软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

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

开源软件供应链安全研究综述*

纪守领[1], 王琴应[1], 陈安莹[1], 赵彬彬[2], 叶 童[1], 张旭鸿[1], 吴敬征[3], 李 昀[4], 尹建伟[1], 武延军[3]

[1](浙江大学 计算机科学与技术学院,浙江 杭州 310007)

[2](浙江大学滨江研究院,浙江 杭州 310053)

[3](中国科学院软件研究所 智能软件研究中心, 北京 100190)

[4](上海华为技术有限公司,中国 上海 201206)

通讯作者: 武延军, E-mail: yanjun@iscas.ac.cn

摘 要: 随着近年来开源软件的蓬勃发展, 现代化软件的开发和供应模式极大地促进开源软件自身的快速迭代和演进, 也提高了社会效益. 新兴的开源协作的软件开发模式使得软件开发供应流程由较为单一的线条转变为复杂的网络形态. 在盘根错节的开源软件供应关系中, 总体安全风险趋势显著上升, 日益受到学术界和产业界的重视. 本文总结了开源软件供应链的关键环节, 基于近10年的攻击事件总结了开源软件供应链的威胁模型和安全趋势, 并通过对现有安全研究成果的调研分析, 从风险识别和加固防御两个方面总结了开源软件供应链安全的研究现状, 最后对开源软件供应链安全所面临的挑战和未来研究方向进行了展望和总结.

关键词: 开源软件供应链; 软件供应链风险识别; 供应链风险管理; 软件安全加固
中图法分类号: TP311

中文引用格式: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.

英文引用格式: Ji SL, Wang QY, Chen AY, Zhao BB, Ye T, Zhang XH, Wu JZ, Li Y, Yin JW, Wu YJ. State-of-the-Art survey of Open-source Software Supply Chain Security. Ruan Jian Xue Bao/Journal of Software, 2021 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

State-of-the-Art Survey of Open-source Software Supply Chain Security

JI Shou-Ling[1], WANG Qin-Ying[1], CHEN An-Ying[1], ZHAO Bin-Bin[2], YE Tong[1], ZHANG Xu-Hong[1], WU Jing-Zheng[3], LI Yun[4], YIN Jian-Wei[1], WU Yan-Jun[3]

[1](College of Computer Science and Technology, Zhejiang University, Hangzhou 310007, China)

[2](Binjiang Institute of Zhejiang University, Hangzhou 310053, China)

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

[4](Shanghai Huawei Technologies Co., Ltd., Shanghai 201206, China)

Abstract: Software development is changing. Since the Internet allows far-flung development teams to collaboratively create software, open-source software supply chains are becoming more complex and sophisticated. This work tries to define the new open-source software supply chain model and presents a detailed survey of the security issues in the new open-source software supply chain architecture. Various emerging technologies, such as blockchain, machine learning (ML), and continuous fuzzing as solutions to the vulnerabilities in the open-source software supply chain have also been discussed. While many researchers and organizations are have

* 基金项目: 国家自然科学基金项目(U1936215); 浙江省自然科学基金杰出青年项目(LR19F020003)

Foundation item: National Natural Science Foundation of China (U1936215); Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars (LR19F020003)

收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00

CNKI 在线出版时间: 0000-00-00

already proposed new technologies and principles to handle the security issues in this area, proper and more effective solutions remain distant. There are new challenges and opportunities to secure the open-source software supply chain, which are also highlighted in this work.

Key words: open-source software supply chain; software supply chain attack detection; security management; enhancement

随着近年来开源软件(opensource software)的蓬勃发展, 开源软件已经成为现代化信息产业不可或缺的一部分, 也给软件的构成和开发过程带来了巨大的变化. 近年来, 软件开发模式从一个组织、一个厂商单独开发一个产品, 到了依赖部分共享的组件进行开发, 形成一条线性的软件供应链. 现今, 现代化软件开发过程中, 开发人员及企业为了提高开发的效率, 会优先考虑安装相关第三方开源组件的依赖, 并在此基础上进行修改、拓展及改进. 这种开发模式可以避免重复设计开发, 降低研发成本,提高生产力和生产效率, 推动软件系统快速迭代, 形成了开源协作的软件开发新模式. 在开源协作模式下, 软件的功能和体系也越来越复杂, 呈现为更加复杂的网络型供应链形式. 在此背景下, 本文采用开源软件供应链的概念, 围绕现代化软件开发新模式下的安全问题进行深入的研究和分析.

开源软件供应链和我国国民生活、城市建设、工业生产等息息相关, 影响着智能家居、汽车、智能电网等众多关键基础设施的开发和管理, 其安全问题也影响着我国现代化建设和发展. 然而随着开源软件供应链体系越来越庞大, 威胁的渗入点不断增多, 分布在开源软件供应链的各个环节. 同时, 开源软件供应体系呈现全球化的态势, 增加了开源软件供应监管的难度. 根据 Verocode 的研究报告[1]显示, 超过 96%的产业机构在其开发实现的软件应用代码库中使用开源组件, 而近 70%的应用程序都存在开源安全缺陷.

为了构建安全可靠的开源软件供应链, 保障软件供应安全和高质量创新, 一大批来自学术界和工业界的学者探索了开源软件供应链的攻击事件和存在的安全风险, 并且前瞻性地提出了一系列针对开源软件供应链的风险识别和加固防御方法, 涵盖了软件开发、分发、使用等环节. 然而, 由于不同学者所处的研究领域不同, 解决问题的角度不同, 因而构建的威胁模型也不同, 研究的攻击或防御方法也各有侧重,缺乏系统的整理、归纳、总结、分析. 虽然, 有学者在 2019-2020 年对软件供应链的研究工作[2][3]进行了系统的整理, 但大多只关注软件供应链中的某个技术或者环节. 与本文工作最接近的是 He 等人[4]和 Hassijia 等人[5]对软件供应链的安全综述, 但他们的工作没有考虑到开源协作模式下软件供应链新的角色和环节, 主要停留在较为传统的软件开发分发供应关系上, 也缺少对最近新兴的物联网场景以及一些智能技术的分析. 因此, 亟需对现有的工作在开源协作模式下对开源软件供应链的攻击和防御的研究进展进行深入系统的整理和科学归纳、总结、分析, 以便为后续学者了解或研究开源软件供应链安全提供指导.

本文首先介绍了开源软件供应链的模型, 接着对开源软件供应链面临的风险和攻击事件进行总结和归纳, 然后从风险识别和加固防御两个角度对开源软件供应链的安全防御研究进行系统地分析、归纳和总结, 并讨论相关研究的局限性. 最后, 本文探讨了开源软件供应链安全研究所面临的挑战以及未来可行的研究方向.

1 开源软件供应链模型

在研究和定义开源软件安全供应链安全前, 首先需明确开源软件供应链的定义. 学术界和工业界对软件供应链提出了很多定义, He 等人[4]定义软件供应链是一个通过一级或多级软件设计、开发阶段编写软件, 并通过软件交付渠道将软件从软件供应商送往软件用户的系统, Du 等人[6]定义软件供应链是开发软件和组装软件的过程, 包括从源代码到最终软件交付给用户的整个开发、发布、部署和维护过程. 上述软件供应链仅仅停留在传统的软件开发和交付的过程上, 没有考虑到当前开源协作模式的不断发展使得软件的构成和开发过程发生了巨大的变化. 当前的软件开发更加关注高效和协作, 会使用已有的第三方组件实现一些基本功能, 也会有多个参与方共同对软件系统进行扩展和改进. 然而, 在这样的开发模式下, 软件系统的构成也越发复杂: 应用软件将依赖大量的第三方组件, 同时第三方组件又依赖更多的其他第三方组件, 形成依赖嵌套. 如图 1 所示, 以自动化展示包表工具为例, 该应用软件依赖 poi、jxls 等组件, 而 poi 组件又依赖 commons-collection4 和 commons-codec 组件. 图中的组件和应用软件均有多个开发人员共同贡献代码. 因此, 需要对开源软件供

应链模型进行新的定义.

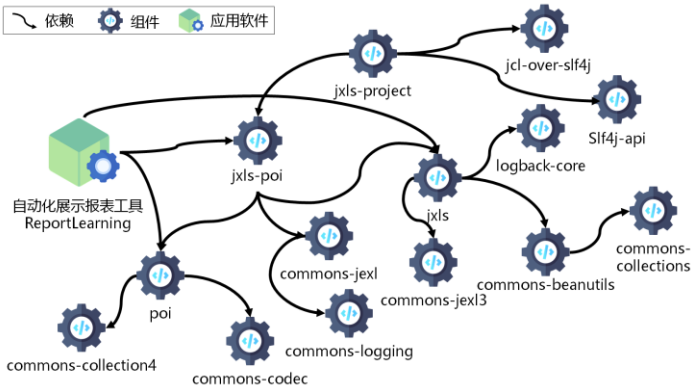

图 1　复杂的组件和应用软件依赖关系图

本文定义开源软件供应链是指开源软件在开发和运行过程中，涉及到的所有开源软件的上游社区、源码包、二进制包、第三方组件分发市场、应用软件分发市场，以及开发者和维护者、社区、基金会等，按照依赖、组合等形成的供应关系网络. 如图 2 所示，本文进一步总结了开源软件供应链的四个关键环节，包括组件开发环节、应用软件开发环节、应用软件分发环节和应用软件使用环节. 下面本文根据四个关键环节来介绍开源软件供应链的模型.

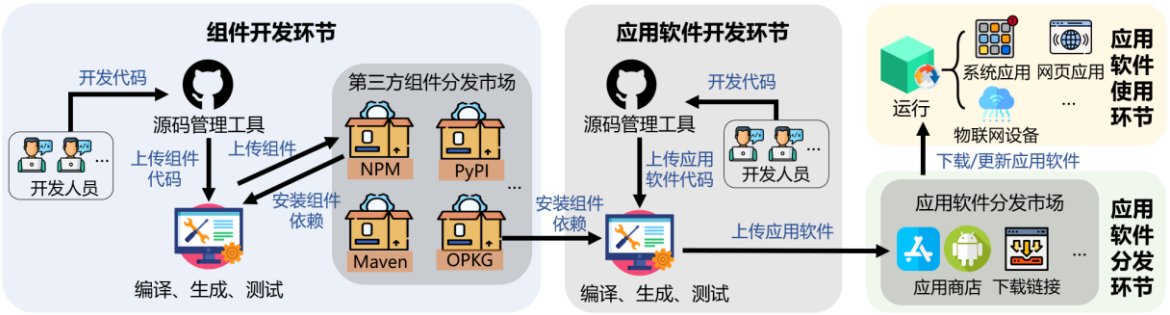

图 2　开源软件供应链主要环节模型

(1)组件开发环节中，多个开发人员协作开发代码，利用源码管理工具(Source Code Management,SCM)管理源代码，通过编译、生成、测试，完成组件的开发,并上传到第三方组件开发市场. 此外，开发人员出于高效协作开发代码的考虑，会从第三方组件分发市场下载第三方组件并安装依赖来完成开发.

其中，流行的源码管理工具包括 GitHub[7]、码云[8]等. 此外，当前第三方组件分发市场数量庞大，包括常见的针对开发语言的包管理器，如 PyPI[9], NPM[10], Maven[11], OPKG[12]和 RubyGem[13]等，也包括新兴的物联网场景下出现的第三方语音技能分发市场、自动化应用程序的组件分发市场等，如亚马逊的语音技能市场[14]、IFTTT 平台[15]等. 第三方组件分发市场需要对第三方组件进行审核管理，开发人员也需要根据开发需求选择相应的第三方组件.

(2)应用软件开发环节中，多个开发人员会从第三方组件分发市场安装组件依赖，和组件开发一样，利用源码管理工具管理源代码，通过编译、生成、测试,完成对应用软件的开发. 其中应用软件指的是可以直接发布给终端用户使用的完整产品,如手机应用、物联网设备和网页应用等.

(3)应用软件分发环节中，应用软件在软件分发市场上被发布. 常见的应用软件分发市场主要是应用商店和软件开发者维护的含有下载连接的软件网站.

(4)应用软件使用环节中, 终端用户将从应用软件分发市场下载软件并使用. 终端用户在使用过程中也可进一步根据软件的迭代对软件进行更新.

2 开源软件供应链的风险分析

开源软件供应链模型的定义为开源软件供应链的安全研究带来了全新的视角. 本章首先总结了开源软件供应链威胁模型, 根据该模型分析供应链各个环节的安全事件, 并总结当前开源软件供应链风险的总体趋势.

2.1 开源软件供应链的威胁模型

已有学者对软件供应链的威胁模型进行了分析, 给出了多种类型的威胁渗入点或攻击向量. Zhou 等人[3]总结软件供应链的威胁主要是两大方面: (1) 攻击者通过伪装和篡改的方式对软件供应链中产出的软件进行污染; (2) 攻击者通过这些被污染的软件获得了在软件运行环境中执行操作的权限, 进一步造成信息泄露等危害. Linux 开源软件基金会(OpenSSF) [16]从本地开发、外部代码贡献、代码中心仓库测试集成、软件包消费等软件供应链阶段分别进行威胁建模, 分析了各环节的 25 种潜在攻击方式. 此威胁模型尚未对攻击方式和影响进行更深层次的探讨. 谷歌公司提出的威胁模型主要基于他们总结的供应链框架即“软件构件的供应链级别(Supply Chain Levels for Software Artifacts, SLSA), 更加关注软件供应链的完整性[17]. 类似地, MITRE 组织[18]提出的供应链缺陷主要涵盖对开发工具、开发环境、公开或私有的源代码仓库等真实攻击对象的操控, 同时将政治、社会、法律等方面的风险因素纳入考虑. 而 Torres-Arias 等人[19]和 Peterson 等人[20]特别研究了开源给软件供应链带来的新兴威胁,包括: 意外引入的开源漏洞、恶意引入的开源漏洞和引入带有恶意逻辑的开源组件. 其中 Torres-Arias 等人还指出了供应链中各个环节的环境被攻陷及供应方信任凭证失窃等风险.

综上研究, 当前学者们已经列举分析了多种多样的软件供应链威胁渗入点, 但缺少系统性的概括, 或缺少某个供应链环节的威胁分析, 或没有考虑到当前开源协作新模式下引入的威胁, 同时也缺少对开源软件供应链攻击面的总结, 缺少对攻击向量和攻击影响的深层讨论. 因此, 本文综合考虑了上述研究和本文建立的开源软件供应链模型, 对威胁模型进行了系统的分析和总结. 如表1所示, 本文从开源软件供应链的环节角度对开源软件供应链进行威胁建模, 总结了攻击目标环节、攻击面和违反的关键安全属性.

表 1 开源软件供应链的威胁模型

环节	攻击面	违反的安全属性
组件和应用软件开发环节	1. 攻击者通过分发市场的缺陷,制作伪装、虚假的第三方组件,或篡改已有的第三方组件,欺骗开发人员下载使用 2. 攻击者攻击开发、编译、构建、测试时使用的开发集成环境,导致开发完成的组件和应用软件具有缺陷或后门,开发人员的隐私数据被泄露等 3. 攻击者向源码管理器提交通过具有缺陷或后门的源代码	1. 组件的完整性 2. 组件及应用的完整性开发集成环境的完整性,组件和应用软件的完整性,数据机密性 3. 组件及应用的完整性
应用软件分发环节	攻击者通过分发市场的缺陷制作并上传伪装、虚假的应用软件或篡改已有的应用软件欺骗终端用户下载使用	应用软件的完整性
应用软件使用环节	1. 攻击者对软件下载更新过程进行劫持 2. 攻击者利用软件的缺陷、后门进行攻击,如窃取软件和运行环境中的敏感数据,对运行环境中的其他数据进行篡改,拒绝服务攻击以及提权攻击等	1. 下载更新过程的完整性,软件的完整性 2. 应用软件的可用性,应用软件数据的机密性,运行环境的完整性

2.2 开源软件供应链的攻击事件分析

下面本文通过对开源软件供应链的攻击事件的分析, 进一步介绍表 1 的威胁模型, 并总结当前开源软件供应链的风险趋势.

2.2.1 组件和应用软件开发环节的安全风险

组件和应用软件开发环节中, 多个开发人员共同协作把代码上传到源码管理器. 接着, 项目代码被编译、构建和测试, 然后组件会被发布到第三方组件分发市场. 应用软件及组件开发集成过程中会从第三方组件分发市场安装组件依赖, 提高开发效率. 考虑到组件开发和应用软件开发共同面临着恶意代码提交、恶意的第三方组件依赖等风险问题, 本文对两个环节进行了总结, 统一分析两个开发环节面临的安全风险.

首先, 针对开发环境, 攻击者可以通过污染开发人员使用的开发、编译、构建、测试时使用的开发集成环境, 导致开发完成的组件和应用软件具有缺陷或后门, 或窃取开发人员的隐私数据. 该攻击面主要利用了污染、操控、攻陷开发集成工具和环境等攻击向量. 在组件和软件开发过程中如果被攻击者在源代码级别植入恶意代码, 由于源码审查的难度较高, 导致这一问题将很难被发现. 且这些恶意代码在披上正规软件厂商的合法外衣后更能轻易躲过安全软件产品的检测, 可以长时间潜伏于用户机器中不被察觉. 典型开发集成环境污染事件如 2019 年 Unix 管理工具 Webmin 的构建服务器中被攻击者发现存在可以隐秘方式修改密码、提升权限的漏洞, 该漏洞允许已知此后门知识的攻击者在缺少输入验证的情况下而执行恶意代码[21]. 类似的构建平台攻陷事件 SolarWinds[22]中, 攻击者攻陷构建平台并添加在每次编译中执行恶意行为逻辑的代码. 2017 年远程终端管理工具 Xshell 由于开发人员的电脑被攻陷, 导致开发人员开发的代码存在后门, 进一步影响了最终交付到终端用户手里的应用软件, 即 Xshell 软件也带有恶意后门, 这一版本的应用软件在国内被大量分发使用[23]. 2015 年 9 月 14 日曝出的针对 Xcode 非官方恶意版本污染事件(XcodeGhost)[24], 攻击者通过修改 Xcode 配置文件, 导致编译链接时程序强制加载恶意库文件, 最终导致经过污染过的 Xcode 版本编译出的 App 程序, 都会被植入恶意逻辑,包括向攻击者回传敏感信息, 并可能导致被远程控制的风险.

