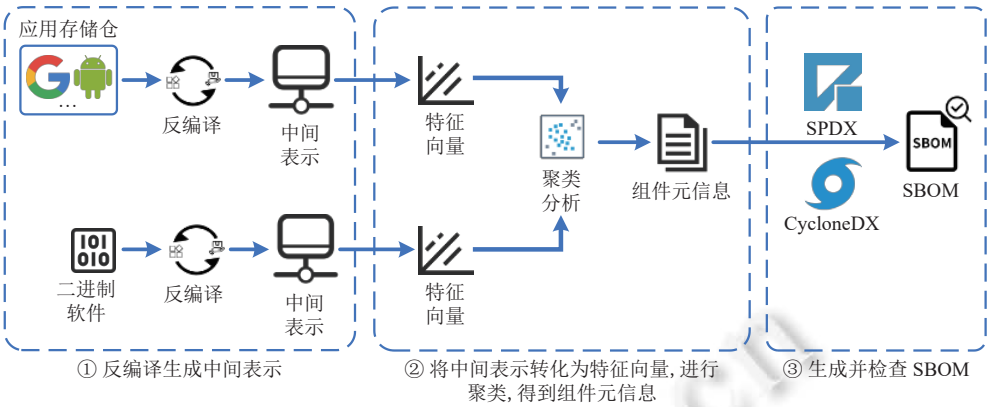

图 6 面向二进制应用软件的 SBOM 生成技术框架

由于代码混淆技术和编译导致的信息丢失带来的影响, 最终得到的组件信息并不非常准确. 代码混淆技术是指开发人员常常将代码转换为功能相同但难以理解的形式, 从而保护代码免受反编译破解. 即使没有采用代码混淆技术, 将代码编译为二进制时也会丢失信息. 因此最终获得的组件信息并不如分析源码得到的组件信息准确可靠. NTIA 在提出 SBOM 时也考虑了信息不准确的问题, 并要求允许 SBOM 存在错误.

图 7 展示了面向第三方库软件的 SBOM 生成技术. 首先, 从第三方库存储仓收集大量的第三方库, 理想情况下是收集二进制形式的第三方库, 以免影响检测结果. 同样使用反编译工具分析第三方库和目标二进制软件, 根据反编译的结果, 获得第三方库和目标二进制软件的中间表示. 接着, 根据中间表示, 得到第三方库和目标二进制软件的特征向量, 进一步使用相似性比较技术比较第三方库和目标二进制软件的特征, 相似度高的第三方库被视为软件的组件. 最后, 根据得到的组件信息生成并检查 SBOM.

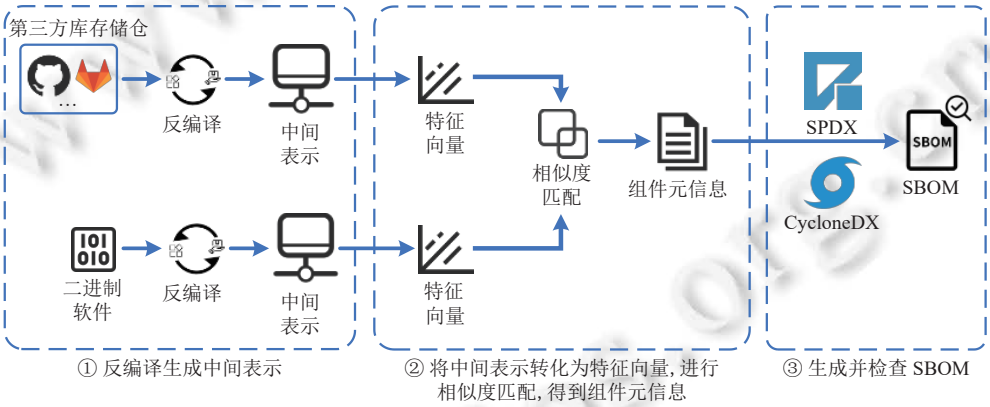

图 7 面向第三方库软件的 SBOM 生成技术框架

除了面向的对象不同, 面向二进制应用软件的 SBOM 生成技术与面向第三方库软件的 SBOM 生成技术在其他方面多有相似之处, 同样使用反编译工具, 根据反编译结果生成中间表示, 中间表示的形式也较为相似, 并且同样都用到相似性分析. 此外, 同样由于代码混淆技术和编译为二进制时的损耗, 最终获得的组件信息并不如源码分析准确可靠.

研究人员对检测二进制软件复用的第三方库进行了深入研究. Zhan 等人[117]总结了 2020 年之前相关文献采用的技术, 按分析技术细分为基于相似性的技术和基于聚类的技术. 两种技术的本质都是获得第三方库的特征, 都分析了特征的相似度. 本文认为这两种技术的本质区别是面向的对象不同, 基于相似性的技术面向的是大量的第三方库, 而基于聚类的技术面向的是大量的二进制软件. 为了使分类更加清晰, 本文按面向的对象将检测二进制软

件复用的第三方库的技术分为面向二进制应用软件和面向第三方库软件的技术. 为避免重复分析, 本文着重梳理了 2021 年至今检测二进制软件复用的第三方库的相关文献.

Ishio 等人[118]提出的方法收集大量的第三方库, 针对第三方库的类文件, 提取特征, 作为类的签名, 进而形成特征数据库. 同样提取待分析软件的类特征, 与特征数据库中的特征进行比较, 与待分析软件的类特征重复最多的第三方库被视为可能的组件.

Wei 等人[119]提出了 BCFinder. BCFinder 收集 C/C++项目, 提取项目的字符串作为项目特征, 构建特征数据库. 使用字符串作为项目特征的原因是这些字符串在编译后仍保留在二进制文件中, 是标识项目的有效依据. 对于待分析的二进制软件, BCFinder 从二进制软件中提取字符串作为特征. 特征数据库中与二进制软件的特征匹配高于阈值的项目被视为二进制软件可能使用的第三方库.

Zhang 等人[120]提出了 OSLDetector. OSLDetector 收集大量的二进制软件, 提取字符串作为特征, 形成特征数据库. 对待分析的二进制软件, 同样提取字符串作为特征, 通过与特征数据库中的特征进行匹配获取二进制软件使用的第三方库. 考虑到第三方库之间的重用, OSLDetector 构建第三方库的克隆森林, 过滤掉由于第三方库间的复用带来的错误匹配, 最终留下更可能的第三方库.

Zhan 等人[121]提出了 ATVHunter. ATVHunter 收集大量的第三方库, 提取每个库的 CFG 作为粗粒度特征, 提取 CFG 中每个基本块的操作码序列作为细粒度特征, 得到特征数据库. ATVHunter 对二进制软件进行反编译, 将应用转换为字节码, 解析 AndroidManifest.xml 文件, 删除主模块以减少干扰. 然后, 利用类依赖关系图将非主模块拆分为独立的候选第三方库. 接着, ATVHunter 采用两阶段识别方法, 首先比较候选第三方库与特征数据库中的粗粒度特征快速定位潜在库, 然后比较候选第三方库与潜在库的细粒度特征确定具体库版本.

Yang 等人[122]提出了 ModX. ModX 利用反编译工具将二进制软件转换为汇编代码, 分析这些代码构建 CG. 接着, 使用改进的 Louvain 算法根据 CG 将二进制软件模块化. ModX 使用相同的方法将收集的第三方库模块化. 然后, ModX 提取每个模块的语法特征, 图相似性特征和函数相似性特征, 计算模块与已知第三方库模块之间的相似性分数确定程序中使用的第三方库.

Tang 等人[123]提出了 LibDB. LibDB 使用反汇编工具将二进制软件转换为汇编代码, 提取字符串和函数名作为基本特征, 同时通过图嵌入网络将函数的 CFG 转化为向量. 接着, LibDB 基于基本特征和函数向量与已知第三方库的基本特征和函数向量进行比较, 生成两个候选列表. 最终根据相似度分数确定最终的库版本.

Wu 等人[124]提出了 LibScan. LibScan 反编译二进制软件, 将其转换为 Java 类. 然后, 根据中间表示提取类级别、字段级别和方法级别的代码特征, 生成每个类的指纹, 并对二进制软件类和第三方库类的特征进行匹配, 若指纹相同则继续比较方法操作码和调用链操作码的相似性. 最后, 计算第三方库类在二进制软件中的覆盖率, 将覆盖率分数与预设阈值比较, 以确定可能的第三方库.

Li 等人[125]提出了 LibAM. LibAM 提取二进制软件和第三方库的方法调用图 (call graph, CG), 并将每个方法转化为向量表示. 然后, 使用匹配算法找出相似的方法对, 并扩展这些方法及其调用的功能, 生成方法区域. 接着, 利用神经网络计算方法区域的相似度, 使用对齐算法判断这些区域是否来自第三方库. 最终确定复用的第三方库.

Domínguez-Álvarez 等人[126]提出了 LibKit. LibKit 从 CocoaPods 中收集大量精确到版本的第三方库, 提取第三方库的类特征, 存储在特征信息数据库中. LibKit 解密 iOS 二进制软件, 提取其类特征, 将这些特征与特征信息数据库中的特征进行匹配, 计算相似度, 确定最佳匹配的库版本.

Huang 等人[127]提出了 LIBLOOM. LIBLOOM 将二进制软件的代码分成包和类两个层次, 为包和类生成结构签名, 将这些签名编码到布隆过滤器中. LIBLOOM 使用包级别的布隆过滤器, 查询二进制软件和第三方库之间的签名集合重叠相似度, 快速筛选出候选第三方库. 对于候选库, LIBLOOM 使用类级别的布隆过滤器进行更精细的分析, 通过子集查询计算每个候选第三方库与二进制软件之间的相似度分数. 最终识别二进制软件中的第三方库.

Almanee 等人[128]提出了 LibRARIAN. LibRARIAN 分析第三方库的元数据、符号表信息和字符串信息, 作为特征数据库. 用同样的方式提取待分析的二进制软件的特征, 通过计算相似系数衡量特征向量之间的相似性, 进而识别出二进制软件使用的第三方库.

Zhang 等人[129]提出了 LibHawkeye. LibHawkeye 收集大量的 Android 应用, 分析每个应用的内部结构, 从 AndroidManifest.xml 中提取出应用的包名、Dalvik 字节码, 解析 Dalvik 字节码中的方法调用、字段引用和类继承关系, 构建应用的依赖关系图. 将图中独立的子图视为潜在的第三方库并生成特征. LibHawkeye 对所有潜在第三方库的特征进行聚类分析, 如果某个库被很多应用复用, 那么该第三方库的特征会形成一个大型集群.

Zhao 等人[130]提出了 FIRMSEC. FIRMSEC 收集第三方库, 分别提取固件和第三方库的语法特征和 CFG 特征, 结合语法特征和 CFG 特征的比较结果确定固件中使用的第三方库.

Wang 等人[131]提出了 JHunter. JHunter 收集第三方库, 生成类依赖图作为包级别的特征, 生成 CFG 和提取字符串作为类级别的特征, 形成特征数据库. 对待分析的二进制软件, 同样生成类依赖图, 使用图神经网络和余弦相似度与特征数据库中的类依赖图进行比较, 得到候选的第三方库列表, 进一步比较待分析软件和候选第三方库的类级别的特征, 得到最可能的第三方库.

表 13 对比总结了上述方法. 这些方法的特征提取和分析结果的准确性都受代码混淆技术的影响. 此外, 众多相关工作中仅 LibHawkeye 是面向二进制应用软件的分析技术, 大多数相关工作更倾向于使用面向第三方库软件的分析技术. 这与 Zhan 等人[117]的分析结果不谋而合. 究其原因, 面向第三方库软件的分析技术直接从具体到版本的第三方库中提取特征, 更能精确地识别软件使用的第三方库.