此外, 由于开源协作的新模式, 源码管理可能存在新的缺陷, 如攻击者可以向源码管理工具提交具有缺陷或后门的源代码, 在软件开发中注入漏洞, 或窃取隐私. 其中隐私信息包括开发管理过程中的敏感资源与流程相关的信息, 例如设置的密钥、生产日志、漏洞追踪、参数配置和临时文件等. 该攻击面主要采用的攻击向量是绕过源码管理工具的验证管控、窃取开发人员的信任凭证等. 针对漏洞注入, 利用源码管理器验证管控的缺陷, 攻击者曾攻击源码管理器 Git, 通过伪造 Git 创建者的签名向源代码中注入可执行任意代码的后门[25]. 此外, Event-stream 事件中攻击者获取了 Event-stream 仓库的发布权限并发布含有窃取加密货币功能的恶意逻辑[26,27]. 攻击者获得源码管理器的控制权限并发布恶意软件的例子也发生在 Arch Linux 社区[28]. Gonzalez 等人在研究中[29]提到了针对 Github 的恶意提交攻击,攻击者向 Github 的开源项目中进行恶意的提交, 来向项目中注入恶意的代码. 攻击者曾通过攻击 SVN 和 Git 的源代码仓库 CodeSpaces[30], 最终控制后端、轻易的删除仓库中的源代码. 针对隐私泄露, 攻击者利用持续集成工具 Bash Uploader 的缺陷来导出存储在用户开发环境中的信息, 包括一些密钥、凭据[31]. 这类攻击事件破坏了开源软件供应链中组件和应用软件的完整性和开发人员数据机密性.

最后, 针对开发依赖, 开源软件供应链面临着恶意第三方组件依赖和代码复用的安全问题. 攻击者通过第三方组件分发市场的缺陷, 制作伪装、虚假的第三方组件,或篡改已有的第三方组件, 欺骗开发人员下载使用. 具体来说, 攻击者可以通过如窃取分发市场的信任凭证, 借助分发市场管理上的错误绕过验证等攻击向量来上传伪装、虚假的第三方组件. 典型攻击事件通过引入恶意第三方库带来的安全风险, 攻击者通过向 PyPI 包管理器上传常用包名称相似的恶意包[32], 如与常用的数值分析包 Numpy 仅有一个字符之差的 Mumpy, 诱骗开发人员误下载并安装. 除了面向开发语言的包管理器之外, 面向操作系统、容器镜像中心、物联网设备语音技能等的第三方组件分发市场也存在着伪装、虚假第三方组件的风险, 如容器镜像中心 DockerHub 托管的系统与软件镜像曾被曝出 17 个后门镜像[33], 其中包括最流行的部分应用程序容器如 MySQL 和 Tomcat 镜像. 攻击者可以通过第三方组件分发市场中不存在的依赖组件或注册更高版本的同名依赖组件, 从而能够

欺骗开发人员下载使用，如苹果、微软等公司等多家知名公司曾受到这一攻击[34]. 上述事件主要是由于第三方组件的来源不可信，第三方组件分发市场的验证审核机制不够完善，导致开源软件供应链中组件的完整性被破坏. 此外，第三方组件依赖及代码复用可能导致上游的漏洞被引入到下游的组件及应用软件中，例如从Stackoverflow 抄写代码，引入不安全的代码内容. 针对第三方组件依赖，2014 年首次披露的 Heart Bleed 攻击[35]是加密程序组件 OpenSSL 暴露的一个漏洞，当年在最热门的启用 TLS 的网站中有 80 万个网站含该组件依赖，且有 1.5%易受心脏出血漏洞的攻击. 针对代码复用，2018 年在 WinRaR 中[36] 发现的漏洞可导致受害者计算机完全被攻击者控制，该漏洞存在于 unacev2.dll 模块中，相关代码被其他 220 余款软件复用，包括 BandZip、好压等，上述软件均受到该漏洞的影响. 2017 年，Fischer 等人[37]对安卓应用程序中的代码复用展开研究，进一步揭示了代码复用问题的普遍性及严重性. Fischer 等人发现了 130 万个应用程序中有 15.4%的程序具有重用的代码，且重用的代码中有 97.9%的代码片段存在安全问题. 最后，代码复用问题还携带着经常被忽视的法律风险，若是未按照开源许可证约定使用开源组件会引发潜在的法律纠纷. 最常见的许可证违反发生在 GPL（GNU 通用公共许可证）的使用中，一旦商用产品的组件依赖沾染到该许可证就必须开放该产品的源代码. 在过往的法律纠纷中，侵权行为的表现通常是商用产品依赖开源组件却不遵循开源协议私自以闭源形式出售、以及开发人员违规进行代码复用. FSF 曾状告思科公司以 Linksys 品牌出售的各种产品违反了 FSF 拥有版权的程序许可条款，包括 GCC、GNU Binutils 和 GNU C 库 [38]. 另一案例发生在近期，起因是在谷歌供职的 Joshua Bloch 直接从 Oracle OpenJDK 复制了 9 行代码到谷歌的 Android 项目中，而 Android 项目没有按 GPL 兼容的方式授权，此行为侵犯了 Oracle 的著作权，为此 Oracle 向谷歌索赔 90 亿美元[39].

2.2.2 应用软件分发环节安全风险

应用软件分发环节主要面临分发管理的风险. 针对分发管理的风险，攻击者通过分发市场的缺陷制作并上传伪装、虚假的应用软件或篡改已有的应用软件欺骗终端用户下载使用. 该攻击面的主要攻击向量包括绕过分发市场的验证管控、包名误用攻击、恶意接管分发市场的管理权限等. 一个典型的例子是 WireX Android BotNet 污染 Google Play 应用市场事件[40]. 该事件中，恶意软件通过伪装成普通的安卓应用在 Google Play 安卓应用市场分发，通过感染大量的安卓设备发动了较大规模的 DDos 攻击. 这类事件导致应用软件来源不可信，破坏了应用软件的完整性.

2.2.3 应用软件使用环节安全风险

应用软件使用环节主要面临两大安全风险，包括下载更新机制和运行环境的风险.

首先,针对下载更新机制，用户下载更新软件过程中存在下载更新通道被劫持的风险. 这主要是当前网络通道和下载更新过程无法保证完全安全，仍存在 DNS 劫持、中间人攻击、钓鱼攻击等攻击向量，导致用户错误地连接到攻击者的恶意分发站点上，欺骗终端用户下载更新. 典型的攻击事件如 2017 年 6 月 27 日晚被发现的 NotPetya 勒索病毒[41]，其主要通过劫持软件更新的方式传播，攻击者劫持了乌克兰专用会计软件 Me-Doc 的升级程序，用户在通过升级程序更新时，就会感染病毒. 这类攻击让用户对可信的更新升级网站产生了巨大的怀疑和不信任，对更新升级的安全性和完整性产生了担忧. 同时，此类攻击事件一旦成功，还将带来巨大的经济财产损失. 另一个典型攻击事件如 2017 年 7 月报告的 bjftzt.cdn.powercdn.com 域名劫持攻击[42]，用户尝试升级若干知名软件客户端时，运营商将 HTTP 请求重定向至恶意软件并执行. 恶意软件会在表面上正常安装知名软件客户端的同时在后台偷偷下载安装推广其他软件. 全国多省的软件升级劫持达到空前规模，360 安全卫士对此类攻击的单日拦截量突破 40 万次. 攻击者也可以直接攻陷更新网站[43]，利用新的恶意软件去覆盖软件更新程序而不是伪装成更新程序，它会覆盖其他应用程序的更新功能，这可能会给用户带来长期风险. 上述事件主要是由于应用软件的更新过程的完整性被破坏，进而导致应用软件的完整性遭到了严重破坏.

其次，针对运行环境，应用软件可能存在来自软件供应链上游的漏洞缺陷. 攻击者可以利用软件的缺陷、后门进行攻击，如利用窃取软件和运行环境中的敏感数据，对运行环境中的其他数据进行篡改，拒绝服务攻击以及提权攻击等攻击向量. Xiao 等人[44]通过研究 Node.js 程序中对象共享和通信过程，设计一种向 Node.js

服务器程序发送具有隐藏属性的数据，从而窃取机密数据、绕过安全检查和发起拒绝服务攻击. 攻击者也可利用依赖中的源码的漏洞，例如借助脏牛漏洞[45](CVE-2016-5195)修改 su 或者 passwd 程序获得不正当的 root 权限. 这类事件破坏了应用软件的可用性，应用软件数据的保密性，运行环境完整性.

2.2.4 开源软件供应链风险的总体趋势分析

本文汇总并分析了开源软件供应链从 2010 年以来的攻击事件，绘制了如图 3 所示的时间图. 从图 3 可以看出，近十年来开源软件供应链攻击事件基本呈现不断上升的趋势，尤其是在 2019 年，平均每 5.6 天就有一起针对开源软件供应链的攻击事件报导. 一方面，这是由于近年来软件系统的蓬勃发展，新兴的开源协作软件开发模式使得软件供应链的供应关系愈发庞大复杂，潜在的威胁渗入点大大增加. 另一方面，复杂的网络型供应链的可视性和可控性更加受限，导致供应链的管理更加困难. 此外，近两年以来，开源软件供应链攻击有所放缓，这得益于软件供应链各个环节的开发人员、管理人员、厂商的不断重视，开展了关于开源软件供应链安全的研究和实践，如开源软件包管理平台对提交上传的包进行恶意检测和识别.

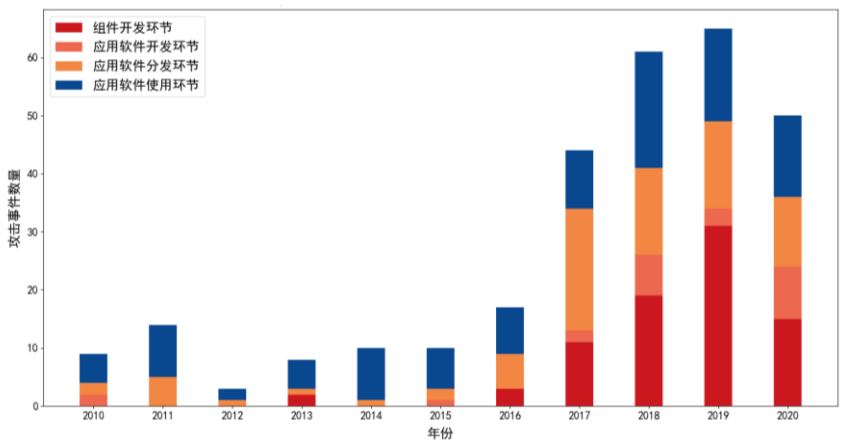

图 3 开源软件供应链近十年的攻击事件趋势分析

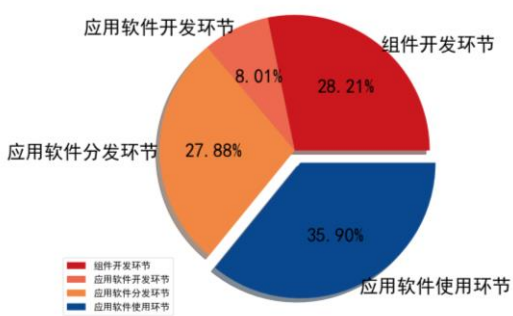

图 4 各个环节出现的攻击事件占比

为了详细探究各类供应链攻击的形成方式，本文绘制了近十年来开源软件供应链四个环节攻击事件的成分图，如图4所示，应用软件使用环节的安全事件占比最高. 这是由于攻击者主要目标是实现对终端用户的攻击，并获取收益，因此大多数攻击都会在该阶段暴露. 而在开发和分发环节，攻击者的目的是为了注入恶意代码或者植入后门，隐蔽性更高，检测发现安全事件的难度更大，所以占比相对较少. 此外，图 3 中显示，2017 年到 2020 年期间开发环节和分发环节的攻击事件占比逐渐增加，甚至超过使用环节. 可见，在新型复杂网络开发背景下，研究人员开始重视对开源软件供应链的上游开发和分发环节的研究和分析。相应地，针对开发和分发环节的威胁检测能力也在不断提高.

基于上述分析, 可见针对新型开源协作模式下的软件供应链的安全研究仍需进一步加强, 积极主动应对新型开源协作模式下复杂开源供应链带来的风险, 尤其是加强对开源软件供应链上游环节的风险检测和加强对开源软件供应链的加固和防御.

3 开源软件供应链的风险识别与评估

基于上述安全风险, 安全人员针对开源软件供应链各个环节的攻击面设计了风险识别技术.

(1) 针对组件和软件开发环节中, 现有工作主要研究了供应链视角下组件及软件的漏洞检测、面向第三方组件分发市场的风险识别, 其中第三方组件分发市场的风险识别包括第三方组件分发市场中的恶意软件识别, 以及第三方组件分发市场的风险检测; 此外, 在组件和软件开发环节, 考虑到协作开源的场景, 研究人员分别从开源项目的安全性以及开源代码贡献人员的隐私性两个方面研究协作开发的风险.

(2) 针对软件分发环节, 当前的研究集中在对软件分发市场如安卓软件商店等中软件的风险识别;

(3) 针对软件使用环节, 当前的研究重点分别是软件下载更新时网络劫持的风险识别及软件风险识别.

综上, 本文将各个环节的风险识别和评估总结为以下四个关键研究方向: 开源软件供应链中第三方组件的风险识别、开源软件供应链视角下的应用软件风险识别、协作开发的风险识别及下载更新过程的风险识别.

其中, 第三方组件的风险识别包括了分发市场中的第三方组件风险识别及其风险识别和第三方组件分发市场的安全评估; 开源软件风险识别可以被应用在开发环节中开发人员的测试, 或应用在分发环节中分发市场或研究人员对软件的审核, 在使用环节, 用户也可对软件进行风险识别后再进行使用. 因此, 本文总结了开源软件供应链视角下各个环节的应用软件风险识别技术, 并放在 3.1 节讨论, 包含应用软件中的第三方组件检测及其风险识别、应用软件代码复用检测以及分发市场中的恶意应用软件识别.

接下来, 本文对上述四个关键研究方向的风险识别技术进行分析和总结.

3.1 开源软件供应链中第三方组件的风险识别

近些年来, 研究人员开始关注开源软件供应链中第三方组件的安全性. 在软件供应链中, 开发人员从第三方组件分发市场下载安装第三方组件依赖. 这一过程中, 研究人员着力开展了对分发市场第三方组件的风险识别, 同时, 也对第三方组件分发市场进行分析和安全评估.

3.1.1 分发市场中的第三方组件风险识别

传统的第三方组件研究主要集中在如 PyPI、NPM、RubyGem、OPKG 等主流第三方组件分发市场. 上述例子中针对特定开发语言的第三方组件分发市场也被称作包管理器. 此外, 面向新兴的物联网语音设备, 也涌现出了第三方语音技能分发市场、自动化应用程序的组件分发市场等, 如亚马逊的语音技能市场、IFTTT 平台. 已有部分工作研究了上述第三方组件分发市场及软件系统的第三方组件风险识别问题.

(1) 面向包管理器的第三方组件风险识别

考虑到新型复杂网络开发流程下, 包管理器中的包一旦出现安全问题, 如存在缺陷导致攻击者可以注入并执行恶意代码等, 将影响到下游海量应用程序的安全. 因此, 已有多个研究指出[46]安全研究人员和分发市场本身需要提出相应检测方案, 支持大批量规模化地对包管理器及其他分发市场上的第三方组件进行检测分析,减少包管理器中潜在的有风险的第三方组件.

面向包管理器的风险识别, 研究人员主要研究了恶意包注入的检测技术和其他支持在包管理器中中批量扫描和检测潜在具有漏洞风险的第三方包的相关研究. 其中, 包名抢注是是近年来研究人员发现的一种恶意包注入的攻击手段, 攻击者以流行包相似名字注册的恶意包, 使开发者误以为恶意包是他要下载的目标包, 大大提升恶意包的下载概率, 并进一步传播到开发的软件中. 如先前研究识别发现 PyPI 市场上 12 个包名误用的恶意包. 如图 5 所示, 此次检测出四个恶意包 diango、djago、dajngo、djanga 是 Django 的拼写错误, 其中 Django 是一个十分流行的 Python 框架. 这些恶意包能够替换用户的比特币地址, 劫持用户资金. 包名误用问题十分严峻, 亟需针对包名抢注的高效检测方案.

图 5 四个针对”Django”包的恶意抢注包

当前的检测方案主要通过包名抢注的两大特性来检测: (1) 攻击目标往往是流行包; (2) 恶意包和原始包的名字十分相似. Taylor 等人[47]利用(1)和(2)提出了一种包名抢注检测技术. TypoGuard 专门针对该攻击建立名称相似性模型, 并结合第三方包流行程度特征实现恶意包的检测, 当某个软件包与某个流行的软件包名称十分相似, 且自身的流行度较低, 则认为该软件包为潜在的恶意软件包. 考虑到(2), Python 的第三方包管理器 PyPI 则提供 PyPi-parker 模块[48], 通过提供一个保存着和正常包名字相近的相关包黑名单, 来帮助开发者在下载选择包时实时地识别恶意包. 当用户试图下载黑名单里记录的包的名字时候, 该工具会提示用户这不是一个正常的包. 不同于上述方案, Vu 等人[49]提出了一种针对 PyPI 包管理器的恶意包检测方法, 需要首先确定正确的包名, 再利用可重复构建的思路, 比较包管理器中的包和源代码存储仓库 Github 上的包的不同来检测包管理器分发的恶意包. 具体来说,该方法通过挖掘包源代码存储库, 最后对存储库和包管理器的包文件的哈希进行比较来识别恶意包, 提高了检测的准确性.

针对包管理器中的存在漏洞风险的第三方包检测, 研究人员主要致力于发现包管理器中存在的新的风险以及提出有效和系统性的方案来对包管理器中的包进行系统性的检测. 针对一些新发现的包风险检测, 研究人员主要采用基于源代码的分析方法, 分为静态检测和动态检测两种方案. 如 James 等人[50]发现了一种 Node.js 生态系统中的新的第三方组件风险, 能实现针对事件处理程序拒绝服务攻击. 针对这一风险, James 等人提出了一种静态检测方案, 利用正则匹配对 NPM 包的元数据进行分析, 来对 NPM 上的包进行批量检测. 一个动态分析的例子是 Staicu 等人[51]设计的 Synode 框架, 用于识别 Node.js 程序中由于特殊指令引入的注入攻击. 作者开发的 Synode 工具先通过静态分析识别相关 API 的调用点, 基于分析结果实现脚本的模拟运行, 来检查是否存在恶意的第三方组件以开发人员可能无法预见的方式实现恶意代码执行. 结合静态分析和动态分析,该方案具有较高的检测效率和准确率. 上述研究方案主要是在发现新风险的基础上设计一个自动化可扩展的工具对分发市场的组件进行批量检测, 静态分析的准确率较低, 而动态分析提高了准确率的同时, 性能开销较大.

相比针对包管理器特定风险的检测研究, Duan 等人提出了一个更有效和普适的方案, 即 MALOSS[46]框架, 该框架通过元数据分析、静态分析和动态分析来对 NPM、PyPI 和 RubyGems 三大市场中的第三方包进行全面、系统的分析, 具体介绍如下:

- 元数据分析: 通过提取代码包的元数据,如包的名称、开发作者、维护者、发布版本、下载次数和依赖关系等来识别出潜在的恶意数据包;
- 静态分析: 通过对代码包的源代码进行分析, 标记网络、文件系统、进程、代码生成相关的 API, 进一步将源代码解析为语法结构树并搜寻标记 API 的使用情况, 最后分析代码数据流, 检查代码数据流的源、sink 节点和传播节点;
- 动态分析: 通过在隔离的 docker 环境中安装执行二进制代码包, 触发导入包时的初始化逻辑来测试 import 包, 用动态模糊测试触发相应功能来测试导出函数, 最后用 sysdig 来捕获代码运行时的系统调用 trace.

除了包管理器这一第三方组件分发市场, 恶意包注入和第三方包风险检测的研究也同样在新兴的物联网分发市场, 即第三方语音技能分发市场以及自动化应用程序组件分发市场, 有一定进展. 其中语音技能是用于物联网场景下智能语音设备上应用的开发, 如亚马逊 Alexa[52]和 Google Assitant[53]提供的语音设备, 他们通过语音通道来和用户互动. 开发人员需要下载特定的语音技能如”打开门”, 并和相关的物联网设备的应用进行集成, 实现用语音就能控制设备的功能, 也就是用户发出语音命令, 智能设备会识别语音指令, 匹配到相

应的语音技能并调用技能实现相应的服务和功能逻辑，如图 6 所示. 自动化应用程序组件则是提供一些自动化流程的建模，如”当室内温度超过 28℃，则把窗户打开”. 开发人员会将这些组件集成到物联网设备中实现智能家居中的自动控制. 由于语音技能和自动化应用程序将会在用户家中部署应用，控制一些关键设备，如门锁、睡眠监控设备等，直接影响用户的生命和财产安全，其风险检测是当前研究重点之一.

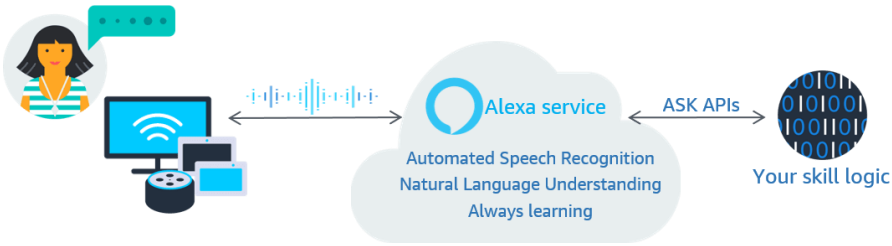

图 6 语音技能的工作流程[52]

(2) 面向第三方语音技能分发市场的第三方组件风险识别

由于物联网的迅速发展，智能语音设备在社会家庭生活中越来越流行. 为了更方便地集成和开发相关语音应用程序，各大物联网厂商如亚马逊、谷歌和小米等提供了语音技能开放平台，允许开发人员上传自己开发的语音技能或者配置部署来自其他用户的语音技能，形成了第三方语音技能分发市场. 针对新兴的第三方语音技能分发市场，恶意包的注入主要体现为语音抢注攻击、语音伪装攻击两种新型攻击，可能导致恶意指令执行、隐私窃取等风险. 其中语音抢注攻击指的是攻击者设计类似发音或者语义的语音名称来劫持用于合法技能的语音命令，如图 7 所示，用户想调用”Capital One”技能可能发出语音指令”Alexa，open Capital One please”，然而如果用户错误地从语音技能分发市场下载并安装了攻击者上传的”Capital One Please”技能，则智能设备听到用户指令后会触发攻击者实现的恶意语音技能. 除此之外，攻击者可能会上传其他相似名称的技能，如图7所示. 该攻击是由于用户利用语音来控制语音智能设备，由于语音命令的含糊不清或用户对语音智能设备的误解，导致智能设备可能匹配到一些与用户预期不同的语音技能. 语音伪装指的是恶意的第三方语音组件在用户和服务商对话期间冒充合法的语音技能来窃取个人信息.

图 7 四个针对”Capital One”语音技能的恶意抢注包

Zhang 等人[54]首次对第三方语音技能组件进行深入研究，并识别发现语音抢注攻击、语音伪装攻击等第三方语音组件特有的新型攻击，提出了检测工具 Skill-name Scanner. Skill-name Scanner 通过对语音技能进行扫描，使用技能响应检查器来捕获来自恶意技能的可疑响应，最后使用用户意图分类却来检查用户发出的命令，来确定是否存在风险. Guo等人[55]则更关注语音技能是否存在窃取用户隐私的风险. 为了分析第三方语音技能的隐私风险，Guo 等人提出了 SkillExplore 框架，对第三方语音技能市场中的语音技能组件进行大规模分析，识别出一些恶意的语音组件技能，如窃取隐私信息、窃听用户对话等. SkillExplore 采用动态分析技术，执行语音技能，并根据语音技能设计回复，实现动态的交互，然后观察目标语音技能的行为，检查用户的隐私是否被获取. 针对语音技能的缺陷检测，Zhang 等人[56]设计了一个引言模型引导的模糊测试工具 LipFuzzer，能够实现黑盒场景下对声音指令进行变异，实现模糊测试，准确率达到 59.52%.

(3) 面向自动化应用程序组件分发市场的第三方组件风险识别

针对流行的自动化应用程序组件分发市场，研究人员研究如何大规模且高效地识别市场中存在潜在漏洞

风险的第三方包, 主要检测是否存在恶意程序组件收集用户的隐私以及程序组件是否存在自身的缺陷易被攻击者利用, 并进一步评估当前分发市场中第三方组件的安全情况. 自动化应用程序分发市场上发布了一系列提供特定应用功能的组件, 由下游开发者或用户对不同功能的组件进行组合配置, 实现自动化的应用程序, 如通过配置温度传感器数值读取组件和控制窗户开关命令的组件, 可以实现一个应用程序, 自动化执行命令: “当温度上升到 35℃时打开窗户”. 针对这一类型流行的自动化应用程序组件的分发市场, 分发市场和研究人员需要检测市场上是否存在恶意组件, 还需要考虑到下游用户安装组件组合的风险, 设计研究方案对分发市场上不同组件的组合进行扫描并分析安全性, 可以帮助用户在安装组件依赖时提供安全风险的评估参考.

针对分发市场上的恶意程序组件, Bastys 等人[57]首次研究了 IFTTT 平台上存在的恶意程序组件, 发现存在用户隐私窃取风险, 如窃取用户私人照片、泄露用户位置和窃听用户和语音控制助手的交流等. Bastys 等人主要使用静态分析的方法, 扫描自动化应用程序组件来识别具有特定安全分类的触发-动作组合命令, 然后搜索出具有这些组合命令的应用程序组件, 判断为潜在的恶意程序.

考虑到下游用户对自动化应用程序组件组合配置的安全风险, 研究人员主要考虑一些程序组件的交互逻辑. 除了软件程序自身的交互逻辑外, Ding 等人[58]发现物联网设备的组件可以共同对环境作用, 当用户安装配置了一些特定的第三方组件的组合时, 形成了一些潜在的物理通道, 攻击者可以利用这些物理通道控制受害者的物联网设备并执行恶意指令. 上述恶意攻击如图 8 所示, 当受害者不在家, 通过加热暖气, 使得温度升高后打开窗户. Ding 等人设计并实现了 IoTMon, 能够发现潜在的物联网设备组件组合潜在的物理交互链,并进行安全评估. 具体而言, IoTMon 对自动化应用程序进行代码分析, 结合自然语言处理对程序的描述进行抽象, 通过深度学习来计算自动化应用程序逻辑的相关性, 来挖掘不同第三方组件组合形成的潜在的物理渠道及其交互, 最终利用行为建模和聚类的方法识别是否存在风险. Wang 等人和 Alhanahnah 等人分别开发了 iRuler 和 IoTCom[59,60]来识别应用层面不同组件组合交互存在的缺陷. 其中 iRuler 对自动化应用程序进行解析, 并结合设备服务的元数据以及配置信息来对规则进行统一的建模, 然后使用模型检查对自动化应用程序的执行逻辑进行检查, 能够识别五种组件组合的潜在风险. IoTCom 首先利用路径敏感的静态分析自动提取自动化应用程序的控制流图, 接着使用图抽象技术来对设备和用户使用的 app 进行连接 ,构建行为规则图, 然后提取为形式化的模型, 最终使用模型检验的方法来检查一些潜在违反安全属性的路径. IoTCom 和 iRuler 的总体思路比较相像, 但进行了一系列改进, 例如更加自动化、支持物理通道的检测、关注更多的控制条件等.

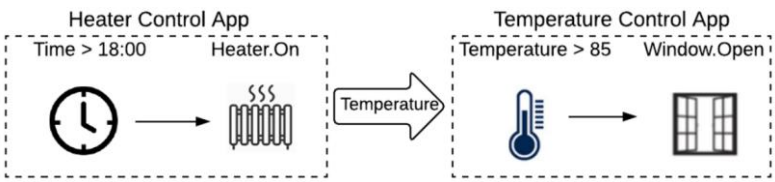

图 8 一个存在风险的利用物理通道交互的例子[58]

综合上述分析, 安全研究人员及分发市场对不同类型的组件分发市场的风险都设计了相应的风险识别方案. 其中针对包名抢注、语音抢注类型的攻击, 研究人员主要利用该攻击的特性, 即包名相似这一点特点, 结合其他特征设计检测方案, 包括设置黑名单、计算流行度等; 针对第三方组件, 研究人员也在探索现有的第三方组件存在新的安全风险, 并设计相应的风险检测方案支持大规模识别分发市场中第三方组件的风险. 针对上述研究, 大部分学者采用了元数据、静态分析等技术手段, 检测效率高, 但也存在高误报率、低检测率的问题; 也有一些研究采用了动态技术如模糊测试等来提高检测的效率, 但检测开销时间较大. 考虑到第三方组件分发市场的多样性, 仍缺少自动化可扩展的工具, 也缺少更加智能高效的检测方法.

此外, 在物联网的快速发展背景下, 新的第三方组件如语音技能、自动化应用程序组件呈现出新的威胁渗入点, 如语音伪装攻击等, 此外开发人员在开发软件系统、语音智能设备和自动应用程序等需依赖大量组件, 组件组合也存在如依赖冲突等风险. 从供应链上游的组件分发市场上大规模分析组件组合配置的风险, 为开发人员提供安全风险提示和评估, 是有效渠道之一, 而针对该方向等相关研究仍较少, 仅有少数研究探

索了自动应用程序的组合风险, 针对其他的风险, 如第三方包组合配置风险识别技术需要更深入的探索.

表 2 经典的分发市场中恶意第三方组件识别方法总结

方法名称	检测的开发语言	分发市场	分析方法		分析场景		性能		可扩展	支持识别的缺陷
			静态	动态	黑盒	白盒	检测能力	检测速率		
TypoGuard[47]	JavaScipt ,Python, Ruby	NPM,PyPI, RubyGems	√	×	×	√	○	●	◑	包名误用
PyPi-parker[48]	Python	PyPI	√	×	×	√	○	●	○	包名误用
Vu 等人[49]	Python	PyPI	√	×	×	√	◑	◑	◑	包名误用、代码缺陷
James 等人[50]	JavaScript	NPM	√	×	×	√	○	●	○	代码缺陷
Synode[51]	JavaScript	NPM	√	√	×	√	◑	●	○	代码缺陷
Staicu 等人[61]	JavaScript	NPM	√	√	×	√	◑	◑	○	代码缺陷
MALOSS[46]	JavaScipt ,Python, Ruby	NPM,PyPI, RubyGems	√	√	×	√	●	◑	◑	隐私泄露、代码缺陷、包名误用、网络劫持等
Skill-name Scanner[54]	\	Alexa skill market	√	√	×	√	●	●	◑	语音抢注、语音伪装
SkillExplorer[55]	\	Alexa skill market	√	√	√	×	◑	◑	●	隐私泄露
LipFuzzer[56]	\	Alexa skill market, Google Assistant vApp	×	√	√	×	◑	●	●	语音抢注、语音伪装
Bastys 等人[57]	Groovy	IFTTT	√	×	×	√	○	●	◑	隐私泄露
IoTMon[58]	Groovy	SmartThings	√	×	×	√	◑	◑	◑	组件组合风险
iRuler[59]	Groovy	IFTTT	√	×	×	√	◑	◑	◑	组件组合风险
IoTCom[60]	Groovy	SmartThings, IFTTT	√	×	×	√	◑	●	◑	代码缺陷

注:●=高, ◑=中, ○=低; √=满足,×=不满足; 检测能力是对检测准确率和误报率的综合评价; 可扩展是方法针对不同开发语言及分发市场的扩展能力的综合评价, 可传递是指方法能识别依赖的组件, 并进一步识别依赖的组件和别的组件依赖关系.

3.1.2 第三方组件分发市场分析及安全评估

面向第三方组件分发市场的分析和安全评估, Kula 等人[62]在 2017 年对 NPM 包管理器中的 JavaScript 组件进行大规模的评估, 主要分析第三方包的分布、依赖关系和开发人员的使用成本. 该项研究证明了第三方包在供应链关系中的依赖链条较长, 第三方包的变化会对整个软件生态系统造成巨大的变化. Dey 等人[63]则对 NPM 包上下游依赖关系做了分析, 从供应链视角分析第三方组件的依赖网络, 并评估第三方组件下载数量的变化. 该研究揭示了供应链中上下游依赖关系对第三方组件的分发有重要影响, 供应链角度的研究能帮助人们更好理解软件开发和软件生态系统. 除了第三方组件的供应关系外, 也有研究对包管理器中软件包的滞后性进行了大规模评估. Zerouali 等人[64]从开发和运行时两个阶段去分析 NPM 包中第三方组件的依赖关系和技术滞后性. 这个研究旨在检测落后的数据包, 避免开发人员部署已经过时或者不期望的软件包. Zimmermannn 等人[65]的研究进一步系统化地分析了当前 NPM 第三方软件包生态系统中的依赖关系、负责包的维护者以及公开报告的安全问题. 该研究指出单个第三方包甚至能够影响整个生态系统的大部分包, 且只需要拥有极少部分维护者账号就能够将恶意代码注入生态系统中大多数的包, 该问题随时间推移越发严重.

认识到第三方组件分发市场中组件漏洞的危害性后, 已有研究人员对漏洞影响力和修复情况进行了研

究. Decan 等人[66]对超过 61 万个 JavaScript 包中的 NPM 依赖网络进行漏洞影响的评估，分析这些漏洞是何时以及如何被发现和修复的，并进一步分析它们在存在依赖约束的情况下对 JavaScript 包生态中的其他包的影响范围. 这个研究为包维护人员和开发人员提供了安全指导，帮助改进处理安全问题的过程. Ruohonen 等人[67]则研究并分析了 Python 第三方包分发市场，从时间序列、软件历史漏洞等多个维度分析 Web 开发中常见的 Python 包存在的漏洞. 研究指出包名抢注已经成为严重的攻击方式之一. 最为全面和完整的研究是 2021 年 Duan 等人[46]的文章. 他们对 NPM、PyPI 和 RubyGems 三种主流语言的包管理平台进行了大规模系统性的安全分析，对参与包管理平台的主要角色进行建模，从功能性、代码审查和响应措施三个角度对包管理平台进行定性分析. 该研究提供了第一个系统化的分析框架用于评估包管理平台的安全性，结果如表 3 所示.