表 13 基于静态分析依赖关系的 SBOM 生成技术方法对比

方法	处理数据	优点和贡献	缺点和局限
文献[118]	类名, 字段, 方法等信息的哈希值	引入扩展的类签名和贪心算法, 提高了检测效率和准确性	识别未知组件的能力有限
BCFinder	二进制软件中的字符串	进行了数据清洗, 设计了快速匹配的方法	功能相似的组件可能导致误报和漏报, 抗混淆能力差
OSLDetector	二进制软件的字符串	进行了特征过滤, 采用了内部克隆森林的方法, 有效减少了特征重复带来的误报	仍存在特征重复导致的误报和漏报
ATVHunter	CFG, 操作码序列	第三方库数据库较为全面, 精确识别第三方库的版本	提取细粒度特征增加了计算成本
ModX	语法特征, 图相似性特征, 函数相似性特征	创新的程序模块化算法	模块化的质量直接影响第三方库检测的准确性
LibDB	字符串, 导出函数名, CFG	直接利用函数内容, 设计了一个改进的比较算法, 使用细粒度的函数特征来识别版本	依赖预构建数据库, 目标和候选之间存在相似函数会导致误报
LibScan	类级别, 字段级别和方法级别	两阶段匹配法, 提高了检测的准确性	对传统的优化技术敏感, 影响结果的准确性
LibAM	CG, 方法区域相似度	提出了新颖的区域匹配框架, 性能优越	检测复杂重用关系的能力不足
LibKit	类特征	支持静态和动态链接库, 支持多种编程语言	数据来源依赖于特定的仓库
LIBLOOM	包和类的结构签名	有效处理复杂的代码混淆, 提出了基于熵的度量方法	与目标应用重名的第三方库会在过滤步骤中被排除
LibRARIAN	元数据, 符号表信息, 字符串信息	解决本地库及其版本识别的难题	检测效果受特征数据库的影响
LibHawkeye	方法签名, Dalvik字节码指令序列, 方法调用序列, Merkle树结构	分析多个特征改进了库识别的准确性	只在小规模数据集上评估
FIRMSEC	语法特征, CFG	专注于固件安全	特征数据库数量小
JHunter	类依赖图, CFG, 字符串	引入了包级别的特征和类级别的特征, 提高了检测的速度和精度	计算资源需求大

4.2.1.2 基于动态分析依赖关系的 SBOM 生成技术

与基于静态分析的生成技术不同, 基于动态分析的技术可以监控软件运行时的状态, 检测软件动态依赖的组件. 图 8 展示了基于动态分析依赖关系的 SBOM 生成技术. 首先, 安装软件到本地, 监控软件的运行, 记录软件运行时的信息. 然后, 分析记录的信息, 获得软件依赖的组件信息. 最后, 整理得到的组件信息, 生成并验证 SBOM.

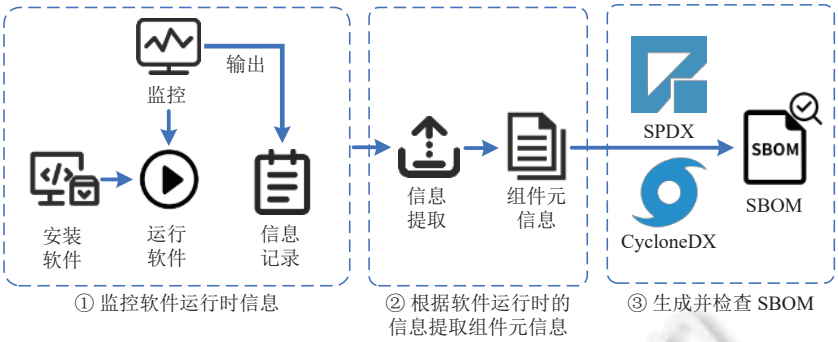

图 8 基于动态分析依赖关系的 SBOM 生成技术框架

基于动态分析的 SBOM 生成技术不仅可以处理二进制软件, 还可以处理源码软件. 源码软件与二进制软件的最大区别在于二进制软件无需编译, 而源码软件需要先编译. 对于源码软件, 同样先安装到本地, 记录并分析软件的运行信息, 得到源码软件依赖的组件信息. 然而, 无论是面向二进制软件还是面向源码软件, 基于动态分析的技术均受到环境配置, 监控限制和版本限制等因素的影响, 导致分析结果不准确.

Cui 等人[132]提出了 LibHunter, 创新的使用动态分析技术分析软件使用的第三方库. LibHunter 在应用中注入监控代码, 实时捕捉应用运行时发出的网络请求及其执行过程. 收集数据后, LibHunter 执行聚类算法对请求及调用栈进行聚类. 如果多个不同的应用发出相似的请求且由相似的调用栈触发, 那么这些请求很可能来源于同一个第三方库. Kawaguchi 等人[44]提出了一个容器管控系统, 其关键组件 SBOM 容器网关启动容器并捕捉运行时加载的库和执行的文件, 得到容器动态依赖的组件信息. Clementi 等人[133]使用 ptrace 系统调用分析正在运行的应用, 捕获应用的依赖项. Imtiaz 等人[85]搜索并筛选了 9 个 SCA 工具, 其中一种商业工具通过监控并记录项目的测试信息来获取依赖信息.

表 14 对比总结了上述方法. 动态分析的方法通过捕获网络请求, 抓取系统调用以及测试软件等方式监控软件运行时的状态, 挖掘监控信息, 识别软件使用的第三方库. 但前提需要配置合适的环境运行软件, 这比静态分析的难度大, 并且检测效果受环境配置等因素的影响, 所以基于动态分析依赖关系的技术较少, 更多的方法选择使用的静态分析的技术.

表 14 基于动态分析依赖关系的 SBOM 生成技术方法对比

方法	处理数据	优点和贡献	缺点和局限
LibHunter	应用运行时发出的所有网络请求及其执行过程	无需分析现有的软件或第三方库存储仓	无法识别不发送网络请求的第三方库
文献[44]	捕捉容器运行时加载的库和执行的文件	提出了一个集成SBOM的容器部署和监控系统	包管理器和文件路径影响系统的准确性
文献[133]	使用 ptrace系统调用捕获应用运行时使用的依赖项	收集的依赖信息更加全面	无法拦截所有系统调用
文献[85]	监控并记录项目的测试信息	筛选出真正使用的第三方库	需要测试用例, 无法保证测试到所有使用情况

4.2.2 面向开源片段引用依赖的 SBOM 生成技术

由于软件的源码存在复用代码片段的情况, 源码编译成的二进制软件同样也存在复用代码片段的情况. 考虑到复用代码片段带来的多种风险和潜在危害, 为了规避这些风险, 细粒度分析二进制软件, 检测二进制片段的相似性是必要的.

图 9 展示了面向开源片段引用依赖的 SBOM 生成技术的框架. 首先, 使用反编译工具得到二进制代码的中间表示, 例如图、汇编指令等. 然后, 根据中间表示的不同形式, 将中间表示转化为特征向量, 接着通过分析特征向

量, 推断出二进制代码片段间的相似度, 进而得到被分析的二进制代码片段的来源, 最后根据二进制代码片段的来源信息生成并验证 SBOM.

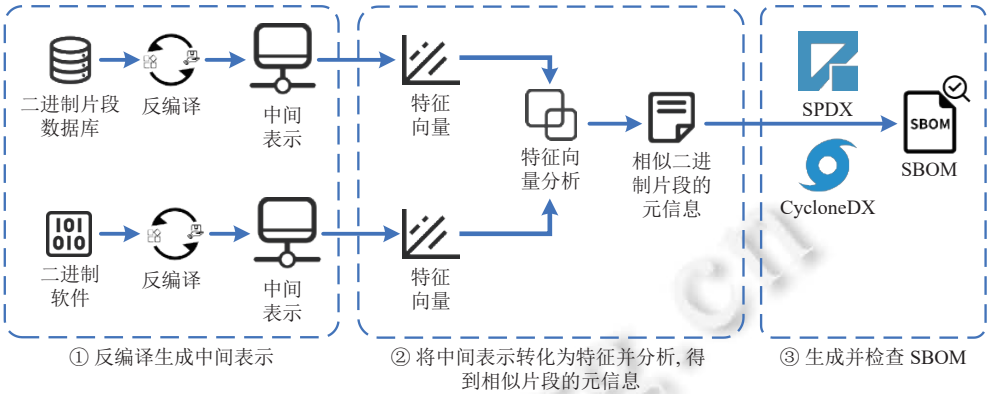

图 9 二进制级面向开源片段引用依赖的 SBOM 生成技术框架

Yang 等人[134]提出了 Asteria. Asteria 使用工具反汇编和反编译, 提取 AST, 将 AST 中的每个节点映射并替换为一个整数值, 并将 AST 转换为左子右兄弟格式. 通过 Siamese 网络将两个预处理后的 AST 分别编码为两个向量, 计算 AST 相似性, 作为最终的函数相似性.

Benoit 等人[135]提出了 PSS. PSS 为程序构建 CG 和 CFG 作为程序的中间表示, 从图中提取程序的特征向量, 并计算特征向量的距离, 得到程序对分别在 CG 和 CFG 下的相似性, 形成综合的相似性度量. 提取并存储程序库中每个程序的特征向量, 当分析某个目标程序时, 用相同的步骤提取目标程序的特征向量, 与程序库中的特征向量进行比较, 即可找到目标程序的来源.

Qasem 等人[136]提出了 BinFinder. BinFinder 收集源代码并编译生成二进制文件, 创建数据集. 使用反编译工具提取二进制文件的特征. 接着使用 Siamese 网络, 根据特征生成特征向量, 分析特征向量, 实现检测二进制函数克隆.

Jiang 等人[137]提出了 BinaryAI. BinaryAI 对二进制文件进行反编译, 提取方法级代码片段. 训练一个基于 Transformer 的模型, 生成这些代码片段的特征向量, 初步与大规模的开源项目的源代码片段进行匹配. 在初步匹配的基础上, 通过分析代码片段在二进制文件中的相对位置和调用关系, 进一步提高匹配的准确性. 最终, 通过匹配到的源代码片段确定二进制代码片段的来源.

Li 等人[138]提出了 GenTAL. GenTAL 随机选择汇编代码的某些指令进行掩码, 针对汇编代码中的每个指令, 连同指令的位置一起映射到一个多维向量空间, 表示整个指令的向量. Transformer 处理这些向量, 输出表示整个汇编代码的抽象语义信息, 将 CLS 向量视为整个代码片段的总结. 将 CLS 向量进行位置编码, 以恢复被掩码的指令, 用于更有效的相似性检测. GenTAL 通过计算向量之间的距离来判断代码片段的相似程度.

Jiang 等人[139]提出了 BinCola. BinCola 将同一源码使用不同的编译器编译成二进制构成正样本, 将不同源码的二进制视为负样本, 从二进制中提取指令, CFG 和 CG 作为特征, 并扩展为一个高维特征输入 Transformer, 训练模型学习相关性, 区分不同点. 最终利用训练好的 Transformer 模型为二进制代码片段生成语义特征, 通过计算语义特征的相似度判断二进制代码间的相似性.

Zhang 等人[140]提出了 PDG2VEC. PDG2VEC 将二进制代码提升为与架构无关的中间表示, 在中间表示的基础上, 借助工具提取函数的程序依赖图 (program dependence graph, PDG), 从 PDG 中提取与变量相关的子图, 在更细的粒度上进行相似性比较, 同时消除 PDG 中的重复变量, 提高特征描述的准确性和完整性. 然后, 基于提取的子图评估不同函数之间的相似性.

此外, 还有研究人员提出增强二进制代码相似性分析性能的方法. Xu 等人[141]提出了 DiEmph. DiEmph 的核心思想是识别并忽略二进制代码中对程序语义来说是无关紧要但对模型有干扰的指令. Wai 等人[142]提出了 BinAug 框架. BinAug 框架分为白箱输入修复和黑箱输入修复. 对于白箱输入修复, BinAug 调整模型的注意力, 对齐图节点来辅助增强模型的性能. 对于黑箱输入修复, BinAug 丰富输入数据的信息, 降低数据的复杂度来辅助增强模型的性能.

表 15 对比总结了上述方法. 不同方法之间的特性在于提取二进制代码片段特征信息的方式不同. 使用工具提取如指令序列, CFG 等多种特征信息, 结合深度学习模型, 如 Siamese 网络、 Transformer 编码器等, 从多个维度捕捉代码的语法、结构和语义特征, 进一步提高相似性检测的精度. 最后计算特征向量之间的相似度, 如余弦相似度或向量距离, 判断二进制代码片段的相似性, 并通过数据增强, 指令重要性分析等技术优化模型的性能, 降低误报率, 提升检测效率.