表 3 对第三方包管理平台的评估方案[46]

特征				包管理器		
				PyPI	Npm	RubyGems
功能性检查	软件包维护者	发布包的权限	密码验证功能	●	●	●
			访问令牌验证功能	◑	●	●
			公钥验证功能	○	○	○
			多因子验证功能	◑	◑	◑
		发布	上传功能	●	●	●
			引用功能	○	○	○
			签名功能	◑	◑	◑
			拼写保护功能	○	●	●
			命名规则推荐的功能	○	◑	○
		管理	删除数据包	◑	◑	◑
			弃用数据包	○	◑	◑
			添加协作者	◑	◑	◑
			转移拥有权	◑	◑	◑
	开发者	选择包作为依赖项	提供信誉评分	●	●	●
			提供代码质量评估	○	○	○
			提供安全实践分析	○	○	○
			提供已知风险分析	○	○	○
			提供包名误用检测功能	○	○	●
		安装包	Hook 安装	●	◑	○
			安装时指定特定依赖项	○	◑	◑
			源代码安装	◑	◑	◑
			嵌入式的二进制安装	◑	◑	◑
审核检查	软件包维护者、开发人员	元数据检查	依赖检查	○	○	○
			更新检查	○	○	○
			二进制检查	○	○	○
			软件包维护者账号	○	○	○
		静态分析	语法错误识别	○	○	○
			逻辑错误识别	○	○	○
			恶意逻辑代码	○	○	○

		动态分析	安装过程检测	○	○	○
			嵌入式二进制检测	○	○	○
			Import 检测	○	○	○
			功能性的检测	○	○	○
补救响应功能	软件包维护者、开发人员、终端用户	删除	删除包	●	●	●
			删除发布者	●	●	●
			删除已安装的包	○	○	○
		通知	通知软件包维护者	○	○	○
			通知相关软件包维护者	○	○	○
			通知开发人员	○	○	○
			通知漏洞建议修复数据库	○	●	●

注:●=强制的, ◑=可选的, ○=不支持

根据 Duan 等人[46]的研究, 可以看出, 现有的第三方包供应链生态在功能性检查、审核检查和补救响应审查上仍缺失大量措施, 审核检查阶段是当前最薄弱的阶段, 在补救响应上仍存在较大缺失. 当前第三方包供应链生态需要开发人员、软件包维护者、管理平台等共同努力, 维护安全可信的供应链.

此外, 面向新兴的物联网语音设备, Cheng 等人[68]对第三方语音技能市场的审核机制进行了分析和评估. 他们的研究证实可以轻松在语音技能分发市场上发布一些违反安全策略的语音技能, 揭示了当前语音技能分发市场存在较大的风险. Alhadlaq 等人[69]对亚马逊语音技能市场的隐私策略进行了安全分析, 发现该市场上75%的第三方语音技能都没有隐私策略. 相比上述研究, Lentzsch 等人[70]实现了更多维度的语音技能分发市场的安全分析, 不仅揭示了当前市场中语音技能审查的缺陷, 还进一步分析当前语音技能并没有严格遵守关于访问用户敏感数据的政策.

综上所述, 研究人员已经在第三方组件分发市场的分析和安全评估上开展了一系列的研究, 通过对第三方组件市场数据的分析能够增强对软件生态系统的理解. 更重要的是, 研究人员揭示了当前第三方组件分发市场中的组件仍存在很多安全隐患而市场的审查机制不够完善, 且和当前学术研究存在一定差距, 3.1.1 节研究提到的前沿的第三方组件风险识别方案仍未应用到当前主流的开发市场的审查机制中.

3.2 开源软件供应链视角下的应用软件风险识别

传统的应用软件风险识别利用静态分析和动态测试[71–73]来挖掘软件潜在的漏洞和后门. 其中, 污点分析[74–78]是静态分析的主流方案, 模糊测试[79–87]是动态测试的主流方案. 本文聚焦于开源软件供应链视角下特有的软件风险及对应的风险识别方案, 包含应用软件中第三方组件检测及其风险识别, 应用软件代码克隆检测及分发市场中的恶意软件识别. 其中, 针对应用软件中的第三方组件检测, 新型开源协作模式下应用软件中可能存在大量来自上游的第三方组件, 一旦这些第三方组件存在缺陷或者被攻击者注入恶意后门, 相应的应用软件也会存在风险. 因此, 越来越多的研究关注如何识别应用软件中的第三方组件, 并进一步基于识别的组件来发现上游组件存在漏洞导致的带“菌“传播风险. 除了识别 1day 漏洞外, 考虑到应用软件可能会违背第三方组件的许可证要求, 研究人员进一步基于识别的第三方组件来发现第三方组件许可证违规的风险. 针对软件代码克隆检测,安全研究人员分析应用程序之间的代码相似性, 进一步用于检测软件代码复用导致的上游漏洞带“菌“传播问题, 以及许可证违规的风险. 针对分发市场中的恶意软件识别, 考虑开源软件供应链的分发环节中攻击者会对软件进行恶意篡改或者添加恶意载荷, 研究人员探索了如何利用软件相似性技术来识别这些恶意的软件.

3.2.1 应用软件中的第三方组件检测及其风险识别

针对特定的开发软件系统, 其使用的第三方组件以及内在的依赖关系检测能够帮助开发者、维护者和使

用者实现对软件系统的总体架构理解, 进一步实现对软件系统的可靠性风险管理. Tellnes 等人[88]认为软件系统工程采用大量第三方依赖库会带来严重的安全隐患和可用性上的限制. 他们的研究指出当前软件和系统的安全性和可用性在很大程度上取决于依赖的第三方组件的生态系统. 因此, 研究人员从基于元数据的第三方组件静态检测、基于源代码的第三方组件静态检测、基于二进制代码的第三方组件静态检测等多个角度研究高效检测软件系统中的第三方组件及其漏洞.

元数据静态检测主要是利用软件系统中提供的第三方组件相关字段等信息进行检测. Maven 组织实现了 OWASP Dependency Check 工具[89], 支持扫描项目中的依赖项文件名、供应商和版本等信息并识别潜在的依赖第三方组件, 进一步通过确定项目依赖项中是否存在通用平台枚举标识符(CPE)来检测项目依赖关系中是否包含公开的披露漏洞, 其中 CPE 标识符唯一地表示受 CVE 影响的应用程序. 考虑到有些被扫描的字段可能发生改变, 导致该工具存在较大的漏报率. Cadariu 等人[90]则进一步扩展 OWASP 依赖关系检测工具, 添加 JAVA 依赖关系知识来提取 JAVA 中的系统依赖关系, 利用 CPE 的匹配方式来确定是否存在漏洞.

上述工具仅仅通过一些简单的字符匹配技术来进行检测, 由于元数据无法真实反映软件的依赖关系且一但代码采用了一些混淆技术, 上述工具就无法识别相应的特征, 存在大量漏报. 因此, 研究人员提出了基于源代码的静态检测方案, 主要利用哈希、控制流图和函数名等信息的匹配. Backes 等人[91]提出基于哈希匹配的安卓库检测工具 LIBSCOUT, 支持提取应用程序中混淆后的配置特征, 能够抵抗常见的代码混淆问题, 并精确定位应用中使用库的版本. LIBSCOUT 主要采用多层哈希的方法, 对安卓库的 method、class、package 类层层递进实现哈希, 最后进行匹配. Zhang 等人[92]提出的 Atvhunter 可以查明应用程序使用的易受攻击的第三方组件版本并提供有关第三方组件漏洞的信息. 作者提出了一种两阶段检测方法来识别特定的第三方组件版本. 具体来说, Atvhunter 通过提取控制流图(CFG)作为粗粒度特征以及提取 CFG 的每个基本块中的操作码作为细粒度特征来识别确切的第三方组件及其版本. 同时, 为了识别特定的易受攻击的应用内第三方组件版本, 作者还构建了一个全面完整的易受攻击的第三方组件数据库, 其中包含来自行业合作伙伴的 1,180 个 CVE 和 224 个安全漏洞. ATVHunter 是最先进的第三方组件检测工具之一, 实现了 90.55% 的准确率和 88.79% 的召回率, 且对常用混淆技术处理过的代码仍具有一定适用性, 并且可扩展用于大规模第三方组件的检测. 针对修改后的第三方组件, 上述技术无法识别嵌套的第三方组件, 也无法实现高扩展性, 难以应对开源软件项目的数量和规模的庞大. Woo 等人[93]提出 CENTRIS, 旨在克服上述第三方组件的分析限制, 以精确且可扩展的方式有效检测修改后的第三方组件. CENTRIS 采用了一种称为代码分段的技术, 使用松散匹配来检查目标软件和第三方组件之间的代码相似性是否大于预定义的阈值, 来嵌套检测相似的第三方组件依赖关系. 这种代码分段使 CENTRIS 降低了误报率, 同时即使在大量修改或嵌套的情况下仍能识别所需的第三方组件.

上述方法都是已知第三方库的特征对未知的软件系统进行匹配识别, 另一种方案是直接对软件系统进行第三方库的提取, 无需提前获取到第三方库的知识. 针对第二种方案, Li 等人[94]提出了一个抗混淆的第三方组件检测工具 LibD, 用安卓程序内部的代码依赖性来检测和分析潜在的第三方组件, 相比代码相似性检测, LibD 使用特征哈希处理混淆的函数名称, 准确率较高. SAP 团队提出了一种检测、评估和缓解第三方组件漏洞的新方法 Eclipse Steady[95], 以代码为中心, 在给定的上下文中结合静态和动态分析来确定库中易受攻击部分的可达性. 为了评估 Eclipse Steady, 作者将其检测能力与 OWASP Dependency Check (OWASP DC) 的检测能力比较, 评估结果具有显著提升. 更自动化且系统化的是 Duan 等人[96]提出的一种可扩展的全自动工具 OSSPolice, 支持在没有目标应用软件源代码场景下对二进制的程序进行第三方组件的识别, 对软件系统的开源软件证书冲突进行了检测, 并对 1-day 漏洞进行识别. OSSPolice 引入了一种新的分层索引方案来比较应用程序的二进制文件和数万个第三方库源代码的数据库, 能够实现高可扩展性和高准确率的第三方组件检测. OSSPolice 同样支持检测许可证违规, 通过识别应用软件中的许可证类型, 判断其是否和他依赖的第三方组件中的许可证冲突.

相比上述先检测第三方组件再分析第三方组件是否存在漏洞, Ohm 等人提出 Buildwatch[97], 分析软件中是否存在恶意的第三方组件, 会进行恶意的行为. 作者设计了 Buildwatch, 用于动态分析软件及其第三方依赖

组件的框架, 发现存在恶意第三方组件的软件在安装过程中会引入大量新的第三方组件. 基于这一发现, Buildwatch 将测试软件运行在沙箱环境中, 分析软件的系统调用, 主要是分析测试软件在安装过程中表现出来的行为来确定软件可能已经被恶意的第三方库感染.

基于上述调研, 关键的应用软件中第三方组件风险识别方法总结如表 4 所示. 可见针对软件系统中的第三方组件的风险识别研究中, 当前的研究方案能够支持白盒和黑盒场景的分析, 但是在抗混淆和可传递性上仍有较大研究空间. 尤其考虑到当前软件依赖关系层层嵌套, 十分复杂, 亟需研究具有高效可传递性的第三方组件依赖检测方案. 此外, 当前的研究方案都针对特定已知的攻击设计, 在发现新漏洞的能力上仍有所欠缺. 最后, 新的开发和应用场景随着现代化的进展也在不断涌现, 当前研究对物联网分发市场的研究仍处于初级探索阶段, 对新的开发和应用场景的第三方组件风险检测研究仍有待研究, 如多架构的物联网设备固件、人工智能平台和模型中也依赖大量第三方组件, 且存在安全风险.

表 4 经典的应用软件中第三方组件识别及漏洞挖掘方法总结

方法名称	检测的开发语言	分析方法		分析场景		性能		可扩展	抗混淆	可传递	支持识别的缺陷
		静态	动态	黑盒	白盒	检测能力	检测速率				
OWASP Dependency Check[89]	Java, NET,Python, Ruby, PHP, NodeJS	√	×	×	√	○	●	◑	○	○	1day 漏洞
VAS[90]	Java	√	×	×	√	○	●	◑	○	○	1day 漏洞
LIBSCOUT[91]	Java	√	×	×	√	◑	◑	◑	◑	◑	1day 漏洞
Atvhunter[92]	Java	√	×	×	√	●	◑	◑	◑	○	1day 漏洞
CENTRIS[93]	C/C++	√	×	×	√	●	◑	◑	◑	●	1day 漏洞、许可证违规
LibD[94]	Java	√	×	×	√	◑	◑	◑	●	◑	\
Eclipse Steady[95]	Java, Python	√	√	×	√	●	◑	◑	◑	○	1day 漏洞
OSSPolice[96]	Java	√	×	√	×	◑	●	◑	●	○	许可证违规、1day 漏洞
Buildwatch[97]	JavaScript	×	√	√	×	◑	○	◑	●	\	恶意的第三方组件

注:●=高, ◑=中, ○=低,\=不具有该特性; √=满足, ×=不满足; 检测能力是对检测准确率和误报率的综合评价; 可扩展是方法针对不同开发语言扩展能力的综合评价, 可传递是指方法能识别依赖的组件,并进一步识别依赖的组件和别的组件的依赖关系.

3.2.2 应用软件代码复用检测

除了第三方组件带来的 1day 漏洞及许可证违规风险, 研究人员也探索了开源软件供应链中代码复用带来的 1day 漏洞及许可证违规风险. 现有研究重点探索了软件代码的新型表示, 进一步结合人工智能技术等完成代码克隆检测及风险的识别.

针对源代码, Fischer 等人[37] 研究了安卓市场中对 Stack Overflow 里分享的代码重用的情况, 先扫描 Stack Overflow 中的代码并利用基于特征的匹配如使用过时的加密算法等来标记代码是安全的还是不安全的. 最后, 在安卓程序中搜索是否有代码块和标记为不安全的 Stack Overflow 的代码块相似. 这一搜索过程主要是将安卓程序和不安全的代码块表示为具有语义的特征向量, 通过计算两个向量的相似度来确定两个代码块间的相似性. 该方法具有较快的检测速率. 然而, 一旦复用的代码经过了一些混淆的设计, 准确率会大幅度下降. Akram 等人[98]设计实现了 SQVDT, 通过收集易受攻击的代码数据集并提取不同的特征来构建指纹签名集, 最后对项目中的文件进行相似性匹配, 识别潜在的 1day 漏洞. SQVDT 主要的优点在于利用 MapReduce 在大规模集群上进行快速分析和处理, 在相似性检测的效果上准确率较高, 但召回率较低. Donovan 等人[99]则人工地

审计了 Github 上的 C++代码对 Stack Overflow 上的代码复用情况，发现了 69 个易受攻击的代码片段，该代码片段已被迁移到 2800 多个项目中. 基于手动审计的方法具有较高的准确率，但是效率十分低下,严重依赖专家知识. 研究人还研究了人工智能技术辅助的代码克隆检测来帮助提高代码复用问题检测的检测速度和准确率. Bilgin 等人[100]探索了对代码的编码表示，再结合人工智能对代码进行训练,并完成漏洞预测的下游任务.但他们的研究较为初步 ,仅仅利用了软件源代码中的函数信息，无法很好地编码代码中的控制流图信息.

针对二进制代码,研究人员主要结合了代码相似性分析的方法. 当前的方法主要依赖于近似图匹配算法，效率和准确率都比较低. Xu 等人[101]针对物联网固件的二进制代码设计了 Gemini 来识别固件复用的函数代码. 如图 7 所示, Gemini 的设计是为了检测来自两个不同平台的二进制函数是否相似，能够帮助检测相似的第三方组件进一步检测漏洞、许可证违规等. Xu 等人将深度学习应用到二进制代码上，基于函数的控制流图来计算代码向量，基于两个函数代码向量之间的距离来进行相似性的检测. 经过实验, Gemini 在相似性检测的准确度上和效率上有所提升. 相比 Gemini, Ding 等人[102]提出的 Asm2vec 和 Li 等人[103]提出的 PalmTree 都对二进制程序表示的改进，考虑了二进制程序中编译优化选项和代码混淆技术导致汇编函数不同的难点.

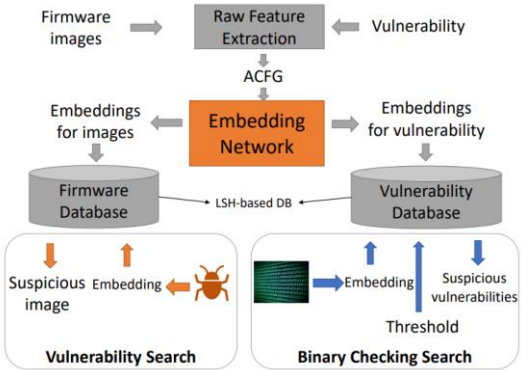

图 9　基于神经网络的第三方组件检测方案 Gemini 的工作流图[101]

综上所述，开源软件供应链视角下的应用软件代码克隆检测主要目的是为了检测软件中的 1day 漏洞及许可证违规的风险. 当前的研究在人工智能技术的辅助下，在源代码和二进制代码的测试上，准确率都有一定提高，但在混淆后的代码上的检测能力仍显不足. 且上述研究仅仅对代码表示进行了优化的设计，而上述代码表示方法在下游的漏洞检测及许可证违规检测的能力仍有待研究.

3.2.3　分发市场中的恶意应用软件识别

考虑到开源软件供应链的分发环节中攻击者会对安全软件进行恶意篡改或者添加恶意载荷，研究人员探索了如何利用软件智能风险检测技术来识别这些恶意的软件.

基于元数据的方案是最简单的方案，检测速度快，但准确率低. Hemel 等人[104]研究了基于元数据的相似恶意软件的检测，并开发了 BAT，用于检测二进制代码相似. BAT 主要是检测物联网固件源代码和二进制格式的软件代码包，通过分析两者是否字符串、压缩数据的大小、二进制增量的数据大小是否一致来实现检测. 该方法较简单，检测速率高，但是存在大量误报和漏洞.

基于相似度函数的相似恶意检测主要是利用哈希作为相似度函数进行比较，依然保持较快的检测速度，但比基于元数据的方案在准确率上提高了很多. Zhou 等人[105] 首次发现安卓应用分发市场中有攻击者会对应用程序进行重打包，然后嵌入一些新的广告，来窃取广告或者重新路由广告收入. 基于这一发现, Zhou 等人研究并开发了一个应用程序相似性测量系统 DroidMOSS，来检测这些重打包的应用程序并进一步分析潜在的风险. 如图 10 所示，DroidMOSS 主要应用了模糊哈希技术对程序的指令序列进行计算，形成应用的指纹，然后通过计算相似性函数来定位和检测应用程序重打包行为. DroidMOSS 具有较快的分析速率，但在经过混淆的代码上准确度显著降低. 与上述基于哈希的代码相似性检测方案不同，Golubev 等人[106]采用基于 SourcererCC[107]的相似性检测，其中相似性函数利用了 Sajnani 等人[107]设计的针对代码块的比较算法.

Golubev 等人将该方法应用到软件许可证违规的检测上，取得显著的效果.

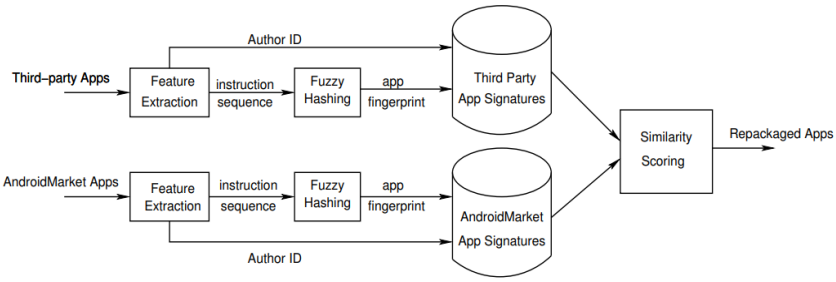

图 10　应用程序重打包检测系统 DroidMOSS 框架[105]

基于人工智能的代码相似性比较是当前比较有效且热门的研究方案，但对训练要求高，训练花费时间长. Crussell 等人[108]研究并实现了 Andarwin，不需要遍历所有程序并进行任意两个程序的对比，通过多个聚类来有效处理大量应用程序，大大提高了可扩展性和效率. 如图 11 所示，Andarwin 先将程序表示为一组在应用程序的程序依赖关系图上计算的向量，接着对所有应用程序的所有向量进行聚类找到相似的代码段，考虑完全和部分应用程序的相似性. 该方法虽然提高了效率和可扩展性，但仍然无法很好地对经过混淆后的代码进行检测. 为了解决这个问题，Gonzalez 等人[109]提出了 DroidKin，能够在多种混淆级别下检测应用程序的相似性. DroidKin 通过对程序元数据和字节码提取特征形成特征向量，基于特征向量对给定程序的潜在候选者进行识别和评分，最后检查相似应用之间的所有可能关系并根据所设计功能的类型进行优先级排序. DroidKin 利用了表示代码的低级语义，一定程度上缓解源码级别的混淆. 实验表明 DroidKin 在检测更改量较大的程序之间的相似性检测效果具有显著效果.

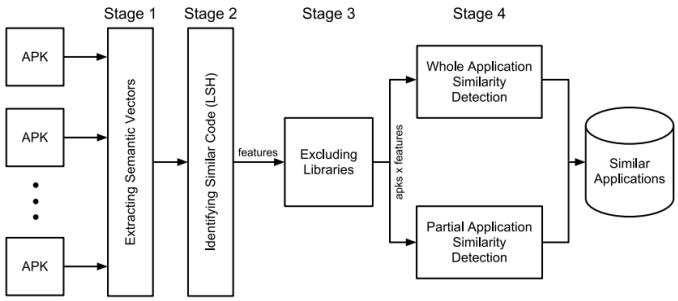

图 11　应用程序相似性检测 Andarwin 框架[108]

3.3 协作开发的风险识别

开源软件项目往往需要多个开发人员、维护人员共同持续开发维护，这涉及到开源软件项目的源码管理工具，也带来了新的风险渗入点，攻击者利用了持续集成(CI)/持续交付(CD)管道和开发工具中的弱点注入隐蔽的安全漏洞、窃取开源软件的隐私信息. 为了解决上述安全风险，研究人员开展了代码仓库接受开发维护提交请求的评估、识别协作开发过程中代码提交的风险和隐私数据泄露的研究等.

针对代码提交请求中的代码风险，Wan 等人[110]深入研究了基于深度学习的高效风险识别方案. Wan 等人首先对 GitHub 开源生态中一些流行的开源存储库(如 Linux)进行了大规模的 Git 提交爬取，其次开发了一个基于 Web 的分类系统，供安全研究人员手动标记提交，基于深度神经网络对提交消息自动识别恶意修复提交 (VFC). 该方法比最先进的方法取得了明显更好的精度，也提高了召回率. 相比 Wan 等人提出的需要大量数据标记的方案，Gonzalez 等人[29]设计了一种更具有扩展性且自动化的恶意提交检测方案 Anomalicious. Gonzalez 等人指出在开源软件存储仓库 GitHub 上，利用提交的日志记录和代码存储仓库元数据来自动化检测异常和潜在的恶意提交. 如图 12 所示，作者基于上述数据来挖掘是否存在这些潜在的敏感行为或者影响因

素，如敏感文件的修改、异常值更改属性、提交作者的名誉等. 接着，Anomalicious 应用了基于规则的决策模型来自动计算和分析敏感行为或者影响因素，最终进行判断是否是恶意的提交. 实验表明，Anomalicious 在对 15 个受恶意软件感染的存储库的数据集的评估中识别了 53.33% 的恶意提交. 更针对性的代码提交过程代码风险检测是 Andrade 等人[111]的工作. 他们提出了一个工具 Salvum 来自动化发现提交代码是否存在访问隐私文件的风险. Salvum 要求用户使用他们的策略语言编写约束，再基于信息流控制 (IFC) 的工具实现对 Java 源代码的静态分析，并判断代码对用户编写的约束的遵守情况. 作者将 Salvum 应用到五个基准项目中检测违规行为来衡量其有效性，发现 80 项违规行为. 但 Salvum 的可扩展性较差，需要用户有较多的专家知识. 2021 年，明尼苏达研究人员发现了协作开发模式下代码提交存在新的风险[112]. 该研究发现了一种”伪装者提交”的风险，也就是研究人员提交一系列看似无害的代码补丁，但是当代码被集成到软件中后，提交的代码和已存在的代码可以共同作用导致引入一些漏洞. 该研究揭示了开源协作新模式下持续集成的新风险. 针对这一问题，OSSFuzz[113]和 ClusterFuzzLite[114]研究了如何持续地对代码提交进行模糊测试. OSS-Fuzz 支持在开发过程中迅速地部署模糊测试工具，而 ClusterFuzzLite 支持针对代码提交更改进行快速且持续地模糊测试.

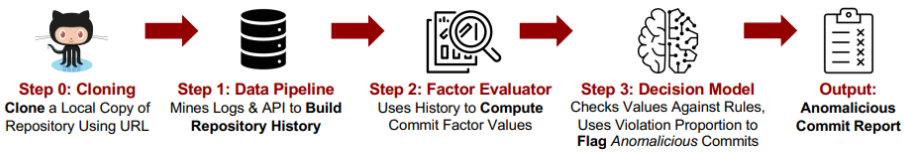

图 12 恶意代码提交的检测框架 Anomalicious[29]

相比上述研究考虑的是代码仓库恶意代码提交的风险，研究人员也探索了代码提交请求中隐私泄露的风险，即作者在提交代码时无意泄露自己的隐私的风险，并研究了相应的检测方案. Sinha 等人[115]早在 2015 年就关注到恶意用户窃取嵌入在公共源代码仓库(如 GitHub 和 Bit Bucket)上托管代码中的 API 密钥以免费运行自己部分项目或者窃取用户资产的风险事件. 针对这一风险,Sinha 等人对开源代码仓库的用户提交中密钥泄露问题探索了四种检测方案.

（1）基于关键词的搜索.当前项目密钥一般会有独特的声明字段,例如“—BEGIN RSA PRIVATE KEY—“通常出现在 SSH 密钥的头部. 基于这一发现,可以通过收集相应密钥使用的关键词字段并进行批量搜索，用于检测密钥的泄露. 该方法检测速度较快，但存在较高的误报率和漏洞率，只能搜索到关键词，无法确定密钥是否已经进行脱敏处理，且无法搜索到没有相应独特声明字段的密钥泄露.

（2）基于模式匹配的搜索. 一些密钥除了有独特声明字段外，自身的格式也十分独特. 基于这一发现，作者将 Java 源文件生成抽象语法树 (AST)，并对字符串文字节点执行模式搜索. 与基于关键词搜索方案相比，该方案能够减少 14 个与 Client ID 模式匹配的代码中的 30 个实例，以及准确识别到与密钥模式匹配的 17 个项目中的 43 个实例. 虽然基于模式匹配的搜索在搜索效率上略有下降，但大大降低了误报率和漏报率. 但该方案需要手动定义规则，且无法支持没有特定格式的密钥的识别，仍存在大量漏报.

（3）基于启发的密钥泄露检测过滤.在上述两种方案的基础上，作者探索了一些额外的启发式规则来帮助过滤一些误报. 例如考虑到客户 ID 和密钥通常成对出现，添加额外的检查搜索客户端 ID 和密钥是否出现在 5 行之内. 虽然这种方法通常是精确的，但很可能会遗漏客户端 ID 和密钥关系不紧密的密钥泄露实例，且很大程度依赖客户端 ID 和密钥识别的准确度. 在匹配 API 密钥模式的字符串中减少误报的另一种方法是尝试猜测它们是自动生成的还是手写的. 作者注意到许多误报实际上是人类可读的字符串,例如“SomeLabelTextInMyAppWhichIsAPerfectMatch”，或占位符,例如“00000000...”. 为了解决这个问题，作者应用了一个标准的密码强度估计器，过滤掉具有重复字符或包含字典单词的字符串，降低了误报.

（4）基于源程序切片的检测. 准确查找流入客户端 API 调用的那些字符串的重量级方法是使用程序切片，但传统的程序切片有很大的时间消耗. 作者设计了一种轻量级的流敏感程序切片算法，切片标准是密钥相关函数的调用，进一步对返回的密钥进行密码强度分析. 该方法实现了 100%的准确度和 84%的召回率.

然而 Sinha 等人提出的方案仅仅关注 AWS 的密钥泄露检测, 具有较大的局限性. Meli 等人[116]则首次对开源代码仓库 GitHub 的密钥泄露进行了大规模的纵向分析, 并提出两种互补的方案进行密钥泄露检测. Meli 等人的方案主要是基于模式匹配和启发式过滤的方案进行自动化检测, 支持检测私钥文件、具有独特 API 密钥格式的 11 个高影响力平台的密钥泄露. Meli 等人收集了数十亿个文件,包括近六个月的实时公共 GitHub 提交扫描和覆盖 13% 的开源存储库的公共快照, 结果发现不仅密钥泄漏普遍存在, 影响超过 100,000 个存储库, 而且每天都有数千个新的、私人的密钥被泄露.

综合上述研究分析可见, 关于协作开发的风险识别的研究较少, 考虑到开源生态环境的来自不同开发人员的代码提交数量多、项目提交动态更新等特点, 当前仍没有较好的方案能实现大规模、自动化且可扩展的风险识别方案, 也无法支持动态增量式的轻量级的风险识别.

3.4 下载更新过程的风险识别

面对用户在下载和更新软件时可能存在网络劫持、钓鱼网站等风险, 研究人员主要研究了更新下载渠道的风险识别. Garrett 等人[117]提出异常检测的方案, 来识别恶意更新的 Node.JS 软件包. Garrett 等人提出的方案主要利用了机器学习技术, 通过学习正常软件包更新的特征, 训练模型分辨恶意更新的数据包. 相比 Garrett 等人的方案, Teng 等人[118]更加聚焦网络劫持的风险识别. Teng 等人指出当前开源软件供应链中软件下载更新环节缺乏对更新信息和更新包的认证是引入风险的根本原因. 因此, Teng 等人设计了一种基于流量分析的软件更新漏洞自动检测和验证方法. 该方法通过提取软件更新过程中的网络流量, 对升级机制进行自动画像, 并将其与漏洞特征向量匹配, 来预判潜在的漏洞.

在开源软件供应链中, 软件下载更新渠道的风险识别方案研究较少, 是因为研究人员希望从本质上提升加固软件下载更新渠道的安全性.

4 开源软件供应链的加固与防御

除了对上述风险的被动检测研究外, 研究人员探索了多种针对开源软件供应链的主动加固和防御方案. 针对开源软件供应链的各个环节, 研究人员分别针对各个环节研究了相应的主动加固方案, 同时也有一部分研究旨在对开源软件供应链多个环节或整个模型设计防御方案. 下面将围绕以上两点展开总结和分析.

4.1 针对开源软件供应链单个环节的加固防御

当前面向开源软件供应链单个环节的加固防御研究, 主要集中在以下四个方面: 组件及应用软件开发环节中可信开发方案的设计、组件及应用软件开发环节中代码的安全加固、应用软件分发环节中可靠的分发方案和应用软件使用环节中安全的使用方案.

4.1.1 可信开发

针对可信开发, 本文围绕协作开发过程和开发依赖介绍了工业界及学术界的可信开发研究.

针对源代码构建, 考虑到协作开发过程中可能有恶意攻击者冒充开发人员提交恶意代码, 业界实践及学术研究最普遍实用的加固防护机制是代码跟踪校验. 下面以最流行的开发协作平台 GitHub[119]和著名软件厂商微软[120]为例介绍业界如何应用代码跟踪校验. 其中 Github 支持开发人员和维护人员协作提交代码请求. Github 要求对每个提交都进行代码提交的签名并对签名进行验证来实现代码提交的可信性. 此外, Github 也提供对项目软件的版本管理[121]和控制. 微软则通过提供可信发行商证书让开发人员对受信任的代码发布者的签名信息进行验证. 在完成源代码构建后, 研究人员需要进一步对软件进行编译、生成、测试. 尽管源代码是安全的, 编译、生成、测试过程中可能引入新的漏洞来破坏用户系统. 基于这一风险, 研究人员研究安全策略来实现对该过程的加固. Lamb 等人[122]创新性地提出一个可复制构建(reproducible builds)的方案. 可复制构建的关键思想在于构建一个给定的源代码树会生成逐位相同的结果, 可以通过比较从多个独立构建起输出的结果来确定构建器是否可信, 来确定二进制程序是否和其源代码一致. Ullah 等人[123] 针对持续部署管道设计了 5 种安全策略, 通过控制开发人员对代码管理器等主要组件的访问以及建立组件之间的安全连接来帮助实

现高效安全的软件部署管道. Bass 等人[124]考虑了整个开发过程的安全性, 指出在开源软件供应链环节中, 有一个关键阶段是从输入、包和构建的测试中构建应用程序, 并将应用程序包放置在许多物理或虚拟机上. 这些阶段中的任何一个过程都可能引入漏洞, 例如部署的系统部分可能不是所需的部分; 构建、打包或部署可能已损坏. 这过程中涉及到各种各样的工具, 其安全机制也各不相同. Bass 等人提出的方案是为了对这一复杂的阶段进行加固, 通过对代码构建、编译、生成、测试和部署的每一个环节添加校验来保证程序的完整性, 帮助实现安全可信的开发过程.

针对开发依赖, 考虑到分发市场可能存在恶意或有缺陷的第三方组件, 工业界往往采用建立自己的开源第三方组件管理库, 如领英公司拥有良好的外部第三方组件管理模式[125], 只有第三方组件通过企业的安全审核后再进入企业的第三方组件管理库中. 企业内部开发人员在开发过程中会从该管理库中安装依赖, 在可控范围内管理组件依赖, 防止引入存在漏洞或恶意逻辑的组件, 同时也防止与非官方渠道通信过程中遭受攻击.

上述研究主要利用了密码学数字签名、可信证书技术, 通过对开发过程中的协作人员行为的验证、开发过程中关键信息的完整性及对第三方组件的评估管理来实现可信开发, 其中对协作人员行为及关键信息完整性验证主要利用数字签名及证书, 较为高效, 已被广泛应用. 然而上述研究方案主要基于密钥来保障签名和证书的安全性, 一旦签名的密钥被泄露或攻击者基于源代码集成环境的缺陷绕过验证, 上述方法将不再有效.