表 15 面向开源片段引用依赖的 SBOM 生成技术方法对比

方法	处理数据	优点和贡献	缺点和局限
Asteria	节点映射为整数值, 左子右兄弟格式的AST	提高了跨平台二进制代码相似性检测的准确性和效率	依赖于反编译结果, 计算开销大
PSS	构建CG和CFG, 计算图的谱	将光谱分析应用到相似性检测中, 提高了检测效率和精度	预处理时间较长, 降低预处理时间会影响结果的全面性
BinFinder	由反编译得到特征向量	抗代码优化和混淆能力强, 适用于多CPU架构	应对部分混淆技术存在困难
BinaryAI	方法级代码片段的特征向量	创新地探索了二进制与源码的相似性	依赖于大规模数据集
GenTAL	汇编代码的指令级别表示	有效地表示汇编代码的语义, 鲁棒性和适应性强	检测性能可能会受到OOV (out-of-vocabulary)的影响
BinCola	二进制代码的指令, CFG和CG特征	多粒度提取特征并融合, 检测性能提升, 可扩展性高	检测性能受优化级别和架构的影响
PDG2VEC	基于与架构无关的中间表示生成PDG	语义信息更丰富, 可扩展性高	复杂度高且计算消耗大
DiEmph	二进制代码	更专注于语义上重要的指令, 提高了模型性能	只能增强Transformer模型的性能
BinAug	二进制代码和模型	增强模型的性能, 丰富输入数据的信息, 降低数据的复杂度	只能增强DNN模型的性能

4.3 SBOM 生成技术比较

表 16 总结比较了不同 SBOM 生成技术的应用效果、适用场景以及局限性, 并根据 SBOM 商业工具对工具的描述将相关工具分类到不同的技术中, 提供不同技术的工具示例.

表 16 SBOM 生成技术比较

生成技术			应用效果	适用场景	局限性	工具示例
源码级SBOM生成技术	面向组件依赖的SBOM生成技术	基于组件依赖清单文件的SBOM生成技术	技术相对简单, 能直接获得软件声明的依赖信息	提供清单文件的软件	不适用复杂环境	FOSSCheck[143], 雳鉴SCA[144]
		基于代码文件依赖关系分析的SBOM生成技术	实现难度中等, 推测出的依赖信息不一定准确	以源码形式发布且清单文件缺失的软件	分析结果受知识库的限制	墨菲安全[145], 雳鉴SCA
	面向开源片段引用依赖的SBOM生成技术		实现难度较高, 可以有效地揭示潜在的版权和安全问题	以源码形式发布, 且倾向于直接复用代码片段的软件, 以及需要进行细粒度分析的场景	分析对复用代码片段进行变异的情况效果不好	FOSSCheck, 雳鉴SCA, 墨菲安全

表 16 SBOM 生成技术比较 (续)

生成技术			应用效果	适用场景	局限性	工具示例
二进制级SBOM生成技术	面向组件依赖的SBOM生成技术	基于静态分析的SBOM生成技术	实现难度较高, 推测出的依赖信息不如源码级分析技术准确	以二进制形式发布的软件	分析结果受代码混淆和优化技术的影响	墨菲安全
		基于动态分析的SBOM生成技术	需要复杂的配置, 实现难度高, 能识别动态依赖	需识别动态依赖, 或依赖信息极其复杂的应用	可能无法触发或发现所有依赖项	Commercial B[85]
	面向开源片段引用依赖的SBOM生成技术		实现难度较高, 可以有效地揭示潜在的版权和安全问题	以二进制形式发布, 且倾向于直接复用代码片段的软件, 以及需要进行细粒度分析的场景	分析结果受代码混淆和优化技术的影响	墨菲安全

面向清单文件的分析技术相对简单, 主要依赖于软件声明的清单文件和包管理器, 但因此也严重受限于清单文件和包管理器, 更适用于已经明确提供清单文件和有成熟的包管理器的软件, 当软件不提供清单文件或者清单文件未准确声明软件的依赖时, 该方法效果不佳. 此外, 自定义或非标准的清单文件可能不会被识别或解析, 进而导致生成的 SBOM 不完整.

面向代码文件的分析技术借助代码提供的依赖信息查询知识库来获得更详细的依赖信息, 实现难度中等, 该方法更适合以源码形式发布但清单文件缺失或不完整的软件. 但是, 如果源码使用了动态加载的组件, 该技术可能无法有效地识别软件的动态依赖, 进而无法识别软件的所有依赖. 此外, 该方法受知识库的完备性和准确性的限制, 可能导致误报或漏报, 并且识别出详细的依赖信息后还要耗费更多的资源推敲依赖项的版本, 避免依赖项冲突.

面向源码代码片段的分析技术通过检测代码片段的相似性来识别软件复用的代码片段, 检测粒度更细, 但实现难度也较高, 更适用于以源码形式发布且倾向于直接复用代码片段的软件, 以及需要对软件进行细粒度分析以发现潜在风险的场景. 此外, 该分析技术在处理大规模代码库时计算资源消耗大, 可能误判或漏判.

基于静态分析的分析技术涉及逆向工程, 实现难度较高, 更适用于以二进制形式发布的软件. 由于代码混淆和编译时的信息损耗, 导致获得的依赖信息不如分析源码获得的依赖信息准确可靠. 此外, 即使是相似的源码, 如果编译时使用了不同的工具, 编译得到的二进制代码会存在差异. 由于并不能保证被分析的二进制软件和收集的二进制软件或第三方库在编译时使用了相同的工具, 所以最终的分析结果可能会受逆向工具的影响. 并且, 静态分析可能无法揭示动态加载的组件, 导致最终得到的信息不够准确.

基于动态分析的分析技术通过监控软件的运行来识别软件的依赖, 需要复杂的配置, 实现难度高, 更适用于识别动态依赖, 或依赖信息极其复杂的应用. 但受测试环境和配置的限制, 可能无法触发或发现所有动态依赖. 并且动态分析通常需要较长的准备和执行时间, 且可能会因环境差异导致结果不一致.

面向二进制代码片段的分析技术依赖于逆向工程和特征生成, 实现难度较高, 更适用于以二进制形式发布且倾向于直接复用代码片段的软件, 以及需要对软件进行细粒度分析以发现潜在问题的场景. 二进制分析依赖于复杂的反编译技术, 可能受代码混淆和优化技术的影响, 导致分析结果不准确, 并且分析过程资源消耗大.

不同技术的相同点是从待分析的软件中尽可能多地获取软件信息, 借助软件信息对比和查询当前已有的信息, 进而获得更详细的软件信息. 不同分析技术的不同点为更全面的获取软件信息提供了思路: 结合使用多种分析技术以期更全面地理解软件. 例如, 面向清单文件的分析技术结合面向代码文件的分析技术可以进一步验证, 补充软件信息. 基于静态分析的技术结合基于动态分析的技术, 可以有效地识别软件的动态依赖, 进一步丰富软件信息. 前述分析方法都是组件级, 结合更细粒度的面向开源片段引用的分析技术能更深层次的挖掘软件信息, 为后续使用 SBOM 管理软件, 分析风险, 分析合规性提供更详细的数据支撑.

5 SBOM 工具及性能分析

5.1 SBOM 工具分类

NTIA 格式与工具工作组对 SBOM 工具进行分类[146−148], 表 17 展示了工具的分类和示例.

表 17 SBOM 工具分类及示例

工具种类	工具的主要功能	举例
生成	构建	bom, in-toto, ort
	分析	bom, sbom-tool, scancode-toolkit, syft, tern, dependency-track, swidGenerator
	编辑	sbom-demo, swidgen
消费	查看	bom, cyclonedx-cli, tools-java, dependency-track
	比较	cyclonedx-cli, ort, tools-java
	导入	ort, tools-java, dependency-track
转换	格式转换	tools-java, syft
	合并	cyclonedx-cli, tools-java
	工具支持	bom, tools-java, SwidTag

生成工具主要用于创建和编辑 SBOM, 细分为构建工具, 分析工具和编辑工具. 构建工具将生成 SBOM 视为构建软件的一部分, 在构建软件时自动生成 SBOM. 通过构建工具生成的 SBOM 反映软件构建时的依赖, 同时确保 SBOM 的内容与当前构建的软件保持一致. 分析工具分析软件的源码或二进制文件生成 SBOM. 通过分析工具生成的 SBOM 适用的场景广, 但分析的准确性可能不高. 编辑工具通过用户手动输入来编辑 SBOM. 通过编辑工具生成的 SBOM 需要耗费大量人力, 可能导致错误, 而且效率较低, 更适合需要人工干预调整 SBOM 的场景.

消费工具主要用于协助用户使用 SBOM, 细分为查看工具, 比较工具和导入工具. 查看工具以人类可读的形式协助用户查看和使用 SBOM. 比较工具比较多个 SBOM 之间的差异, 协助用户识别软件不同版本之间的变化. 导入工具将 SBOM 导入到系统中, 还用于发现和检测系统中的 SBOM, 协助 SBOM 的迁移.

转换工具主要用于将一种形式的 SBOM 转换为另一种形式, 细分为格式转换工具, 合并工具和工具支持. 格式转换工具在保持内容不变的情况下将 SBOM 由一种文件类型转换为另一种文件类型, 确保 SBOM 适用于不同的使用场景. 合并工具将多个 SBOM 和其他数据合并到一起, 整合不同来源的 SBOM, 提供统一的视图. 工具支持借助 API 等技术的支持使用户直接调用现成的技术就能轻松的处理 SBOM, 也便于在不同场景中使用 SBOM.

许多 SBOM 工具因其丰富的功能和灵活性, 涵盖了多个工具种类. 例如, Syft 不仅可以在软件的构建过程生成 SBOM, 还可以为已有的软件生成 SBOM, 还能导入和处理 SBOM 数据. CycloneDX CLI 不仅可以生成和编辑 SBOM, 还支持转换 SBOM 的格式. 工具的多功能性使得这些工具能够适应不同的使用场景和需求, 优化软件供应链管理和风险分析流程.

Mirakhorli 等人[64]实证研究了 84 个开源和专有的 SBOM 工具的现状, 发现 SBOM 生成工具是当前最常见的类型, 占总工具数的 64%, SBOM 消费工具占总工具数的 37%, SBOM 转换工具占总工具数的 18%. 特别地, Mirakhorli 等人还发现总工具中有 17% 的工具用于提升 SBOM 质量, 并将这些工具称为 SBOM 质量保证工具. 此外, 总工具中还有 15% 的工具提供存储和管理 SBOM 数据的服务, 并将这些工具称为 SBOM 服务工具.

5.2 工具性能分析

尽管存在众多工具, 但始终缺乏一个评价标准, 对这些工具的评价只是开发组织或研究人员单方面的观点. 参考 Linux 提出的评估 SBOM 工具的指标[149], 表 18 总结了评估 SBOM 工具可参考的指标.

Dalia 等人[66]对生成 SBOM 的现有工具进行了比较分析, 发现 Syft 没有与 Maven 集成, 用户友好度低, 但支持多个平台, 多种输出格式, 支持容器, 并且能深入分析依赖关系. OpenSBOM's SPDX 仅支持多个平台, 能深入分析依赖关系. kubernetes-sigs 支持多个平台和容器, 用户友好度高. Microsoft SBOM tool 支持多个平台, 集成了 Maven, 用户友好度高. Tern 支持多种输出格式, 支持容器, 用户友好度高. CycloneDX 支持多个平台, 集成了 Maven, 并且能深入分析依赖关系.

表 18 评估 SBOM 工具的指标

指标		描述	说明
支持语言		工具支持分析的编程语言	理想情况下工具应该是与语言无关的, 但很少有工具能实现这一点. 支持的语言越多, 工具的普适性越广, 分析复杂软件的能力越强
数据库	知识数据库	存储了开源代码	知识库涵盖的范围越广, 更新的越频繁, 能识别出来的开源代码就越多
	漏洞数据库	存储了已知的漏洞信息	漏洞数据库来源越广, 更新的越频繁, 识别漏洞的效果越好
	许可证数据库	存储了许可证信息	许可证数据库来源越广, 更新的越频繁, 识别的效果越好
检测能力		工具检测分析风险的能力	工具除了需要扫描源码, 二进制, 还应该支持扫描代码片段, 确定代码片段的来源
集成能力		工具与其他系统, 平台和工具协同工作的能力	强大的集成能力使工具在各种环境中都能轻松使用
输出格式		检测报告输出的格式	工具不仅要提供详尽的检测报告, 还应支持多样化的输出格式, 满足不同场景下的需求, 也便于报告的分享和交流
用户友好		工具是否用户友好	用户友好的SBOM工具应该设计简单, 容易使用

此外, 许多研究人员实际探究了 SBOM 工具生成 SBOM 的性能. Torres-Arias 等人[60]使用测评工具测评了 4 个 SBOM 生成工具. Balliu 等人[61]使用 6 个 SBOM 工具分别为 26 个 Java 项目生成 SBOM, 评估工具的精度和召回率. Rabbi 等人[63]使用 4 个 SBOM 工具分别为 50 个 JavaScript 项目生成 SBOM, 评估工具的准确性, 精确性和召回率. Halbritter 等人[65]探究了 8 个 SBOM 工具的准确性和可靠性. Zhao 等人[83]分析了 4 个开源 SCA 工具以及 2 个商业工具 T1 和 T2. 表 19 汇总了不同研究人员对 SBOM 工具的评估.

表 19 SBOM 工具的评估

工具	测评结果
Syft	生成的SBOM没有覆盖SBOM基本元素[60,65]; 不提供组件间的依赖关系, 不提供校验和, 识别组件名称和版本的准确率, 精确率和召回率均维持在80%左右[63]
bom	生成的SBOM没有覆盖SBOM基本元素[60]
Tern	生成的SBOM没有覆盖SBOM基本元素[60]
Trivy	生成的SBOM没有覆盖SBOM基本元素[60]
Build-Info-Go	可重复性差, 不提供组件的使用范围, 召回率大于80%, 精确率接近100%, 表现最佳[61]
CycloneDX-Generator	召回率接近100%, 精确率大于80%[61]; 识别组件名称和版本的准确率, 精确率均维持在80%左右, 召回率维持在90%左右, 识别组件间依赖关系的准确率, 精确率和召回率均在50%以下[63]; 生成的SBOM没有覆盖SBOM基本元素[65]
CycloneDX-Maven-Plugin	召回率大于60%, 精确率接近100%[61]
Depscan	召回率接近100%, 精确率大于80%[61]
jbom	不提供依赖的层次结构, 可重复性差, 召回率约20%, 精确率大于20%, 表现最差[61]
OpenRewrite	不提供校验和, 召回率接近40%, 精确率接近100%[61]
OSS Review Toolkit	不提供组件间的依赖关系, 识别组件名称和版本的准确率、精确率和召回率均维持在70%左右[63]
cycloneDX-node-module	不提供校验和, 识别组件名称的准确率、精确率、召回率均接近100%, 识别组件版本的准确率和召回率维持在80%左右, 精确率接近100%, 识别组件间依赖关系的准确率和召回率接近80%, 精确率维持在90%左右[63]
CycloneDX-Python	生成的SBOM基本覆盖SBOM基本元素[65]
SBOM4Python	生成的SBOM没有覆盖SBOM基本元素[65]
CycloneDX-Conan-Extension	生成的SBOM基本覆盖SBOM基本元素[65]
SBOM4Rust	生成的SBOM没有覆盖SBOM基本元素[65]
CycloneDX-Npm	生成的SBOM基本覆盖SBOM基本元素[65]
Covenant	生成的SBOM没有覆盖SBOM基本元素[65]
OWASP Dependency Check	识别组件间依赖关系的精确率和$F1$分数大于90%, 召回率接近90%[83]
Eclipse Steady	识别组件间依赖关系的精确率大于90%, 召回率大于70%, $F1$分数大于80%[83]

表 19 SBOM 工具的评估 (续)

工具	测评结果
Dependabot Alerts	识别组件间依赖关系的精确率大于50%, 召回率接近30%, *F*1分数大于30%[83]
OSSIndex	识别组件间依赖关系的精确率和*F*1分数大于90%, 召回率大于80%[83]
T1	识别组件间依赖关系的精确率大于90%, 召回率大于70%, *F*1分数大于80%[83]
T2	识别组件间依赖关系的精确率、召回率、*F*1分数均大于80%[83]

6 SBOM 应用

6.1 软件供应链管理

NTIA 将软件供应链简化为 3 个核心角色: 开发者, 选择者和操作者. 开发者开发软件, 选择者采购软件, 操作者使用软件. 对开发者来说, SBOM 能协助优化软件管理, 识别生命周期处于终点的组件, 并提前规划替代方案, 迅速明确软件是否受到漏洞影响, 减少不必要的检查时间. 此外, SBOM 还能协助开发者主动遵守许可证条款, 避免法律风险. 对选择者来说, SBOM 能协助实现购入高质量的软件, 识别购入的软件是否存在漏洞, 是否合规, 组件是否被篡改. 对操作者来说, SBOM 能协助评估软件的安全性. 当发现漏洞时, SBOM 能协助操作者在软件供应商发布补丁前主动制定应对漏洞的措施.

许多领域也在积极响应使用 SBOM 的号召. 大西洋理事会分析了针对软件供应链的攻击, 强调 SBOM 对软件供应链的重要性, 建议将 SBOM 标准纳入国家层面的标准. 用于增强美国国防部供应链安全的网络安全成熟度模型认证框架建议在资产管理和配置管理中使用 SBOM. 爱迪生电气协会开发的采购合同示范语言涵盖了 SBOM 相关的内容. 欧盟网络安全机构建议使用 SBOM 保护物联网供应链安全. 美国食品和药物管理局建议设备应提供材料清单. SBOM 在多个领域中的应用, 突出了其在提高供应链透明度, 增强供应链安全等方面的重要性. 然而, 进一步推广 SBOM 的应用必须先生成高质量的 SBOM, 而这也是当前 SBOM 面临的一大难题.

6.2 安全风险分析

后文图 10 展示了应用 SBOM 分析安全风险的步骤. 理想状态下, 开发者生成并维护软件的 SBOM. 当发现新的安全威胁时, 通过 SBOM 快速定位受影响的组件并及时修复漏洞, 更新 SBOM 以反映更改. 选择者在软件采购过程中, 获取并分析供应商提供的 SBOM, 检查组件是否存在已知漏洞或潜在安全风险, 决定是否采购该软件, 避免采购高风险的软件. 操作者导入并查看 SBOM, 利用自动化工具持续监控漏洞情况, 迅速定位受影响的组件并采取自主防御措施, 而不必等待供应商更新.

以 Apache Log4j 事件为例. 开发者在软件开发阶段通过 SBOM 跟踪所有组件, 确保组件的安全性并定期更新 SBOM. 在 Log4j 漏洞披露后, 开发者利用 SBOM 迅速判断软件是否受漏洞影响, 并进行必要的修补或升级. 选择者通过分析供应商提供的 SBOM 来评估软件是否受 Log4j 影响, 决策是否购买软件. 操作者通过 SBOM 和自动化工具持续监控软件, 在 Log4j 漏洞披露的第一时间就能响应新的安全威胁, 从而最大限度地减少漏洞可能带来的影响.

许多其他领域为应对安全风险也在积极响应使用 SBOM 的号召. 美国食品药品监督管理局要求设备设计应提供机器可读的电子格式网络安全物料清单. 由医疗保健利益相关者起草的联合安全计划将 SBOM 用于评估第三方的安全性, 旨在提高供应链安全性和透明度. 北美电力可靠性公司提出的标准要求电力公司使用 SBOM 识别并评估供应链中的网络安全风险. 美国国家公路交通安全管理局在 2020 年发布的与车辆安全相关的草案中提到了使用 SBOM 辅助评估车辆安全.

6.3 合规性分析

即使软件以开源形式提供, 也仍受知识产权保护. 图 11 展示了应用 SBOM 分析合规性的步骤. 理想状态下, 开发者生成并维护 SBOM, 记录组件及其许可证信息, 定期审核以确保符合许可证要求, 及时更新不合规的组件. 选择者在采购软件时分析软件的 SBOM, 检查许可证是否符合内部政策和法规要求, 决定是否采购该软件. 操作者

使用软件时持续监控软件的 SBOM, 确保软件在整个生命周期内均符合内部政策和法规要求, 当发现软件不符合政策或法规要求时采取必要的措施, 避免法律风险.

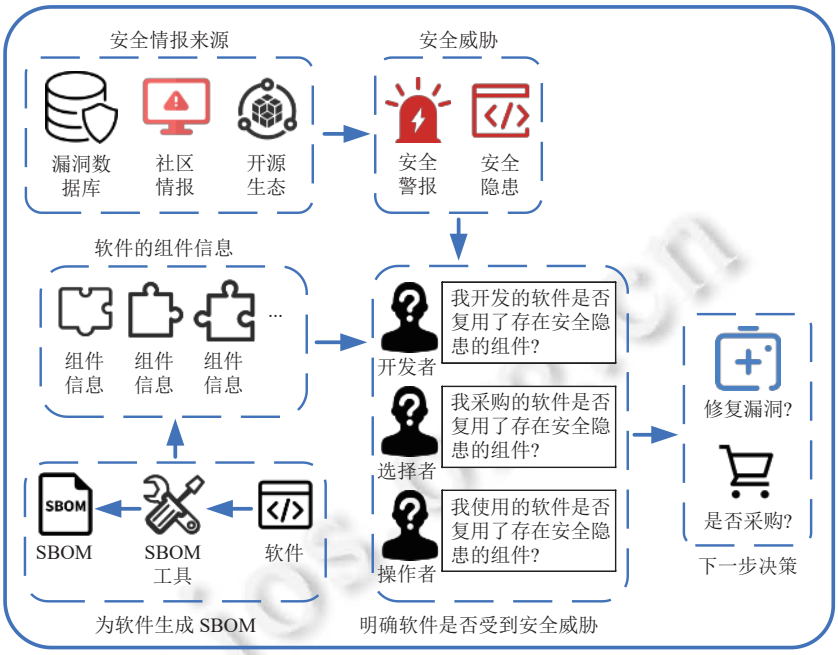

图 10 借助 SBOM 进行漏洞检测

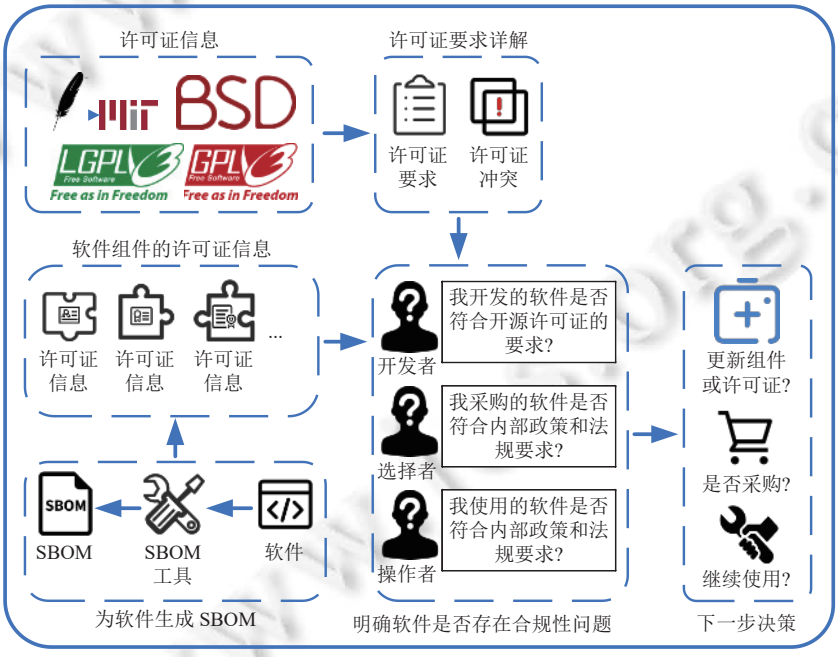

图 11 借助 SBOM 进行合规性分析

以 Jacobsen 与 Katzer 的案件为例. 开发者 Katzer 生成并维护软件的 SBOM, 定期审核 SBOM. 当发现使用了 Jacobsen 的软件但并未遵守软件许可条款时, 及时替换该软件. 选择者分析 Katzer 提供的 SBOM, 发现软件不符合

许可条款的要求, 拒绝采购存在合规问题的软件. 操作者持续监控 Katzer 提供的 SBOM, 发现软件违规, 并迅速采取必要措施, 避免法律风险.