4.1.2 代码加固

针对组件及应用软件开发环节中的代码加固研究, 研究人员主要从减少攻击风险及构建代码补丁两方面展开了研究. 在减少攻击风险的研究中, 研究人员主要关注的是开发过程中是否引入不必要或存在问题的第三方组件, 同时也关注开发和部署的完整性. 为了实现开发过程中的安全, Koishybayev 等人[126]提出 Mininode, 帮助对 Node.Js 软件进行自检, 并减少软件的攻击面. Mininode 的主要思路是对软件依赖关系图进行检测, 并删除没有使用的代码和第三方依赖库, 减少软件攻击面. 相比软件开发后的依赖检测, Nguyen 等人[127]涉及实现了一个轻量级的为安卓开发实现的扩展, 可以在研究人员开发过程中实时提供自动化的快速修复, 识别开发项目中过时的依赖项, 实现代码加固. 但该研究无法保证替换的新的第三方组件和当前开发的代码是否有功能上的短缺和冲突, 且不支持自动化提取已发布的针对代码依赖项的漏洞并通知开发者, 也不支持提取代码依赖项的隐私风险信息并通知开发者. Vasilakis 等人[128]提出了 BreakApp, 支持开发者在开发过程中限制引入的第三方组件行为. 通过 BreakApp, 可以将引入的第三方组件进行模块化地分离, 选择性地禁止某些模块行为来减少攻击面. 上述研究仅仅是针对 JavaScript 库进行加固, 这是由于这些解释性语言在引入和使用第三方库功能的代码实现上更加结构化, 且比较容易和个人实现的代码区分开. 上述研究均指出他们的方案暂时还无法轻易扩展到编译型的开发语言, 如 C 语言和 Rust 语言上. Vasilakis 等人[129]在 2021 年最新的研究中尝试了对基于 C/C++语言开发的组件及应用软件的代码加固, 设计了一个针对字符串处理的第三方组件的安全加固, 通过观察组件正常运行情况下的行为来重新生成一个安全的组件版本. 该加固方案可以在生成安全的第三方组件并应用到开发过程中, 也可以用安全生成的第三方组件替换到潜在的恶意第三方组件. 该方案是针对供应链视角下对 C/C++语言语法的组件及应用软件的代码加固的一个良好实践, 但目前该方案仅仅支持字符串处理相关的, 其他重要的如加密计算相关的第三方组件的加固方案仍值得进一步探索.

此外, 软件的补丁研究是开发过程中的重要加固技术. 在开源软件供应链视角下, 漏洞会从上游传播到下游, 下游组件及应用软件的补丁难以第一时间得到响应, 相关研究面临着新的风险和挑战. 一方面, 下游厂商也无法确定含有大量组件依赖和代码复用的组件及应用软件中是否修复了哪些漏洞, 继承的组件依赖及复用的代码中是否已经修复了相关代码. 另一方面, 当一个漏洞披露时, 下游厂商难以确定当前依赖的组件或复用的代码是否受到影响, 且受到影响的组件及复用的代码是否已经进行了修复. 因此, 针对上述问题, 在研究软件的补丁过程中, 需在对开源软件供应链漏洞影响的评估、对软件是否存在补丁的评估和自动打补丁的研究. 针对漏洞评估, 安全开发人员和维护人员在发现新漏洞时, 需要对其进行评估, 以便对其进行优先级排序进行打补丁. 基于 CVSS 度量标准[130]是漏洞评估流行的标准之一, 主要基于可利用性和影响度量衡量严重性, 但缺失对访问复杂性的评估. 此外, 也有研究基于攻击面入口点和到达能力更全面地评估漏洞可利用性

风险[131]，另一部分研究人员从补丁程序出发，通过分析补丁引入的代码更改来评估漏洞的影响[132] [133]. 上述研究得到的任务补丁的效果和违反安全规则的影响可以建模为需要自动解决的约束问题，然后利用了基于规则的比较和符号执行等对漏洞进行分类和评估. 除了漏洞的影响力评估外，安全开发和维护人员需要进一步确认漏洞的影响版本，补丁的定位,以及下游组件及应用软件是否部署了该补丁. Zhang 等人[134]提出了 Fiber. Fiber 首先仔细解析和分析开源安全补丁，然后生成细粒度的二进制签名，忠实地反映补丁引入的最具代表性的语法和语义变化，用于搜索目标二进制文件. 与之前的工作相比，Fiber 主要关注补丁和最小上下文的小变化，而不是整个功能或文件，提高了效率. 针对补丁存在性研究，Jiang 等人[135]提出了基于语义的补丁存在性检测框架 PDiff. 如图 13 所示, PDiff 基于程序代码的语义总结，将补丁采用前后的目标内核与其主流版本进行比较，优先选择更接近的参考版本来确定补丁状态. 与之前对补丁存在性测试的研究不同，该方法基于补丁的语义来检查相似性，因此对代码级变化提供了很高的容忍度. 相比上述方法需要源代码，Dai 等人[136]提出了 BSCout，直接检查 Java 可执行文件中是否存在整个补丁. BScout 使用整个补丁识别漏洞是否存在，不需要特征构建，通过建立 Java 字节码对源代码的链接，实现对整个目标可执行文件中的细粒度补丁语义的准确检测.

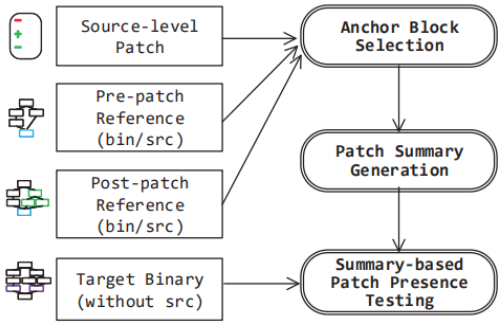

图 13　补丁存在性检测 PDiff 框架[135]

为了减少人工的参与度及减轻对大量下游组件软件打补丁的压力，软件自身加固的研究主要集中在自动打补丁的研究上. Chen 等人[137]研究并提出了 KARMA，是一个适用于 Android 内核的自适应实时修补系统. KARMA 中的补丁可以放置在内核中多个级别来过滤恶意输入，且自动适应数千种 Android 设备. Duan 等人[138]提出并实现了 OSSPATCHER，能够直接从源代码补丁构建功能级的二进制补丁，并在发现漏洞的应用程序上执行补丁. 考虑到系统可能将很多开源软件和大量的补丁捆绑在一起，而攻击者可能趁机引入一些不安全的补丁，需要区分安全的补丁和不安全的补丁. Wang 等人[139,140]开发了一个基于机器学习的工具集帮助来识别安全补丁和存在漏洞的补丁.

上述研究展示了开源软件供应链视角下安全加固的两个重要研究方向，一方面从代码层面添加了加固的设计，来减少风险，增强应用软件自身的防御能力，但当前开发过程中自动化分析并处理依赖项的工具仍较匮乏，相关代码加固研究在更广泛的 C/C++语言开发的组件及软件上以及支持更复杂功能的第三方组件上仍处于初步阶段，需要更深入的探索. 一方面，针对开源软件供应链的补丁研究仍有许多亟待解决的难题，如更准确的漏洞影响范围的评估、更高效的补丁存在性检测、自动化持续性的打补丁研究等.

4.1.3　可靠分发

当前应用软件的可靠分发，主要目标是保障应用软件的完整性和可追溯性. 观察分发环节的业界实践，本文发现分发市场和应用商店通常设计并实现软件审核和检测机制以对抗伪装、虚假的应用软件，如苹果的应用商店提供了严格的审查流程以规范应用程序的发布[141]，需要对应用程序进行 API 的扫描、代码相似性的检测以及人工审核测试程序是否能够正常运行且没有安全风险. 在 JavaScript 语言生态中，最流行的包管理器 npm 提供了自动的项目漏洞扫描和兼容性更新修复的功能，用于遏制组件依赖中的漏洞影响到当前软件包[142]. 此外，第 3.4 节的软件风险识别方案也可以帮助软件分发市场实现对软件的审核和检测.

研究人员也开展了一些关于抵御重打包攻击的研究, 主要包括自签名策略和代码混淆. 其中, 自签名策略指的是在发布软件的过程中, 软件包的哈希值和开发者签名也需要被发布, 让用户下载软件时可以对软件进行完整性的验证. 如 Android Studio 在发布他们的终端软件的页面上[143]也列出了每个应用软件版本相对应的 SHA-256 的验证码. 代码混淆能够使得源代码和机器代码的逆向更加困难, 使得攻击者在篡改软件代码时更加困难. Proguard[144]是安卓提供的一个基础的混淆工具, 可以实现对源代码的混淆, 包括把类名、方法名和成员变量等变为无意义的字符串, 但是由于代码逻辑没有改变, 攻击者仍然能够根据分析软件逻辑推测类和方法所扮演的角色. Song 等人[145]设计了一个增强的对抗重打包的系统, 设计了一个具有时间多样性的联锁保护网来防止安卓应用程序的篡改, 会根据相应的威胁模型随机构建不同结构的防护网, 利用了 Java 原生接口的跨层调用机制为代码、核心算法和敏感数据提供多级保护.

上述研究展示了保障应用软件可靠分发的两个重要加固手段. 一方面需加强应用软件分发时的安全审核, 需基于第 3.2 节提出的软件风险识别技术来提高审核能力. 一方面需加强应用软件抵御重打包攻击的能力, 代码混淆对抗仍是当前软件工程及网络空间安全研究的重要议题, 如何结合智能技术如混淆逻辑门、深度学习等来加强代码混淆能力以及对代码混淆的评估仍是重要研究方向之一.

4.1.4 使用安全

针对使用环节, 研究人员主要研究安全运行、安全下载更新这两个方面. 针对安全运行, 研究人员提出了多种可信运行环境, 使得应用软件运行时免遭攻击者的威胁. 当用户运行时, Gregor 等人[146]设计了一种基于可信计算环境的应用软件. 通过该系统可以运行在不受信任的环境中. 该系统结合了很多现有的安全机制, 如 TLS 协议帮助安全传输, 设计访问控制策略限制访问接口等. 该系统能够对应用软件的代码和数据进行秘密管理, 保证系统的执行没有受到攻击者的恶意篡改.

针对安全更新, Ozga 等人[147]提出了 TSR, 是基于可信运行环境的进一步应用研究, 结合 SGX 提出的安全更新方案 , 能够保证软件包升级的机密性和完整性. Karthik 等人[148]提出了一个针对汽车的安全更新框架 Uptane, 专为智能汽车实现固件更新, 使用了多个服务端来对要下载的固件进行检验. Uptane 设计了控制服务器来管理固件映像决定哪些固件映像需要被安装和验证, 设计了时间服务器通过从车辆接受令牌并返回包含令牌和当前时间的签名序列来通知车辆当前时间, 避免重放攻击.

基于上述研究可见, 结合一些新兴的技术如可信计算等进行拓展应用来保障安全运行和下载更新, 针对新场景下使用环节的加固防御研究是当前主要的两个研究方向.

4.2 面向多环节的可靠开源软件供应链方案设计

研究人员面向开源软件供应链提出了多种面向多环节的设计方案. 当前的研究方案主要考虑到组件及应用软件开发环节和应用软件分发环节之间的供应关系, 设计对组件及应用软件开发和分发过程进行全面管理和监控的方案, 来使得开发人员或终端用户可以从分发市场上下载可信安全的组件或应用程序.

针对该目标, Singi 等人[149] 提出一个可信的软件供应链治理框架, 把软件供应过程中产生的所有数据包括依赖的第三方组件、开发人员对应开发的代码等存储在区块链中, 并进行统一的表示, 接着以智能合约的形式指定合规/监管规则和最佳实践, 并分析事件数据以识别不合规问题并发出警报. 该框架允许在整个应用程序开发生命周期中记录、监视和分析各种活动, 从而使开发过程透明、可审计、遵守法规和最佳实践, 从而实现软件的可信赖性. 该方案主要从数据提取和风险监控来实现对开源软件供应链的管理, 该方案利用区块链开销大, 利用智能合约实现的检测功能有限.

除此之外, 有研究积极探索了开源软件供应链的框架设计. 研究方案 in-toto [19] 则关注组件及应用软件从开发到分发过程中代码的完整性, 并保证开发人员可以按预期完成开发过程. in-toto 定义了项目管理者, 功能开发维护者和终端用户三个角色. 其中, 项目管理者会设置称为布局的数据即.po 文件, 布局数据定义了软件项目在软件供应链开发环节中的每一个步骤包括代码书写、测试和分发等涉及的人员及资源信息. 如图 14 所示, 项目管理者可以定义由三个步骤组成的供应链: 标记、构建和打包步骤. 项目管理者还定义了资源将如何在供应链中流动. 之后, 项目管理者可以分配工作人员来执行这些步骤中. 负责执行该任务的软件开发维

护者需要使用布局中定义的私钥数据对每个环节中的输入输出和执行过程进行签发，作为供应元数据．当产品交付到终端用户时，终端用户可以对以下几点进行校验：(1) 供应链布局由项目管理者签发，并且尚未过期(时效性)；(2) 检查供应元数据是否和布局一致，验证供应的每个步骤是否按预期执行；(3) 检查供应元数据确认供应链的完整性．

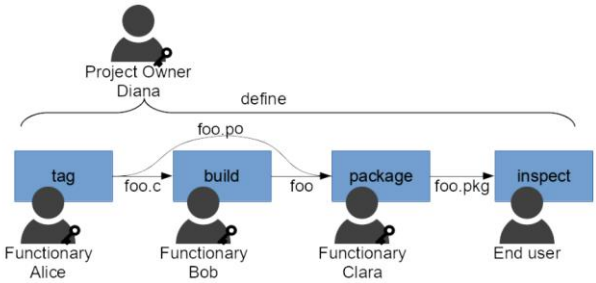

图 14　添加了 in-toto 元素的软件供应链的图形描述[19]

该方案主要基于密钥来保障签名的安全性，一旦签名的密钥被泄露或攻击者基于源代码集成环境的缺陷绕过验证，该方法将不再有效．针对这一缺陷，研究人员探索了增强的可信开发过程的管理方案，一大研究方向是基于多级密钥管理来降低密钥泄露的风险，另一个研究方向是基于区块链的方式来实现可信开发管理方案．

基于多级密钥管理的研究方案主要是由一个根密钥管理者生成并保存一个密钥，基于该密钥向下级开发人员或维护人员派生分发相应的密钥，来将对代码修改维护等操作的权限委托给可信的协作开发人员．基于该思路，Samul 等人[150]提出了 TUF，定义了不同的用户角色和信任委托机制，对开发过程中的角色包括源码管理仓库的管理者、协作开发人员以及下游组件或应用软件的使用者进行划分和限制，防止各环节参与方权限滥用、破坏开发的组件及应用程序的完整性．它定义了在密钥泄露的情况下如何实现可信开发．他定义了不同的用户角色和授权，在密钥泄露的情况下，基于多级密钥管理实现的信任层次结构使攻击者难以破坏整个系统．然而，TUF 给社区存储库带来了巨大的可用性挑战，因为它要求开发者在交付一个组件或应用软件视就需要实时在相应的分发市场向该框架进行注册登记．为了克服这一点，Kuppsamy 等人[151]提出了 Diplomat．然而，Diplomat 框架容易受到回滚攻击．Kuppusamy 等人[152]在后来提出的 Mercury 框架中修复了回滚攻击的影响，它使用一种新技术来紧凑地传播版本信息，同时仍然可以防止回滚攻击．由于处理密钥撤销的技术不同，即使软件存储库遭到破坏，用户也可以免受回滚攻击．Mercury 需要较少的带宽，使用增量压缩仅仅传输先前和当前版本信息列表之间的差异时来降低传输开销．与上述工作类似的还有 Brown 等人[153]提出的 SPAM 和 Cappos 等人[154]设计的 Stork 框架．SPAM 主要解决密钥重用和密钥攻陷的问题，支持自动化管理开发协作人员，Stork 主要研究如何实现基于角色的最小信任量委派，以及考虑到开发的组件及应用软件可能需要动态更新或撤销，研究如何在密钥分配管理中支持动态的密钥派生和撤销．

上述基于多级密钥管理的研究方案主要贡献在于降低了密钥泄露的风险，而新兴的基于区块链的研究方案则进一步加强了开发过程的透明性，同样积极探索了动态的密钥派生和撤销以及开发过程中引入风险识别知识来加强开发代码的安全性等研究．Nikitin 等人提出了 SkipChain [155]，这是一种类似于区块链的分布式数据结构．SkipChain 需要协作开发人员均可信，使用了集体签名协议来签署开发的组件或应用软件．SkipChain 将开发过程中的所有相关信息都存储到区块链中，保证了开发流程的透明性，但该系统没有考虑软件供应链中开发过程的动态性，没有解决繁琐的证书撤销问题．为了解决动态的证书撤销和派生问题，Stengele 等人[156]基于以太坊区块链，提出了一个用于发布和撤销二进制文件完整性保护信息的方案，设计了智能合约用于强制执行对完整性保护信息的发布和撤销的访问控制．以太坊区块链用作已发布和已撤销二进制文件的防篡改、可公开验证的日志，提供开发过程的透明性．在 Stengele 等人的基础上，Guarniz 等人[157]提出了 SmartWitness．该方案除了能够提供开发组件及应用软件的透明性、高效且细粒度的密钥撤销外，还动态将一

些安全检测方法构建为智能合约, 实现对开发代码安全性的评估, 并作为元数据添加保存.

基于上述研究,可见当前针对开源软件供应链多环节的管理方案主要有以下两个方向: 从设计上加强开源软件供应链结构上的改进, 来实现组件及应用软件的可信开发及可信分发目标; 结合智能技术针对开源软件供应管的知识管理、风险监控进行进一步探索.