许多领域还使用 SBOM 辅助分析合规性. Linux 基金会的 OpenChain 规范要求创建和管理开源组件的 SBOM, 特别是每个组件及其许可证信息. 美国汽车工程师学会制定的标准为汽车行业在网络安全方面的合规性提供了指导.

7 SBOM 趋势与挑战

7.1 SBOM 生成准确率问题

尽管有多种方式获得软件信息, 但无法保证信息的完整性和准确性. 大多数 SBOM 生成工具优先选择分析软件的清单文件获得软件信息, 但清单文件也并非是完整和准确的. Cao 等人[150]研究了 Python 项目中的依赖问题, 并发现普遍存在 3 种情况: ① 依赖缺失: 源码声明的依赖项未在清单文件中声明, ② 依赖膨胀: 清单文件声明的依赖项未在源码中声明, ③ 版本约束不一致: 同一个依赖项在不同清单文件中声明的版本不一致. 而且, 生成 SBOM 的工具并不验证清单文件的准确性. 只要软件提供了清单文件, 生成 SBOM 的工具就能根据该文件生成 SBOM, 即使这个清单文件属于另外一个软件. 虽然可以通过其他方式获得软件信息, 但也因各种因素导致无法获得完整准确的软件信息. Balliu 等人[61]发现, 虽然理论上生成 SBOM 是容易的, 但实验结果表明生成 SBOM 并非一件容易的事情.

即使得到了完整准确的软件信息, 仍要面对可达性问题. 可达性包括组件可达性和漏洞可达性. 组件可达性是指组件在实际运行中是否被调用. 漏洞可达性是指漏洞在实际运行中是否被触发. 可达性分析需要分析代码实际的运行情况, 是较难做到的, 但又是比较重要的. Wu 等人[151]研究了 Maven 生态系统中的漏洞, 发现漏洞误报率高达 86%, 约有 73.3% 的软件虽然依赖于易受攻击的组件但实际上是安全的. 如果发现漏洞就去修复, 工作量太大, 并且最终可能发现漏洞并没有影响到软件.

7.2 SBOM 标准化问题

虽然 NTIA 提供了 SBOM 的标准格式, 但业界仍需要进一步探讨 SBOM 格式的标准化. Bi 等人[58]发现, GitHub 显示解决与 SBOM 相关的问题平均需要花费 20.3 天, 并且仍存在许多未解决的问题, 其潜在的难点可能是缺乏标准化导致维护困难. 虽然 SBOM 有 3 种标准格式, 但这些标准格式各有侧重的使用场景. SPDX 更注重合规性分析, 而 CycloneDX 更注安全风险分析.

为满足特定的使用场景, 使用 SBOM 的组织更倾向于使用自定义的 SBOM 格式. Stalnaker 等人[59]调查了包括生产商、消费者和工具制造商在内的 50 个组织, 发现约有 1/5 的组织使用其他格式的 SBOM. 私人组织和闭源项目选择使用标准格式或自定义的格式. Xia 等人[55]发现, 一些组织仍选择使用自定义的非标准格式生成 SBOM, 虽然大多数受访者同意采用标准化的 SBOM 格式, 但仍有超过 1/4 的受访者支持使用自定义的格式. Zahan 等人[57]研究了 SBOM 的现状, 发现数据质量, 数据交换格式和工具的标准化问题阻碍 SBOM 的发展.

7.3 SBOM 数据隐私与安全问题

攻击者可能会在 SBOM 分发的过程中篡改其中的内容. Stalnaker 等人[59]认为需要对获得的 SBOM 进行验证, 检查 SBOM 是否在传输过程中被恶意修改, 可以使用基于散列和校验的方法确保 SBOM 的完整性和可信度. Xia 等人[55]发现近一半的受访者认为 SBOM 缺乏可靠的验证措施, 这意味着采购者只能依靠 SBOM 作者的声誉粗略评估 SBOM 的质量. 许多从业者还担心 SBOM 为攻击者提供攻击路线图, 协助攻击者发现软件的漏洞和弱点.

此外, SBOM 可能包含专有和敏感信息, 共享这些信息可能会带来潜在的风险. Xia 等人[55]发现大多数受访者认为 SBOM 应该供内部使用, 而不需要提供给客户. 主要原因是 SBOM 中可能包含敏感信息, 无法公开, 例如, 大多数受访者都不会为财务软件生成 SBOM. 使用内容定制和访问控制的方法可以解决敏感信息带来的问题. 这意味着只有符合规定的用户可以访问特定信息, 从而确保只允许正确的人看到正确的信息.

7.4 SBOM 动态更新机制

随着软件的更新, 软件使用的组件以及组件之间的关系都可能随之发生变化, 因此保持最新的 SBOM 对于软件的安全和稳定至关重要. 但 Xia 等人[55]发现并不是每个 SBOM 作者都能及时更新 SBOM. 理想情况下, SBOM 应该在软件开发的早期阶段生成并不断丰富和更新, 而实际上 SBOM 的生成是延迟的且不是动态的. Zahan 等人[57]也发现, 每当更新代码、修补漏洞、增加新功能和进行其他修改时, 就应该更新 SBOM, 但更新基本都是手动完成, 而软件的更改随时都可能发生, 可见 SBOM 的动态更新是必要的.

7.5 SBOM 元素的扩展

当前早已出现扩展 SBOM 基本元素的趋势. Xia 等人[55]发现一些组织选择根据自身情况定制非标准的 SBOM, 并且很多受访者同意使用定制化的 SBOM. 此外, 一些软件供应商选择 SBOM 基本元素的子集, 或者自定义 SBOM 基本元素, 以满足使用需求. 生成或构建 SBOM 工具的组织希望 SBOM 尽可能支持更多的信息, 因为更全面的 SBOM 可以更有效地帮助组织管理软件供应链, 并且生成的 SBOM 包含的信息越全面工具的市场竞争力就越强. 总之, 对 SBOM 的基本元素进行扩展以适应更多的应用场景必然成为发展趋势.

针对上述标准的 SBOM 无法满足个人和组织的使用需求, 本文提出了 SBOM+. 图 12 展示了 SBOM+的基本元素. SBOM+源自“源图 3.0”[7], 在 SBOM 的基础上增加了安全信息和合规性信息. 其中, SBOM+的安全信息源自“源图 3.0”的安全分析, “源图 3.0”的安全分析从公开漏洞库, 论坛等来源获得漏洞数据, 构建知识图谱, 通过匹配知识图谱, 获得被分析软件的安全情报, 组件级漏洞和实时感知的漏洞. SBOM+的合规性信息源自“源图 3.0”的合规性分析.“源图 3.0”的合规性分析通过深度学习智能识别许可证条款及其责任, 结合许可证条款及其责任以及扫描被分析软件获得的许可证, 感知被分析软件的合规冲突, 并给出建议使用的许可证. SBOM+同时传递软件的安全性和合规性信息, 进一步提高了软件的透明度, 协助用户进一步理解与掌控软件.

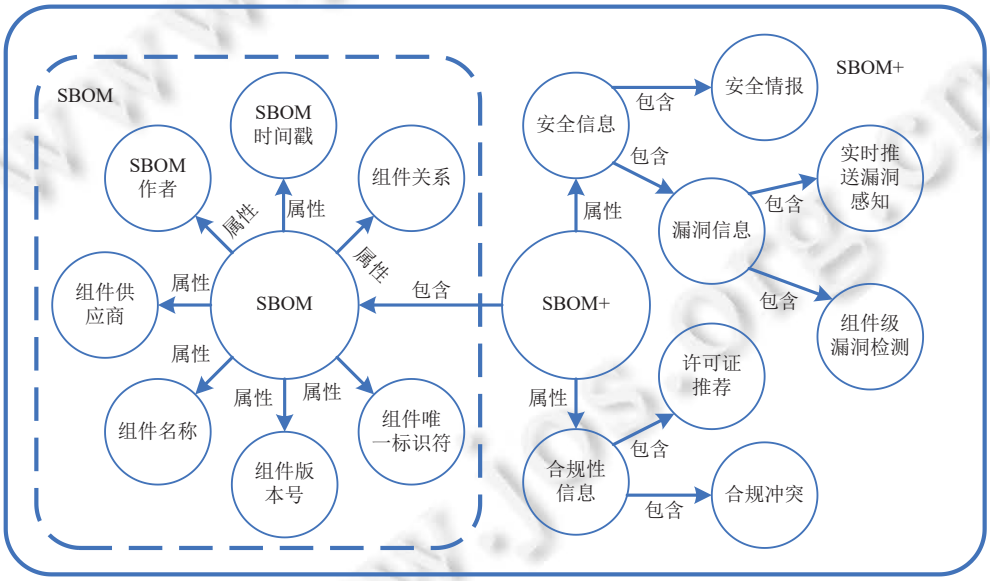

图 12 SBOM+与 SBOM 的对比

8 总 结

为应对当前软件开发主流模式存在的风险, 美国提出了 SBOM. SBOM 是软件的“配料表”, 帮助厘清软件的组成和可能的威胁, 增强软件的透明度, 方便跟踪漏洞和检查许可证合规性. 从业者预测, 未来将形成以 SBOM 为中

心的生态系统. 然而, 目前 SBOM 的生成和分发机制不是很成熟, 究其根本是缺乏 SBOM 的深入研究. 当前针对 SBOM 的研究聚焦在现状, 工具和应用上, 缺乏理论化, 体系化的深入探讨. 本文从背景, 概念, 生成技术, 工具, 应用, 挑战等方面综述 SBOM, 并提出 SBOM+的概念, 旨在为 SBOM 和相关的从业人员提供支撑.

References:

[1] Wikipedia. Value chain. 2024. https://en.wikipedia.org/wiki/Value_chain

[2] Holdsworth J. Software Process Design. New York: McGraw-Hill, 1995.

[3] He XX, Zhang YQ, Liu QX. Survey of software supply chain security. Journal of Cyber Security, 2020, 5(1): 57–73 (in Chinese with English abstract). [doi: 10.19363/J.cnki.cn10-1380/tn.2020.01.06]

[4] Liang GY, Wu YJ, Wu JZ, Zhao C. Open source software supply chain for reliability assurance of operating systems. Ruan Jian Xue Bao/Journal of Software, 2020, 31(10): 3056–3073 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6070.htm [doi: 10.13328/j.cnki.jos.006070]

[5] Ji SL, Wang QY, Chen AY, Zhao BB, Ye T, Zhang XH, Wu JZ, Li Y, Yin JW, Wu YJ. Survey on open-source software supply chain security. Ruan Jian Xue Bao/Journal of Software, 2023, 34(3): 1330–1364 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6717.htm [doi: 10.13328/j.cnki.jos.006717]

[6] Linskens A. Introducing our 9th annual state of the software supply chain report. 2023. https://www.sonatype.com/blog/introducing-our-9th-annual-state-of-the-software-supply-chain-report

[7] Major progress has been made in the construction of critical infrastructure for open source software supply chain with the release of “sourcegraph 3.0”. 2023 (in Chinese). http://www.iscas.ac.cn/tpxw2016/202308/t20230823_6865504.html

[8] Constantin L. Developer sabotages own npm module prompting open-source supply chain security questions. 2022. https://www.csoonline.com/article/572327/developer-sabotages-own-npm-module-prompting-open-source-supply-chain-security-questions.html

[9] Constantin L. SolarWinds attack explained: And why it was so hard to detect. 2020. https://www.csoonline.com/article/570191/solarwinds-supply-chain-attack-explained-why-organizations-were-not-prepared.html

[10] Moss S. Supply chain attack on SolarWinds used to breach us government agencies. 2020. https://www.datacenterdynamics.com/en/news/supply-chain-attack-solarwinds-used-breach-us-government-agencies/

[11] Hill M. The Apache Log4j vulnerabilities: A timeline. 2022. https://www.csoonline.com/article/571797/the-apache-log4j-vulnerabilities-a-timeline.html

[12] Wikipedia. Jacobsen v. katzer. 2024. https://en.wikipedia.org/wiki/Jacobsen_v._Katzer

[13] The White House. Executive order on improving the nation’s cybersecurity. 2021. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

[14] Royce ER. Cyber supply chain management and transparency act of 2014. 2014. https://www.congress.gov/bill/113th-congress/house-bill/5793/all-actions

[15] National Telecommunications and Information Administration. NTIA software component transparency. 2021. https://www.ntia.gov/other-publication/2021/ntia-software-component-transparency

[16] National Telecommunications and Information Administration. Software component transparency: Healthcare proof of concept report. 2019. https://www.ntia.gov/sites/default/files/publications/ntia_sbom_healthcare_poc_report_2019_1001_0.pdf

[17] NTIA Multistakeholder Process on Software Component Transparency Standards and Formats Working Group. Survey of existing SBOM formats and standards. 2019. https://www.ntia.gov/sites/default/files/publications/ntia_sbom_formats_and_standards_whitepaper_-_version_20191025_0.pdf

[18] NTIA Multistakeholder Process on Software Component Transparency use Cases and State of Practice Working Group. Roles and benefits for SBOM across the supply chain. 2019. https://www.ntia.gov/sites/default/files/publications/ntia_sbom_use_cases_roles_benefits-nov2019_0.pdf

[19] NTIA Multistakeholder Process on Software Component Transparency Framing Working Group. Framing software component transparency: Establishing a common software bill of material (SBOM). 2019. https://www.ntia.gov/sites/default/files/publications/framingsbom_20191112_0.pdf

[20] National Telecommunications and Information Administration. The minimum elements for a software bill of materials (SBOM). 2021. https://www.ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom

[21] Young SD. Enhancing the security of the software supply chain through secure software development practices. 2022. https://www.whitehouse.gov/wp-content/uploads/2022/09/m-22-18.pdf

[22] European Commission. Regulation of the European parliament and of the council on horizontal cybersecurity requirements for products

with digital elements and amending regulation (EU) 2019/1020. 2022. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex%3A52022PC0454

[23] Development plan for the software and information technology services industry in the 14th five-year plan. 2021 (in Chinese). https://www.gov.cn/zhengce/zhengceku/2021-12/01/content_5655205.htm

[24] The 14th five-year plan for national informatization. 2021 (in Chinese). https://www.gov.cn/xinwen/2021-12/28/content_5664872.htm

[25] CAICT publishes Practical guide to software bill of materials (with full text). 2022 (in Chinese). https://www.secrss.com/articles/43238

[26] Implementation plan for guiding engineering of new industry standardization (2023–2035). 2023 (in Chinese). https://www.gov.cn/zhengce/zhengceku/202308/content_6899527.htm

[27] Cybersecurity technology—Security requirements for software supply chain. 2024 (in Chinese). https://openstd.samr.gov.cn/bzgk/gb/newGbInfo?hcno=94FEB278E715BD48566C48804F1A56EC

[28] Cybersecurity technology—Evaluation method for open source code security of software products. 2024 (in Chinese). https://openstd.samr.gov.cn/bzgk/gb/newGbInfo?hcno=CC2F9496C58572E21E60A58F149F294C

[29] Barack Obama. National strategy for global supply chain security. 2012. https://www.presidency.ucsb.edu/documents/press-release-national-strategy-for-global-supply-chain-security

[30] Computer Security Resource Center. Cybersecurity supply chain risk management. 2016. https://csrc.nist.gov/Projects/cyber-supply-chain-risk-management

[31] Peters GC. Securing open source software act of 2023. 2023. https://www.congress.gov/bill/118th-congress/senate-bill/917

[32] Cybersecurity and Infrastructure Security Agency. CISA open source software security roadmap. 2023. https://www.cisa.gov/resources-tools/resources/cisa-open-source-software-security-roadmap

[33] Cybersecurity and Infrastructure Security Agency. CISA, NSA, and partners release new guidance on securing the software supply chain. 2023. https://www.cisa.gov/news-events/alerts/2023/11/09/cisa-nsa-and-partners-release-new-guidance-securing-software-supply-chain

[34] European Commission. Open source software strategy. 2020. https://commission.europa.eu/about-european-commission/departments-and-executive-agencies/digital-services/open-source-software-strategy_en

[35] European Union Agency for Cybersecurity. Threat landscape for supply chain attacks. 2021. https://www.enisa.europa.eu/publications/threat-landscape-for-supply-chain-attacks

[36] Canadian Centre for Cyber Security. Protecting your organization from software supply chain threats. 2023. https://www.cyber.gc.ca/en/guidance/protecting-your-organization-software-supply-chain-threats-itsm10071

[37] Jaatun LA, Sørlien SM, Borgaonkar R, Taylor S, Jaatun MG. Software bill of materials in critical infrastructure. In: Proc. of the 2023 IEEE Int'l Conf. on Cloud Computing Technology and Science (CloudCom). Naples: IEEE, 2023. 319–324. [doi: 10.1109/CloudCom59040.2023.00059]

[38] Nocera S, Romano S, Di Penta M, Francese R, Scanniello G. Software bill of materials adoption: A mining study from GitHub. In: Proc. of the 2023 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Bogotá: IEEE, 2023. 39–49. [doi: 10.1109/ICSME58846.2023.00016]

[39] von Stockhausen HM, Rose M. Continuous security patch delivery and risk management for medical devices. In: Proc. of the 2020 IEEE Int'l Conf. on Software Architecture Companion (ICSA-C). Salvador: IEEE, 2020. 204–209. [doi: 10.1109/ICSA-C50368.2020.00043]

[40] Caven PJ, Gopavaram SR, Camp LJ. Integrating human intelligence to bypass information asymmetry in procurement decision-making. In: Proc. of the 2022 IEEE Military Communications Conf. (MILCOM). Rockville: IEEE, 2022. 687–692. [doi: 10.1109/MILCOM55135.2022.10017736]

[41] OConnor TJ, Jessee D, Campos D. Towards examining the security cost of inexpensive smart home IoT devices. In: Proc. of the 47th IEEE Annual Computers, Software, and Applications Conf. (COMPSAC). Torino: IEEE, 2023. 1293–1298. [doi: 10.1109/COMPSAC57700.2023.00196]

[42] Kim H, Kim KH. Deep learning-based SBOM defect detection for medical devices. In: Proc. of the 2024 Int'l Conf. on Artificial Intelligence in Information and Communication (ICAIIC). Osaka: IEEE, 2024. 47–51. [doi: 10.1109/ICAIIC60209.2024.10463483]

[43] Mounesan M, Siadati H, Jafarikhah S. Exploring the threat of software supply chain attacks on containerized applications. In: Proc. of the 16th Int'l Conf. on Security of Information and Networks (SIN). Rajasthan: IEEE, 2023. 1–8. [doi: 10.1109/SIN60469.2023.10474901]

[44] Kawaguchi N, Hart C. On the deployment control and runtime monitoring of containers based on consumer side SBOMs. In: Proc. of the 21st Consumer Communications & Networking Conf. (CCNC). Las Vegas: IEEE, 2024. 1022–1025. [doi: 10.1109/CCNC51664.2024.10454654]

[45] Bandara E, Shetty S, Mukkamala R, Rahman A, Foytik P, Liang XP, De Zoysa K, Keong NW. DevSec-GPT—Generative-AI (with custom-trained meta's Llama2 LLM), blockchain, NFT and PBOM enabled cloud native container vulnerability management and pipeline verification platform. In: Proc. of the 2024 IEEE Cloud Summit. Washington: IEEE, 2024. 28–35. [doi: 10.1109/Cloud-Summit61220.2024.00012]

[46] Foster R, Priest Z, Cutshaw M. Infrastructure expression for codified cyber attack surfaces and automated applicability. In: 2021 Resilience Week (RWS). Salt Lake City: IEEE, 2021. 1–4. [doi: 10.1109/RWS52686.2021.9611807]

[47] Hyeon DE, Park JH, Youm HY. A secure firmware and software update model based on blockchains for Internet of Things devices using SBOM. In: Proc. of the 18th Asia Joint Conf. on Information Security (AsiaJCIS). Koganei: IEEE, 2023. 53–58. [doi: 10.1109/AsiaJCIS60284.2023.00019]

[48] Wu W, Wang P, Zhao L, Jiang W. An intelligent security detection and response scheme based on SBOM for securing IoT terminal devices. In: Proc. of the 11th Int'l Conf. on Information, Communication and Networks (ICICN). Xi'an: IEEE, 2023. 391–398. [doi: 10.1109/ICICN59530.2023.10393435]

[49] Crawford A, Yakubovich E, Szumski R. Enforcing SBOMs through the Linux kernel with eBPF and IMA. In: Proc. of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED). Copenhagen: ACM, 2023. 77–78. [doi: 10.1145/3605770.3625206]

[50] Sun JM, Chen JS, Xing ZC, Lu QH, Xu XW, Zhu LM. Where is it? Tracing the vulnerability-relevant files from vulnerability reports. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 200. [doi: 10.1145/3597503.3639202]

[51] O'Donoghue E, Reinhold AM, Izurieta C. Assessing security risks of software supply chains using software bill of materials. In: Proc. of the 2024 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering—Companion (SANER-C). Rovaniemi: IEEE, 2024. 134–140. [doi: 10.1109/SANER-C62648.2024.00023]

[52] Xia BM, Zhang DW, Liu Y, Lu QH, Xing ZC, Zhu LM. Trust in software supply chains: Blockchain-enabled SBOM and the AIBOM future. arXiv:2307.02088, 2024.

[53] Kishimoto R, Kanda T, Manabe Y, Inoue K, Higo Y. Osmy: A tool for periodic software vulnerability assessment and file integrity verification using SPDX documents. In: Proc. of the 2024 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Rovaniemi: IEEE, 2024. 445–449. [doi: 10.1109/SANER60148.2024.00052]

[54] Kloeg B, Ding AY, Pellegrom S, Zhauniarovich Y. Charting the path to SBOM adoption: A business stakeholder-centric approach. In: Proc. of the 19th ACM Asia Conf. on Computer and Communications Security. Singapore: ACM, 2024. 1770–1783. [doi: 10.1145/3634737.3637659]

[55] 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 Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 2630–2642. [doi: 10.1109/ICSE48619.2023.00219]

[56] Chaora A, Ensmenger NL, Camp LJ. Discourse, challenges, and prospects around the adoption and dissemination of software bills of materials (SBOMs). In: Proc. of the 2023 IEEE Int'l Symp. on Technology and Society (ISTAS). Swansea: IEEE, 2023. 1–4. [doi: 10.1109/ISTAS57930.2023.10305922]

[57] Zahan N, Lin E, Tamanna M, Enck W, Williams L. Software bills of materials are required. Are we there yet? IEEE Security & Privacy, 2023, 21(2): 82–88. [doi: 10.1109/MSEC.2023.3237100]

[58] Bi TT, Xia BM, Xing ZC, Lu QH, Zhu LM. On the way to SBOMs: Investigating design issues and solutions in practice. ACM Trans. on Software Engineering and Methodology, 2024, 33(6): 149. [doi: 10.1145/3654442]

[59] Stalnaker T, Wintersgill N, Chaparro O, Di Penta M, German DM, Poshyvanyk D. BOMs away! Inside the minds of stakeholders: A comprehensive study of bills of materials for software systems. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Lisbon: ACM, 2024. 44. [doi: 10.1145/3597503.3623347]

[60] Torres-Arias S, Geer D, Meyers JS. A viewpoint on knowing software: Bill of materials quality when you see it. IEEE Security & Privacy, 2023, 21(6): 50–54. [doi: 10.1109/MSEC.2023.3315887]

[61] Balliu M, Baudry B, Bobadilla S, Ekstedt M, Monperrus M, Ron J, Sharma A, Skoglund G, Soto-Valero C, Wittlinger M. Challenges of producing software bill of materials for Java. IEEE Security & Privacy, 2023, 21(6): 12–23. [doi: 10.1109/MSEC.2023.3302956]

[62] Bifolco D, Nocera S, Romano S, Di Penta M, Francese R, Scanniello G. On the accuracy of GitHub's dependency graph. In: Proc. of the 28th Int'l Conf. on Evaluation and Assessment in Software Engineering (EASE). Salerno: ACM, 2024. 242–251. [doi: 10.1145/3661167.3661175]

[63] Rabbi MF, Champa AI, Nachuma C, Zibran MF. SBOM generation tools under microscope: A focus on the npm ecosystem. In: Proc. of the 39th ACM/SIGAPP Symp. on Applied Computing (SAC). Avila: ACM, 2024. 1233–1241. [doi: 10.1145/3605098.3635927]

[64] Mirakhorli M, Garcia D, Dillon S, Laporte K, Morrison M, Lu H, Koscinski V, Enoch C. A landscape study of open source and proprietary tools for software bill of materials (SBOM). arXiv:2402.11151, 2024.