5 开源软件供应链的法律政策

5.1 国家层面的法律政策

目前, 多个国家已经制定了针对开源软件供应链的法律政策. 在 2015 年, 美国国家标准技术研究院(NIST)专门制定 SP800-161《联邦信息系统和组织供应链风险管理/方法实践清单》[158], 用于指导美国联邦政府机构管理 ICT 供应链的安全风险, 旨在指导联邦部门和机构识别、评估和减轻 ICT 供应链风险. 2018 年, NIST 发布 Security Considerations for Code Signing[159], 该报告指出: (1) 各种各样的软件产品(也称为代码), 包括固件,操作系统,移动应用程序和应用程序容器映像, 必须以安全和自动的方式分发和更新, 以防止伪造和篡改; (2) 数字签名代码既可以提供数据完整性以证明未修改代码, 也可以提供源身份验证来标识谁对代码进行签名; (3) 代码签名解决方案应用于代码签名用例(固件签名、驱动签名、应用软件签名)时存在安全问题.

2021 年 4 月, 美国网络安全与基础设施安全局(CISA)和 NIST 联合发布 Defending Against Software Supply Chain Attacks 提供软件供应链风险的概况描述, 并为厂商和用户提供使用 C-SCRM 和 SSDF 两个框架来识别、评估和缓解此类风险的建议[160]. 英国国家网络安全中心(UK National Cyber Security Center, NCSC)在 2018 年 11 月发布《供应链安全指南》(Supply chain security guidance), 提出归属于 4 个方面的 12 条安全细则[161]. 同年 12 月发布《安全开发与部署指南》(Secure development and deployment guidance)[162], 列举 8 项安全原则. 2021 年 2 月, 再次发布针对自动化构建管道恶意攻击的警告[163]. 中国 12 部门于 2020 年 4 月 27 日联合发布《网络安全审查办法》, 同年 6 月 1 日实施. 中国制定了 ICT 供应链安全的相关标准, 如 GB/T 24420-2009 供应链风险管理指南、GB/T 31168-2014 云计算服务安全能力要求、GB/T 32921-2016 信息技术产品供应方行为安全准则、GB/T 22080-2016 信息技术-安全技术-信息安全管理体系要求、GB/T 22239-2019 信息系统安全等级保护基本要求、GB/T 29245-2012 政府部门信息安全管理基本要求、GB/T 36637-2018 信息安全技术 ICT 供应链安全风险管理指南等. 目前, 国际供应链安全标准已渐成体系, 而国内供应链安全要求大多分散在多个标准中, GB/T 36637-2018 作为我国第一个 ICT 供应链安全国家标准, 弥补了 ICT 供应链安全风险管理的空白, 标志着我国供应链安全标准正在起步[164].

5.2 社会层面的行业规范

针对开源软件供应链层出不穷的安全问题, 相关社会组织也提出了相应的措施来减少开源软件供应链的潜在风险. OpenChain 是由 Linux 基金会联合知名企业共同建立的项目, 旨在提供开源软件供应链规范. 目前, OpenChain 2.1,即由 Linux 基金会、Joint Development 基金会和 OpenChain 项目共同制定的 ISO/IEC 5230:2020 标准已经被核准成为国际标准. 该标准能够进一步解决开源软件供应链中的信任问题, 在提升开源合规方面做出了重要的贡献. 众多国际知名企业都加入了由 OpenChain 领导的开源软件供应链社区,例如丰田、谷歌以及思科等企业[165]. 除此之外, MITRE 组织在 2021 年 1 月发布技术报告[166], 提出加强软件供应链完整性的框架[166]. 该框架提出了三项规范软件供应链的措施, 首先, 软件行业必须采用基于软件物料清单(SBOM)的供应链元数据方法, 该方法可以跟踪软件产品中每个组件的组成和出处, 为每个软件组件及其谱系提供元数据完整性, 使用该元数据可以系统地描述和管理风险. 其次, 密码学相关的代码签名和相关的验证基础设施需要足够成熟以反映出现今软件供应链的复杂性和多样性, 并为后量子电子签名预期新标准的快速部署做准备. 第三, 涉及构建和分发软件及软件更新的系统必须满足更高级的标准.

5.3 企业层面的具体实践

除了来自国家层面以及社会组织层面的开源软件供应链规范，部分企业也出台了各自相应的管制措施. 早在 2011 年，微软率先发布了《网络供应链风险管理:实现透明和信任的全球共享》报告[167]，该报告提出了一个旨在有效管理供应链风险的框架. 该框架具有以下四个特点:协作、透明、灵活以及互惠. 该报告认为政府应重新审视开源软件供应链的潜在风险，并和企业一同合作来解决这些问题. 2016 年，华为发布了《全球网络安全挑战——解决供应链风险,正当其时》白皮书[168]. 该白皮书点明了当前供应链的风险，号召政府、企业界以及学术界联合起来应对供应链风险. 华为将供应链安全管理纳入其端到端全球网络安全保障体系，并将实践经验总结到了该白皮书中. 除此之外，该白皮书还总结了多个可以减少供应链风险的措施，例如可以利用 NIST 等框架有效管理供应链风险. 2020 年 10 月，红帽公司提出了可信软件供应链规范，旨在提供软件安全性，合规性，隐私性和透明性四个方面的保障[169]. 该规范要求开发人员在使用静态代码分析和安全扫描工具验证代码之前，代码不允许投入生产环境. 同时，该规范还要求开发人员从受信任的代码仓库中提取可用组件. 2021 年 6 月，谷歌提出了 Supply chain Levels for Software Artifacts (SLSA)[17]，一个用于确保整个软件供应链完整性的端到端框架.SLSA 旨在防御常见的供应链攻击，例如向源代码库提交恶意代码以及诱导开发者使用恶意组件.

6 现实挑战与未来研究方向

根据上述对开源软件供应链安全问题背景的介绍，结合对开源软件供应链安全研究现状的调研，本文总结了现阶段开源软件供应链安全研究所面临的四大挑战和四大未来研究方向及研究内涵，如表 5 所示.

表 5 开源软件供应链的现实挑战和未来研究方向

现实挑战	未来研究方向	研究方向内涵
协作模式下软件依赖复用关系高度复杂化,漏洞检测能力受限	高效智能的组件及应用软件风险识别技术	1. 自动化可扩展的第三方组件风险识别技术 2. 抗混淆深层次的的应用软件风险识别技术
开源趋势下软件供应链体系不断扩大,新模式下的风险识别技术匮乏	新场景及新模式下的风险智能识别技术	1. 新的场景下开源软件供应链的风险识别技术 2. 针对如许可证违规,组件组合配置等新风险的识别技术 3. 开源协作开发模式下持续性的风险识别方案
加固防御能力不足，难以应对海量复杂的开源软件供应链渗入点	开源软件供应链的新型加固防御技术	1. 自动化高效的代码加固技术 2. 复杂语义空间下抗重打包技术 3. 下载更新过程的验证加固
安全可控不足增大了开源软件供应链带“菌”发展的安全风险	结合新兴技术的开源软件供应链管理技术	1. 可靠的开源软件供应链安全设计 2. 开源软件供应链的理解和管理

6.1 现实挑战

基于上述研究及表 5，本文详细分析了当前开源软件供应链安全研究所面临的四大现实挑战.

(1) 协作模式下软件依赖关系高度复杂化，漏洞检测能力受限.

软件的复杂度正随着对软件功能需求的不断提高而不断上升，由于信息技术产业的不断发展以及其它产业对软件依赖的加深，软件的复杂性也不断增加. 软件复杂性主要体现在多个开发人员协同的代码的开发中，该过程会使用大量第三方组件或进行代码复用. 同时漏洞也随着软件复杂的依赖关系进行扩散，对整个开源软件供应链体系的风险检测能力的要求也大大提高. 具体来说，在海量及复杂类型的第三方组件不断涌现的背景下，针对它们的风险识别技术仍存在自动化程度低、可扩展性差等难题；在软件组成高度复杂化，混淆对抗技术流行的背景下，检测软件中依赖的第三方组件及复用的代码仍存在检测精度低，依赖组件识别浅层等难题.

(2) 开源趋势下软件供应链体系不断扩大,新模式下的风险识别技术匮乏.

开源给软件供应链引入了新的环节和步骤, 也增加了软件供应链的攻击面. 开源软件均需要经过复杂的开源软件供应链的供应关系, 涉及到多个开发人员、项目管理者、源码管理工具和分发市场, 依赖大量第三方组件和开发人员的代码贡献. 利用这一庞大体系, 攻击者有更多机会来发起攻击, 包括植入恶意代码、劫持供应渠道. 而复杂的供应关系也使得攻击者的攻击更加难以检测,难以定位威胁渗入点. 具体来说, 新的开源协作模式中, 现有的研究方案仍较少支持对持续性的开发过程或针对性地检测增量改变的代码进行漏洞识别的相关研究, 检测效率较低, 难以识别持续性开发过程中引入的隐蔽漏洞; 此外, 复杂供应关系及对大量第三方组件和开发人员代码贡献的依赖也引入了新的风险, 如知识产权等法律问题以及第三方组件组合的风险等, 当前研究方案针对上述新风险的研究仍较匮乏, 对软件中许可证的自动化提取和语义理解、基于许可证要求对组件及应用软件代码的一致性检测和对海量的第三方组件组合实现高效的风险识别仍是解决上述新风险的重要挑战; 最后, 新的物联网和人工智能场景给开源软件供应链的风险识别带来了新的挑战,现有的研究方案对如语音智能设备中引入新的第三方组件语音技能, 深度学习模型中对第三方神经网络模块的复用等新的场景和研究对象的认识仍属于初步阶段, 识别能力有限.

(3) 加固防御能力不足,难以应对海量复杂的开源软件供应链渗入点.

面向开源软件供应链的攻击事件越来越多, 在发现攻击后, 有必要对软件进行及时的修补和加固. 而当前海量、依赖关系复杂的开源软件给软件的安全加固带来了更大的难度. 首先, 针对海量软件, 难以及时监控和确定软件是否存在漏洞及相应的补丁并及时对软件进行修补; 针对软件的多个不同漏洞, 无法在短时间内全部修复, 需要衡量软件漏洞的严重程度, 评估打补丁的优先级别; 针对海量的软件漏洞, 手动对不同软件的不同漏洞进行修补, 需要大量专家知识并花费大量人力, 增加了开源软件供应链防御的负担;最后, 现有的应用软件代码保护能力弱,仍面临着恶意代码注入、重打包、知识产权侵犯等重要风险, 如何对应用软件的代码进行加固, 在保留原有功能及性能的基础上避免攻击者恶意篡改或盗用仍是亟需解决的重要问题.

(4) 安全可控不足增大了开源软件供应链"带菌"发展的安全风险.

当前开源软件的供应更注重效能, 对网络安全方面的需求主要是结合已有的防御手段进行适配应用, 或对已知的攻击设计防御, 导致当前对开源软件供应链的防御管控处于比较被动的状态. 且当前的开源软件供应链安全仍然主要依靠开发人员、组件分发市场、软件管理者、应用软件分发市场和终端用户的努力, 对每一个环节的代码软件进行审查, 具有较低的效率和可信度, 安全管控力度远远不够, 容易将漏洞、后门或者错误代码驻留在供应链中, 造成"带菌"开发、分发和使用, 带来的安全威胁不言而喻. 针对日益复杂庞大的开源软件供应链如何实现设计可靠的开源软件供应链监管方案仍是当前亟需解决的一个难点.

6.2 未来研究方向

基于上述研究及表 5, 本文进一步总结了四大未来研究方向及相关可行思路, 以应对当前开源软件供应链的现实挑战, 并探讨了在这些研究方向中如何结合新兴的技术.

(1) 高效智能的组件及应用软件风险识别技术

针对开源软件供应链依赖复杂背景下漏洞检测能力受限这一难点, 开源软件供应链中各个环节的风险识别技术相较于传统的漏洞挖掘技术具有更高的要求, 包括支持自动化可扩展的第三方组件风险识别和抗混淆深层次的的应用软件风险识别. 针对自动化可扩展的第三方组件风险识别技术, 潜在的解决方法是结合多种动静态检测方法和智能技术来提高检测能力, 如有机耦合基于元数据、基于静态分析和动态分析的检测方案来识别分发市场上的第三方组件的安全风险. 另一方面, 通过深度学习等技术辅助提取可靠第三方组件的特征来区分恶意冒充的第三方组件来抵抗包名无用攻击等. 针对抗混淆深层次的应用软件风险识别, 潜在的解决方案是结合深层次代码语义表征、基于人工智能的程序语义解混淆等智能化技术来提高应用软件风险识别的抗混淆能力, 结合复杂图数据构建与分析等技术来提取目标程序中的复杂依赖关系并进行进一步的分析.

(2) 新场景及新模式下的风险智能识别技术

新的开发和应用场景随着现代化的进展也在不断涌现, 开源软件供应链各个环节的风险识别技术需基于

新的场景和模式进行相应的补充和优化.

针对新涌现的物联网及人工智能场景，当前研究对其分发市场的研究仍处于初级探索阶段，相应的第三方组件及应用软件的风险识别研究仍十分匮乏，如物联网设备固件、人工智能平台和模型中也同样利用了大量第三方组件，且存在安全风险. 新的场景在攻击面上也会有多样的变化，带来了新的挑战，如物联网设备固件架构多样，深度学习模型中的网络模块复用等问题，需基于新场景下第三方组件和应用软件的特点设计相应的检测方案. 此外，由于考虑到大量代码依赖和复用带来的许可证违规和组件组合配置风险等，需要加强对上述新风险的检测方案的研究，如结合自然语言处理技术增强许可证的语义理解、结合 CodeQL 等对代码、执行流程的搜索技术来实现对代码的许可证违规查询、基于深度学习技术实现对第三方组件关联组合关系挖掘和风险识别等. 最后，由于协作开发新模式的引入，需研究持续性风险识别技术来对不同研究人员的代码提交进行自动化持续性的检测，如持续性模糊测试，对修改或添加的代码进行定向模糊测试等.

(3) 开源软件的新型加固防御

针对开源软件的新型加固防御技术主要考虑以下两个研究方向：如何实现组件及应用软件开发环节中高效自动化的代码加固技术及如何实现应用软件分发环节中复杂语义空间下的抗重打包技术.

针对组件及应用软件开发环节，考虑到开源软件供应链的依赖关系网络，一个组件的漏洞可以很快利用该网络快速传播，由点及面，导致大量下游组件软件面临漏洞威胁，直接影响到大规模的用户. 漏洞在开源软件供应链上能以极快的传播速度传播到广大的软件项目中. 针对这一现象，亟需在组件和应用软件开发环节实现对目标组件及软件的补丁存在性检测、漏洞评估及自动化补丁加固等技术，且要求能够通过有效的渠道快速准确地部署到相应的软件系统上.

此外，针对应用软件分发环节中严重的重打包攻击，需研究新型安全防御技术如代码水印、代码混淆技术来实现应用软件的知识产权保护，避免攻击者篡改应用软件.

(4) 结合新兴技术的开源软件管理方案

开源软件供应链庞大复杂，具有多个环节，开发实现的软件系统具有大量组件依赖关系. 各个分发市场平台和用户的检测能力有限，也难以应对风险. 需要软件供应商实现对整个开源供应链的供应关系的管理和检测，实时提供预警信息. 目前这方面的研究仍处于初步阶段. 针对可靠的安全管理方法，一是需要研究如何加强顶层设计，通过对开源软件链供应结构的改进，如添加签名和完整性验证等技术，将供应链中供应过程记录在区块链上保证供应链的完整性和不可否认性等，从本质上提高开源软件供应链的安全性，二是提高对开源软件供应链的理解和管理力度，通过设计对完整供应链的管理手段和工具，结合新兴的知识图谱、人工智能等技术探索端到端的监管方案，设计相关流程和工具，如利用知识图谱对开源软件供应链中第三方组件、应用软件、开发人员等知识进行建模，并结合深度学习等技术，帮助实现开源软件供应链的知识分析、管理和评估，对风险检测和加固防御等下游任务具有重要意义.

7 结束语

随着开源协作新模式下软件供应链的发展，软件在开发、分发和使用等关键环节中出现了新的威胁渗入点，如包名误用、恶意代码提交等，新的场景如物联网、人工智能也对开源软件供应链带来了独特的风险，如语音抢注攻击、组件组合配置风险等. 针对上述问题，一大批来自学术界和工业界的学者对开源软件供应链的安全问题进行了广泛关注和深入研究，并且在开源软件供应链的风险识别、加固和管理等方向上取得了许多瞩目的研究成果. 然而，到目前为止，开源软件供应链安全的研究仍处于初级阶段，在新兴的时代背景下仍然面临着许多关键的科学问题和挑战，包括如何设计高效智能的组件及应用软件风险识别技术来应对高效协作模式下软件依赖复用关系高度复杂化、漏洞检测能力受限的挑战，如何研究新场景及新模式下的风险智能识别技术作为不断扩大的开源软件供应链体系下的风险识别技术补充，如何设计新型可靠的安全加固方案以应对海量复杂的开源软件供应链渗入点，以及如何结合新兴技术实现对开源软件供应链的安全可控来降低其带”菌”发展的安全风险.

基于以上内容, 本文从开源软件供应链的理论模型、风险分析、风险识别及加固防御四个层面系统地研究了当前开源软件供应链安全问题, 回顾了大量具有影响力的研究成果并进行了科学分类、总结和分析. 同时, 本文指出了当前开源软件供应链安全研究面临的现实挑战, 探讨了未来研究方向, 旨在为推动开源软件供应链安全研究的进一步发展和应用提供指导和参考.