[65] Halbritter A, Merli D. Accuracy evaluation of SBOM tools for Web applications and system-level software. In: Proc. of the 19th Int'l Conf. on Availability, Reliability and Security. Vienna: ACM, 2024. 55. [doi: 10.1145/3664476.3670926]

[66] Dalia G, Visaggio CA, Di Sorbo A, Canfora G. SBOM ouverture: What we need and what we have. In: Proc. of the 19th Int'l Conf. on Availability, Reliability and Security. Vienna: ACM, 2024. 116. [doi: 10.1145/3664476.3669975]

[67] NTIA Multistakeholder Process on Software Component Transparency Framing Working Group. Framing software component transparency: Establishing a common software bill of materials (SBOM). 2021. https://www.ntia.gov/sites/default/files/publications/ntia_sbom_framing_2nd_edition_20211021_0.pdf

[68] NTIA Multistakeholder Process on Software Component Transparency Framing Working Group. Software identification challenges and guidance. 2021. https://www.ntia.gov/files/ntia/publications/ntia_sbom_software_identity-2021mar30.pdf

[69] NTIA Multistakeholder Process on Software Component Transparency Standards and Formats Working Group. Survey of existing SBOM formats and standards. 2021. https://www.ntia.gov/sites/default/files/publications/sbom_formats_survey-version-2021_0.pdf

[70] Linux Foundation Projects Site. System Package Data Exchange (SPDX®). 2024. https://spdx.dev/

[71] SPDX. Composition of an SPDX document. 2022. https://spdx.github.io/spdx-spec/v2.3/composition-of-an-SPDX-document/

[72] OWASP. CycloneDX. https://cyclonedx.org/

[73] CycloneDX. Specification overview. 2023. https://cyclonedx.org/specification/overview/

[74] NVD. SWID. https://nvd.nist.gov/products/swid

[75] Waltermire D, Cheikes BA, Feldman L, Witte G. Guidelines for the creation of interoperable software identification (SWID) tags. Gaithersburg: National Institute of Standards and Technology, 2016. [doi: 10.6028/NIST.IR.8060]

[76] NTIA Software Transparency Healthcare POC. How-to guide for SBOM generation. 2021. https://www.ntia.gov/sites/default/files/publications/howto_guide_for_sbom_generation_v1_0.pdf

[77] NTIA Formats and Tooling Working Group. Software consumers playbook: SBOM acquisition, management, and use. 2021. https://www.ntia.gov/sites/default/files/publications/software_consumers_sbom_acquisition_management_and_use_-_final_0.pdf

[78] NTIA Formats and Tooling Working Group. Software suppliers playbook: SBOM production and provision. 2021. https://www.ntia.gov/sites/default/files/publications/software_suppliers_sbom_production_and_provision_-_final_0.pdf

[79] IEEE Spectrum. The top programming languages 2024. 2024. https://spectrum.ieee.org/top-programming-languages-2024

[80] Pandas. Pandas. 2024. https://github.com/pandas-dev/pandas

[81] NumPy. NumPy. 2024. https://github.com/numpy/numpy

[82] 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]

[83] Zhao LD, Chen S, Xu ZZ, Liu CW, Zhang L, Wu JH, Sun J, Liu Y. Software composition analysis for vulnerability detection: An empirical study on Java projects. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 960–972. [doi: 10.1145/3611643.3616299]

[84] Sharma P, Shi ZP, Şimşek Ş, Starobinski D, Medina DS. Understanding similarities and differences between software composition analysis tools. IEEE Security & Privacy, 2024: 2–13. [doi: 10.1109/MSEC.2024.3410957]

[85] Imtiaz N, Thorn S, Williams L. A comparative study of vulnerability reporting by software composition analysis tools. In: Proc. of the 15th ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement (ESEM). Bari: ACM, 2021. 5. [doi: 10.1145/3475716.3475769]

[86] Horton E, Parnin C. DockerizeMe: Automatic inference of environment dependencies for python code snippets. In: Proc. of the 41st Int'l Conf. on Software Engineering (ICSE). Montréal: IEEE, 2019. 328–338. [doi: 10.1109/ICSE.2019.00047]

[87] Horton E, Parnin C. V2: Fast detection of configuration drift in Python. arXiv:1909.06251, 2019.

[88] 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]

[89] Ye HJ, Zhou JH, Chen W, Zhu JX, Wu GQ, Wei J. DockerGen: A knowledge graph based approach for software containerization. In: Proc. of the 45th IEEE Annual Computers, Software, and Applications Conf. (COMPSAC). Madrid: IEEE, 2021. 986–991. [doi: 10.1109/COMPSAC51774.2021.00133]

[90] Wang JW, Li L, Zeller A. Restoring execution environments of jupyter notebooks. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on

Software Engineering (ICSE). Madrid: IEEE, 2021. 1622–1633. [doi: 10.1109/ICSE43902.2021.00144]

[91] Ye HJ, Chen W, Dou WS, Wu GQ, Wei J. Knowledge-based environment dependency inference for Python programs. In: Proc. of the 44th Int'l Conf. on Software Engineering (ICSE). Pittsburgh: ACM, 2022. 1245–1256. [doi: 10.1145/3510003.3510127]

[92] Cheng W, Zhu XR, Hu W. Conflict-aware inference of Python compatible runtime environments with domain knowledge graph. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 451–461. [doi: 10.1145/3510003.3510078]

[93] Cheng W, Hu W, Ma XX. Revisiting knowledge-based inference of Python runtime environments: A realistic and adaptive approach. IEEE Trans. on Software Engineering, 2024, 50(2): 258–279. [doi: 10.1109/TSE.2023.3346474]

[94] 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]

[95] 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]

[96] Wu JH, Xu ZZ, Tang W, Zhang LY, 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 Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 270–282. [doi: 10.1109/ICSE48619.2023.00034]

[97] Chen QY, Li SP, Yan M, Xia X. Code clone detection: A literature review. Ruan Jian Xue Bao/Journal of Software, 2019, 30(4): 962–980 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5711.htm [doi: 10.13328/j.cnki.jos.005711]

[98] Sun XJ, Wei Q, Wang YS, Du J. Survey of code similarity detection technology. Journal of Computer Applications, 2024, 44(4): 1248–1258 (in Chinese with English abstract). [doi: 10.11772/j.issn.1001-9081.2023040551]

[99] Wu YM, Feng SY, Zou DQ, Jin H. Detecting semantic code clones by building AST-based Markov chains model. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 34. [doi: 10.1145/3551349.3560426]

[100] Hu YT, Fang YL, Sun YF, Jia YR, Wu YM, Zou DQ, Jin H. Code2Img: Tree-based image transformation for scalable code clone detection. IEEE Trans. on Software Engineering, 2023, 49(9): 4429–4442. [doi: 10.1109/TSE.2023.3295801]

[101] Hu TC, Xu ZJ, Fang YL, Wu YM, Yuan B, Zou DQ, Jin H. Fine-grained code clone detection with block-based splitting of abstract syntax tree. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 89–100. [doi: 10.1145/3597926.3598040]

[102] Zou DQ, Feng SY, Wu YM, Suo WQ, Jin H. Tritor: Detecting semantic code clones by building social network-based triads model. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 771–783. [doi: 10.1145/3611643.3616354]

[103] Mehrotra N, Sharma A, Jindal A, Purandare R. Improving cross-language code clone detection via code representation learning and graph neural networks. IEEE Trans. on Software Engineering, 2023, 49(11): 4846–4868. [doi: 10.1109/TSE.2023.3311796]

[104] Alhazami EA, Sheneamer AM. Graph-of-code: Semantic clone detection using graph fingerprints. IEEE Trans. on Software Engineering, 2023, 49(8): 3972–3988. [doi: 10.1109/TSE.2023.3276780]

[105] Liu JH, Zeng J, Wang X, Liang ZK. Learning graph-based code representations for source-level functional similarity detection. In: Proc. of the 45th Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 345–357. [doi: 10.1109/ICSE48619.2023.00040]

[106] Wang YK, Ye YH, Wu YM, Zhang WW, Xue YX, Liu Y. Comparison and evaluation of clone detection techniques with different code representations. In: Proc. of the 45th Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 332–344. [doi: 10.1109/ICSE48619.2023.00039]

[107] Li J, Tao CY, Jin Z, Liu F, Li J, Li G. ZC^3 : Zero-shot cross-language code clone detection. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Luxembourg: IEEE, 2023. 875–887. [doi: 10.1109/ASE56229.2023.00210]

[108] Shan JJ, Dou SH, Wu YM, Wu HR, Liu Y. Gitor: Scalable code clone detection by building global sample graph. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 784–795. [doi: 10.1145/3611643.3616371]

[109] Xu ZW, Qiang SH, Song DH, Zhou M, Wan H, Zhao XB, Luo P, Zhang HY. DSFM: Enhancing functional code clone detection with deep subtree interactions. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 221. [doi: 10.1145/3597503.3639215]

[110] Wu YM, Feng SY, Suo WQ, Zou DQ, Jin H. Goner: Building tree-based N-gram-like model for semantic code clone detection. IEEE Trans. on Reliability, 2024, 73(2): 1310–1324. [doi: 10.1109/TR.2023.3312294]

[111] Feng SY, Suo WQ, Wu YM, Zou DQ, Liu Y, Jin H. Machine learning is all you need: A simple token-based approach for effective code clone detection. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 222. [doi: 10.1145/3597503.

3639114]

[112] Xu HY, Pei XH, Su X, You S, Xu C. TCNAS: Transformer architecture evolving in code clone detection. In: Proc. of the 2024 IEEE Int'l Conf. on Acoustics, Speech and Signal Processing (ICASSP). Seoul: IEEE, 2024. 5745–5749. [doi: 10.1109/ICASSP48485.2024.10445958]

[113] Li HR, Wang SQ, Quan WH, Gong XL, Su HY, Zhang J. Prism: Decomposing program semantics for code clone detection through compilation. In: Proc. of the 46th IEEE/CVF Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 217. [doi: 10.1145/3597503.3639129]

[114] Zhang LH, Luo SL, Pan LM, Wu ZT, Gong K. FSD-CLCD: Functional semantic distillation graph learning for cross-language code clone detection. Engineering Applications of Artificial Intelligence, 2024, 133: 108199. [doi: 10.1016/j.engappai.2024.108199]

[115] Du YK, Ma TF, Wu LF, Zhang XH, Ji SL. AdaCCD: Adaptive semantic contrasts discovery based cross lingual adaptation for code clone detection. In: Proc. of the 38th AAAI Conf. on Artificial Intelligence. Vancouver: AAAI 2024. 17942–17950. [doi: 10.1609/aaai.v38i16.29749]

[116] Yuan DW, Fang S, Zhang T, Xu Z, Luo XP. Java code clone detection by exploiting semantic and syntax information from intermediate code-based graph. IEEE Trans. on Reliability, 2023, 72(2): 511–526. [doi: 10.1109/TR.2022.3176922]

[117] Zhan X, Liu TM, Fan LL, Li L, Chen S, Luo XP, Liu Y. Research on third-party libraries in Android APPs: A taxonomy and systematic literature review. IEEE Trans. on Software Engineering, 2022, 48(10): 4181–4213. [doi: 10.1109/TSE.2021.3114381]

[118] Ishio T, Kula RG, Kanda T, German DM, Inoue K. Software ingredients: Detection of third-party component reuse in java software release. In: Proc. of the 13th Int'l Conf. on Mining Software Repositories. Austin: ACM, 2016. 339–350. [doi: 10.1145/2901739.2901773]

[119] Tang W, Chen D, Luo P. BCFinder: A lightweight and platform-independent tool to find third-party components in binaries. In: Proc. of the 25th Asia-Pacific Software Engineering Conf. (APSEC). Nara: IEEE, 2018. 288–297. [doi: 10.1109/APSEC.2018.00043]

[120] Zhang D, Luo P, Tang W, Zhou M. OSLDetector: Identifying open-source libraries through binary analysis. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Virtual Event: ACM, 2021. 1312–1315. [doi: 10.1145/3324884.3415303]

[121] Zhan X, Fan LL, Chen S, We F, Liu TM, Luo XP, Liu Y. ATVHunter: Reliable version detection of third-party libraries for vulnerability identification in Android applications. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 1695–1707. [doi: 10.1109/ICSE43902.2021.00150]

[122] Yang C, Xu ZZ, Chen HX, Liu Y, Gong XR, Liu BX. ModX: Binary level partially imported third-party library detection via program modularization and semantic matching. In: Proc. of the 44th Int'l Conf. on Software Engineering (ICSE). Pittsburgh: ACM, 2022. 1393–1405. [doi: 10.1145/3510003.3510627]

[123] Tang W, Wang YL, Zhang HY, Han S, Luo P, Zhang DM. LibDB: An effective and efficient framework for detecting third-party libraries in binaries. In: Proc. of the 19th Int'l Conf. on Mining Software Repositories. Pittsburgh: ACM, 2022. 423–434. [doi: 10.1145/3524842.3528442]

[124] Wu YF, Sun C, Zeng DR, Tan G, Ma SQ, Wang PC. LibScan: Towards more precise third-party library identification for Android applications. In: Proc. of the 32nd USENIX Conf. on Security Symp. Anaheim: USENIX Association, 2023. 3385–3402.

[125] Li SY, Wang YP, Dong CP, Yang SG, Li H, Sun H, Lang Z, Chen ZX, Wang WJ, Zhu HS, Sun LM. LibAM: An area matching framework for detecting third-party libraries in binaries. ACM Trans. on Software Engineering and Methodology, 2024, 33(2): 52. [doi: 10.1145/3625294]

[126] Domínguez-Álvarez D, De La Cruz A, Gorla A, Caballero J. LibKit: Detecting third-party libraries in iOS APPs. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 1407–1418. [doi: 10.1145/3611643.3616344]

[127] Huang JJ, Xue B, Jiang JS, You W, Liang B, Wu JZ, Wu YJ. Scalably detecting third-party Android libraries with two-stage bloom filtering. IEEE Trans. on Software Engineering, 2023, 49(4): 2272–2284. [doi: 10.1109/TSE.2022.3215628]

[128] Almanee S, Ünal A, Payer M, Garcia J. Too quiet in the library: An empirical study of security updates in Android APPs' native code. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering: Companion Proceedings. Madrid: IEEE, 2021. 170. [doi: 10.1109/ICSE-Companion52605.2021.00072]

[129] Zhang YH, Wang JC, Huang HX, Zhang YQ, Liu P. Understanding and conquering the difficulties in identifying third-party libraries from millions of Android APPs. IEEE Trans. on Big Data, 2022, 8(6): 1511–1523. [doi: 10.1109/TBDATA.2021.3093244]

[130] Zhao BB, Ji SL, Xu JC, Tian Y, Wei QY, Wang QY, Lyu C, Zhang XH, Lin CT, Wu JZ, Beyah R. One bad apple spoils the barrel: Understanding the security risks introduced by third-party components in IoT firmware. IEEE Trans. on Dependable and Secure Computing, 2024, 21(3): 1372–1389. [doi: 10.1109/TDSC.2023.3279846]

[131] Wang Z, Zhang HT, Guo JD, Xi LL, Tambadou S, Zuo F, Li H, Hu Y. Precise and efficient third-party Java libraries identification tool

for collaborative software. In: Proc. of the 27th Int'l Conf. on Computer Supported Cooperative Work in Design (CSCWD). Tianjin: IEEE, 2024. 2541–2546. [doi: 10.1109/CSCWD61410.2024.10580266]

[132] Cui HJ, Meng GZ, Li YQ, Li YJ, Zhang Y, Sun JY, Zhu DL, Wang WP. LibHunter: An unsupervised approach for third-party library detection without prior knowledge. In: Proc. of the 2022 IEEE Symp. on Computers and Communications (ISCC). Rhodes: IEEE, 2022. 1–7. [doi: 10.1109/ISCC55528.2022.9912796]

[133] Clementi L, Papadopoulos P. POSTER: Fingerprinting application dependencies. In: Proc. of the 2014 IEEE Int'l Conf. on Cluster Computing (CLUSTER). Madrid: IEEE, 2014. 288–289. [doi: 10.1109/CLUSTER.2014.6968762]

[134] Yang SG, Cheng L, Zeng YC, Lang Z, Zhu HS, Shi ZQ. Asteria: Deep learning-based AST-encoding for cross-platform binary code similarity detection. In: Proc. of the 51st Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). Taipei: IEEE, 2021. 224–236. [doi: 10.1109/DSN48987.2021.00036]

[135] Benoit T, Marion JY, Bardin S. Scalable program clone search through spectral analysis. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 808–820. [doi: 10.1145/3611643.3616279]

[136] Qasem A, Debbabi M, Lebel B, Kassouf M. Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures. In: Proc. of the 2023 Asia Conf. on Computer and Communications Security. Melbourne: ACM, 2023. 443–456. [doi: 10.1145/3579856.3582818]

[137] 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]

[138] Li LT, Ding SHH, Charland P. GenTAL: Generative denoising skip-gram Transformer for unsupervised binary code similarity detection. In: Proc. of the 2023 Int'l Joint Conf. on Neural Networks (IJCNN). Gold Coast: IEEE, 2023. 1–8. [doi: 10.1109/IJCNN54540.2023.10191550]

[139] Jiang S, Fu C, He S, Lv JQ, Han LS, Hu H. BinCola: Diversity-sensitive contrastive learning for binary code similarity detection. IEEE Trans. on Software Engineering, 2024, 50(10): 2485–2497. [doi: 10.1109/TSE.2024.3411072]

[140] Zhang YT, Fang BX, Xiong ZH, Wang YH, Liu YW, Zheng C, Zhang QN. A semantics-based approach on binary function similarity detection. IEEE Internet of Things Journal, 2024, 11(15): 25910–25924. [doi: 10.1109/JIOT.2024.3389014]

[141] Xu XZ, Feng SW, Ye YP, Shen GY, Su ZA, Cheng SY, Tao GH, Shi QK, Zhang Z, Zhang XY. Improving binary code similarity Transformer models by semantics-driven instruction deemphasis. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 1106–1118. [doi: 10.1145/3597926.3598121]

[142] Wong WK, Wang HJ, Li ZJ, Wang S. BinAug: Enhancing binary similarity analysis with low-cost input repairing. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 7. [doi: 10.1145/3597503.3623328]

[143] FOSSCheck. 2024. https://www.fosscheck.com/index

[144] Software Component Analysis SCA. 2024 (in Chinese). https://www.yun88.com/product/2608.html

[145] Murphy Security. 2024 (in Chinese). https://www.murphysec.com/

[146] NTIA Formats and Tooling Working Group. Tooling ecosystem working with SPDX. 2024. https://docs.google.com/document/d/1A1jFIYihB-IyT0gv7E_KoSjLbwNGmu_wOXBs6siemXA/edit?usp=embed_facebook

[147] NTIA Formats and Tooling Working Group. Tooling ecosystem working with CycloneDX. 2024. https://docs.google.com/document/d/1biwYXrtoRc_LF7Pw10TO2TGIhlM6jwkDG23nc9M_RiE/edit?usp=embed_facebook

[148] NTIA Formats and Tooling Working Group. Tooling ecosystem working with SWID. 2024. https://docs.google.com/document/d/1oebYvHcOhtMG8Uhnd5he0l_vhty7MsTjp6fYCOwUmwM/edit?pli=1&usp=embed_facebook

[149] Haddad I. An open guide to evaluating software composition analysis tools. 2020. https://www.linuxfoundation.org/resources/publications/an-open-guide-to-evaluating-software-composition-analysis-tools

[150] Cao YL, Chen L, Ma WWY, Li YH, Zhou YM, Wang LZ. Towards better dependency management: A first look at dependency smells in python projects. IEEE Trans. on Software Engineering, 2023, 49(4): 1741–1765. [doi: 10.1109/TSE.2022.3191353]

[151] Wu YL, Yu ZL, Wen M, Li Q, Zou DQ, Jin H. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 1046–1058. [doi: 10.1109/ICSE48619.2023.00095]

附中文参考文献:

[3] 何熙巽, 张玉清, 刘奇旭. 软件供应链安全综述. 信息安全学报, 2020, 5(1): 57–73. [doi: 10.19363/J.cnki.cn10-1380/tn.2020.01.06]

[4] 梁冠宇, 武延军, 吴敬征, 赵琛. 面向操作系统可靠性保障的开源软件供应链. 软件学报, 2020, 31(10): 3056–3073. http://www.jos.org.cn/1000-9825/6070.htm [doi: 10.13328/j.cnki.jos.006070]
[5] 纪守领, 王琴应, 陈安莹, 赵彬彬, 叶童, 张旭鸿, 吴敬征, 李昀, 尹建伟, 武延军. 开源软件供应链安全研究综述. 软件学报, 2023, 34(3): 1330–1364. http://www.jos.org.cn/1000-9825/6717.htm [doi: 10.13328/j.cnki.jos.006717]
[7] “源图 3.0”重磅发布开源软件供应链重大基础设施建设取得重要进展. 2023. http://www.iscas.ac.cn/tpxw2016/202308/t20230823_6865504.html
[23] “十四五”软件和信息技术服务业发展规划. 2021. https://www.gov.cn/zhengce/zhengceku/2021-12/01/content_5655205.htm
[24] “十四五”国家信息化规划. 2021. https://www.gov.cn/xinwen/2021-12/28/content_5664872.htm
[25] 信通院发布《软件物料清单实践指南》(附全文). 2022. https://www.secrss.com/articles/43238
[26] 新产业标准化领航工程实施方案 (2023–2035 年). 2023. https://www.gov.cn/zhengce/zhengceku/202308/content_6899527.htm
[27] 网络安全技术软件供应链安全要求. 2024. https://openstd.samr.gov.cn/bzgk/gb/newGbInfo?hcno=94FEB278E715BD48566C48804F1A56EC
[28] 网络安全技术软件产品开源代码安全评价方法. 2024. https://std.samr.gov.cn/gb/search/gbDetailed?id=173829859DA71AA5E06397BE0A0AA311
[97] 陈秋远, 李善平, 鄢萌, 夏鑫. 代码克隆检测研究进展. 软件学报, 2019, 30(4): 962–980. http://www.jos.org.cn/1000-9825/5711.htm [doi: 10.13328/j.cnki.jos.005711]
[98] 孙祥杰, 魏强, 王奕森, 杜江. 代码相似性检测技术综述. 计算机应用, 2024, 44(4): 1248–1258. [doi: 10.11772/j.issn.1001-9081.2023040551]
[144] 雳鉴 SCA 软件成分分析系统. 2024. https://www.yun88.com/product/2608.html
[145] 墨菲安全. 2024. https://www.murphysec.com/

孙泽雨(2000－), 女, 硕士, 主要研究领域为 SBOM, 软件成分分析.

魏怡琳(1998－), 女, 工程师, CCF 专业会员, 主要研究领域为开源软件供应链, 软件物料清单.

吴敬征(1982－), 男, 博士, 研究员, 博士生导师, CCF 杰出会员, 主要研究领域为开源软件供应链安全, 系统安全, 漏洞挖掘, 操作系统.

罗天悦(1990－), 男, 工程师, CCF 专业会员, 主要研究领域为操作系统安全分析, 代码漏洞挖掘, 人工智能安全.

凌祥(1992－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为智能软件安全.

武延军(1979－), 男, 博士, 研究员, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 机器学习系统软件, 系统安全.