References:

[1] Veracode. State of Software Security: Open Source Edition. https://info.veracode.com/report-state-of-software-security-open-source-edition.html.

[2] Wu Z, Zhang C, Sun H, et al. Application Research of Program Reverse Analysis in Pollution Detection of Software Supply Chain: A Survey. Journal of Computer Applications, 2020, 40(01): 103–115.

[3] Zhenfei Z. Research on Pollution Mechanism and Defense of Software Supply Chain. Beijing University of Posts and Telecommunications. 2018.

[4] He X, zhang Y, Liu Q. Software Supply Chain Security: A Survey. Journal of Cyber Security, 2020, 5(01): 57–73.

[5] Hassija V, Chamola V, Gupta V, et al. A Survey on Supply Chain Security: Application Areas, Security Threats, and Solution Architectures. IEEE Internet of Things Journal, 2021, 8(8): 6222–6246.

[6] Du S, Lu T, Zhao L, et al. Towards An Analysis of Software Supply Chain Risk Management. 2013: 6.

[7] GitHub. Build Software Better, Together. https://github.com.

[8] Gitee. Software Development and Collaboration Platform. http://gitee.com/.

[9] PyPI. The Python Package Index. https://pypi.org/.

[10] NPM. Npm. https://www.npmjs.com/.

[11] Maven. Maven – Welcome to Apache Maven. https://maven.apache.org/.

[12] OpenWrt Wiki. Opkg Package Manager. https://openwrt.org/docs/guide-user/additional-software/opkg.

[13] RubyGems. Your Community Gem Host. https://rubygems.org/.

[14] Amazon. Alexa Skills and Features. https://www.amazon.com/alexa-skills/b/?ie=UTF8&node=13727921011&ref_=topnav_storetab_a2s.

[15] IFTTT. My Applets - IFTTT. https://ifttt.com.

[16] OpenSSF. Identifying Security Threats in Open Source Projects. https://github.com/ossf/wg-identifying-security-threats.

[17] Google. Introducing SLSA, an End-to-End Framework for Supply Chain Integrity. https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html.

[18] CISA. Supply Chain Compromise. https://www.cisa.gov/supply-chain-compromise.

[19] Torres-Arias S, Afzali H, Kuppusamy T K, et al. In-Toto: Providing Farm-to-Table Guarantees for Bits and Bytes.

[20] WhiteSource. Software Supply Chain Attacks. https://www.whitesourcesoftware.com/resources/blog/software-supply-chain-attacks/.

[21] Webmin. Webmin 1.890 Exploit - What Happened?. https://www.webmin.com/exploit.html.

[22] CrowdStrike Intelligence Team. SUNSPOT Malware: A Technical Analysis. https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/.

[23] SECURELIST. ShadowPad in Corporate Networks. https://securelist.com/shadowpad-in-corporate-networks/81432/.

[24] Wikipedia. XcodeGhost. https://en.wikipedia.org/w/index.php?title=XcodeGhost&oldid=1022461786.

[25] The Good Hacker. PHP GIT Server Hacked and Backdoor Injected in PHP Source Code. https://thegoodhacker.com/posts/php-git-server-hacked-and-backdoor-inserted-in-php-source-code/.

[26] Pelayo D. I don't know what to say. https://github.com/dominictarr/event-stream/issues/116.

[27] Cimpanu C. Hacker Backdoors Popular JavaScript Library to Steal Bitcoin Funds. https://www.zdnet.com/article/hacker-backdoors-popular-javascript-library-to-steal-bitcoin-funds/.

[28] Cimpanu C. Malware Found in Arch Linux AUR Package Repository. https://www.bleepingcomputer.com/news/security/malware-found-in-arch-linux-aur-package-repository/.

[29] Gonzalez D, Zimmermann T, Godefroid P, et al. Anomalicious: Automated Detection of Anomalous and Potentially Malicious

Commits on GitHub. International Conference on Software Engineering: Software Engineering in Practice, 2021.

[30] Nutt C. Cloud Source Host Code Spaces Hacked, Developers Lose Code. https://www.gamasutra.com/view/news/219462/Cloud_source_host_Code_Spaces_hacked_developers_lose_code.php.

[31] Codecov. Post-Mortem/Root Cause Analysis. https://about.codecov.io/apr-2021-post-mortem/.

[32] Vu D-L, Pashchenko I, Massacci F, et al. Typosquatting and Combosquatting Attacks on the Python Ecosystem. IEEE European Symposium on Security and Privacy Workshops, 2020.

[33] Catalin Cimpanu. 17 Backdoored Malicious Images Removed From Docker Hub, But Are You Really Any Safer?. https://www.bleepingcomputer.com/news/security/17-backdoored-docker-images-removed-from-docker-hub/.

[34] Birsan A. Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610.

[35] Heartbleed. The Heartbleed Bug. https://heartbleed.com/.

[36] Checkpoint. Extracting a 19 Year Old Code Execution from WinRAR. https://research.checkpoint.com/2019/extracting-code-execution-from-winrar/.

[37] Fischer F, Böttinger K, Xiao H, et al. Stack Overflow Considered Harmful? The Impact of Copy and Paste on Android Application Security. IEEE Symposium on Security and Privacy, 2017 .

[38] Computerworld. Cisco Sued for Copyright Infringement over Linksys Router. https://www.computerworld.com/article/2529871/cisco-sued-for-copyright-infringement-over-linksys-router.html.

[39] Anandashankar Mazumdar. Oracle Victory Stirs Uncertainties in Software Copyright. https://news.bloombergtax.com/ip-law/oracle-victory-stirs-uncertainties-in-software-copyright.

[40] Jaime Cochran. The WireX Botnet: How Industry Collaboration Disrupted a DDoS Attack. https://blog.cloudflare.com/the-wirex-botnet/.

[41] Wikipedia. Petya (Malware). https://en.wikipedia.org/w/index.php?title=Petya_(malware)&oldid=1030666409.

[42] TitanWolf. Pandora's Box Opened: Large-Scale Software Upgrade Hijacking Attacks Broke out in Many Provinces across the Country. https://titanwolf.org/Network/Articles/Article?AID=88595e24-dfc3-48d3-8d22-247fbdd63b89#gsc.tab=0.

[43] Computerworld. New Malware Overwrites Software Updaters. https://www.computerworld.com/article/2755831/new-malware-overwrites-software-updaters.html.

[44] Xiao F, Huang J, Xiong Y, et al. Abusing Hidden Properties to Attack the Node.Js Ecosystem. 2021.

[45] Wikipedia. Dirty COW. https://en.wikipedia.org/wiki/Dirty_COW.

[46] Duan R, Alrawi O, Kasturi R P, et al. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. Network and Distributed System Security Symposium, 2021.

[47] Taylor M, Vaidya R, Davidson D, et al. Defending Against Package Typosquatting. International Conference on Network and System Security, 2020: 112–131.

[48] Bullock M. Pypi-Parker[M/OL]. . https://github.com/mattsb42/pypi-parker.

[49] Vu D L, Pashchenko I, Massacci F, et al. Towards Using Source Code Repositories to Identify Software Supply Chain Attacks. ACM SIGSAC Conference on Computer and Communications Security, 2020: 2093–2095.

[50] Davis J C, Williamson E R, Lee D. A Sense of Time for JavaScript and Node.Js: First-Class Timeouts as a Cure for Event Handler Poisoning. USENIX Security Symposium, 2018.

[51] Staicu C-A, Pradel M, Livshits B. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. Network and Distributed System Security Symposium, 2018.

[52] Alexa Developer Official Site. Amazon Alexa Voice AI. https://developer.amazon.com/en-US/alexa.

[53] Google. Google Assistant, Your Own Personal Google. https://assistant.google.com/.

[54] Zhang N, Mi X, Feng X, et al. Dangerous Skills: Understanding and Mitigating Security Risks of Voice-Controlled Third-Party Functions on Virtual Personal Assistant Systems. IEEE Symposium on Security and Privacy, 2019 .

[55] Guo Z, Lin Z, Li P, et al. SkillExplorer: Understanding the Behavior of Skills in Large Scale, USENIX Security Symposium. 2020: 2649–2666.

[56] Zhang Y, Xu L, Mendoza A, et al. Life after Speech Recognition: Fuzzing Semantic Misinterpretation for Voice Assistant

Applications. Network and Distributed System Security Symposium, 2019.

[57] Bastys I, Balliu M, Sabelfeld A. If This Then What? Controlling Flows in IoT Apps. ACM SIGSAC Conference on Computer and Communications Security, 2018: 1102–1119.

[58] Ding W, Hu H. On the Safety of IoT Device Physical Interaction Control. ACM SIGSAC Conference on Computer and Communications Security, 2018: 832–846.

[59] Wang Q, Datta P, Yang W, et al. Charting the Attack Surface of Trigger-Action IoT Platforms. ACM SIGSAC Conference on Computer and Communications Security, 2019: 1439–1453.

[60] Alhanahnah M, Stevens C, Bagheri H. Scalable Analysis of Interaction Threats in IoT Systems. ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020: 272–285.

[61] Staicu C-A, Pradel M. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. USENIX Security Symposium, 2018.

[62] Kula R G, Ouni A, German D M, et al. On the Impact of Micro-Packages: An Empirical Study of the Npm JavaScript Ecosystem. arXiv preprint arXiv:1709.04638, 2017.

[63] Dey T, Mockus A. Are Software Dependency Supply Chain Metrics Useful in Predicting Change of Popularity of NPM Packages?. International Conference on Predictive Models and Data Analytics in Software Engineering, 2018: 66–69.

[64] Zerouali A, Mens T, Gonzalez-Barahona J, et al. A Formal Framework for Measuring Technical Lag in Component Repositories — and Its Application to Npm. Journal of Software: Evolution and Process, 2019, 31(8): e2157.

[65] Zimmermann M, Staicu C-A, Pradel M. Small World with High Risks: A Study of Security Threats in the Npm Ecosystem. USENIX Security Symposium, 2019: 17.

[66] Decan A, Mens T, Constantinou E. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. International Conference on Mining Software Repositories. Association for Computing Machinery, 2018: 181–191.

[67] Ruohonen J. An Empirical Analysis of Vulnerabilities in Python Packages for Web Applications. International Workshop on Empirical Software Engineering in Practice, 2018.

[68] Cheng L, Wilson C, Liao S, et al. Dangerous Skills Got Certified: Measuring the Trustworthiness of Skill Certification in Voice Personal Assistant Platforms. ACM SIGSAC Conference on Computer and Communications Security. 2020: 1699–1716.

[69] Alhadlaq A, Tang J, Almaymoni M. Privacy in the Amazon Alexa Skills Ecosystem. Workshop on Hot Topics in Privacy Enhancing Technologies, 2017: 2.

[70] Lentzsch C, Shah S J, Andow B, et al. Hey Alexa, Is This Skill Safe?: Taking a Closer Look at the Alexa Skill Ecosystem. Network and Distributed System Security Symposium, 2021.

[71] Li Y, Ji S, Chen Y, et al. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. USENIX Security Symposium, 2021 .

[72] Lyu C, Ji S, Zhang C, et al. MOPT: Optimized Mutation Scheduling for Fuzzers. USENIX Security Symposium, 2019: 1949–1966.

[73] Wang Q, Ji S, Tian Y, et al. MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols. USENIX Security Symposium, 2021: 4205–4222.

[74] Wang L, Feng L, Lian L, et al. Principle and Practice of Taint Analysis. Journal of Software, 2017, 28(4): 860–882.

[75] Bleser J D, Stiévenart Q, Nicolay J, et al. Static Taint Analysis of Event-Driven Scheme Programs. European Lisp Symposium. ACM, 2017: 80–87.

[76] Karim R, Tip F, Sochůrková A, et al. Platform-Independent Dynamic Taint Analysis for JavaScript. IEEE Transactions on Software Engineering, 2020, 46(12): 1364–1379.

[77] Kreindl J, Bonetta D, Mössenböck H. Towards Efficient, Multi-Language Dynamic Taint Analysis. ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, 2019: 85–94.

[78] Staicu C-A, Torp M T, Schäfer M, et al. Extracting Taint Specifications for JavaScript Libraries[C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. New York, NY, USA: Association for Computing Machinery, 2020: 198–209.

[79] Manes V J M, Han H, Han C, et al. The Art, Science, and Engineering of Fuzzing: A Survey. IEEE Transactions on Software Engineering, 2019(01): 1–1.

[80] Ren Z, Zheng H, Zhang J, et al. A Review of Fuzzing Techniques. Journal of Computer Research and Development, 2021, 58(5): 944.

[81] Lee S, Yoon C, Lee C, et al. DELTA: A Security Assessment Framework for Software-Defined Networks. Network and Distributed System Security Symposium, 2017.

[82] Han H, Cha S K. IMF: Inferred Model-Based Fuzzer. ACM SIGSAC Conference on Computer and Communications Security, 2017: 2345–2358.

[83] Chen J, Diao W, Zhao Q, et al. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-Based Fuzzing. Network and Distributed System Security Symposium, 2018.

[84] Yun I, Lee S, Xu M, et al. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. USENIX Security Symposium, 2018: 745–761.

[85] Li Y, Ji S, Lyu C, et al. V-Fuzz: Vulnerability Prediction-Assisted Evolutionary Fuzzing for Binary Programs. IEEE Transactions on Cybernetics, 2020: 1–12.

[86] Zheng Y, Davanian A, Yin H, et al. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. USENIX Security Symposium, 2019: 1099–1114.

[87] Google. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[88] Tellnes J. Dependencies: No Software Is an Island. Master. Master's thesis, 2013.

[89] OWASP. OWASP Dependency-Check Project. https://owasp.org/www-project-dependency-check/.

[90] Cadariu M, Bouwers E, Visser J, et al. Tracking Known Security Vulnerabilities in Proprietary Software Systems. International Conference on Software Analysis, Evolution, and Reengineering, 2015.

[91] Backes M, Bugiel S, Derr E. Reliable Third-Party Library Detection in Android and Its Security Applications. ACM SIGSAC Conference on Computer and Communications Security, 2016: 356–367.

[92] Zhan X, Fan L, Chen S, et al. ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. International Conference on Software Engineering, 2021 .

[93] Woo S, Park S, Kim S, et al. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. International Conference on Software Engineering, 2021.

[94] Li M, Wang W, Wang P, et al. LibD: Scalable and Precise Third-Party Library Detection in Android Markets. International Conference on Software Engineering, 2017.

[95] Ponta S E, Plate H, Sabetta A. Detection, Assessment and Mitigation of Vulnerabilities in Open Source Dependencies. Empirical Software Engineering, 2020, 25(5): 3175–3215.

[96] Duan R, Bijlani A, Xu M, et al. Identifying Open-Source License Violation and 1-Day Security Risk at Large Scale. ACM SIGSAC Conference on Computer and Communications Security, 2017: 2169–2185.

[97] Ohm M, Sykosch A, Meier M. Towards Detection of Software Supply Chain Attacks by Forensic Artifacts, International Conference on Availability, Reliability and Security, 2020: 1–6.

[98] Akram J, Luo P. SQVDT: A Scalable Quantitative Vulnerability Detection Technique for Source Code Security Assessment. Software: Practice and Experience, 2021, 51(2): 294–318.

[99] Stack Overflow. Copying Code from Stack Overflow? You Might Paste Security Vulnerabilities, Tool. https://stackoverflow.blog/2019/11/26/copying-code-from-stack-overflow-you-might-be-spreading-security-vulnerabilities/.

[100] Bilgin Z, Ersoy M A, Soykan E U, et al. Vulnerability Prediction From Source Code Using Machine Learning. IEEE Access, 2020, 8: 150672–150684.

[101] Xu X, Liu C, Feng Q, et al. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. ACM SIGSAC Conference on Computer and Communications Security, 2017: 363–376.

[102] Ding S H H, Fung B C M, Charland P. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. IEEE Symposium on Security and Privacy, 2019.

[103] Li X, Qu Y, Yin H. PalmTree: Learning an Assembly Language Model for Instruction Embedding. ACM SIGSAC Conference on Computer and Communications Security, 2021 .

[104] Hemel A, Kalleberg K T, Vermaas R, et al. Finding Software License Violations through Binary Code Clone Detection. Working

Conference on Mining Software Repositories, 2011: 63–72.

[105] Zhou W, Zhou Y, Jiang X, et al. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. ACM Conference on Data and Application Security and Privacy, 2012: 317–326.

[106] Golubev Y, Eliseeva M, Povarov N, et al. A Study of Potential Code Borrowing and License Violations in Java Projects on GitHub. International Conference on Mining Software Repositories, 2020: 54–64.

[107] Sajnani H, Saini V, Svajlenko J, et al. SourcererCC: Scaling Code Clone Detection to Big-Code. International Conference on Software Engineering, 2016: 1157-1168.

[108] Crussell J, Gibler C, Chen H. AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. IEEE Transactions on Mobile Computing, 2015, 14(10): 2007–2019.

[109] Gonzalez H, Stakhanova N, Ghorbani A A. DroidKin: Lightweight Detection of Android Apps Similarity. International Conference on Security and Privacy in Communication Networks, 2015: 436–453.

[110] Wan L. Automated Vulnerability Detection System Based on Commit Messages. Doctoral dissertation, Nanyang Technological University, 2019.

[111] Andrade R. Privacy and Security Constraints for Code Contributions. ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, 2015: 27–29.

[112] Wu Q, Lu K. On the Feasibility of Stealthily Introducing Vulnerabilities in Open-Source Software via Hypocrite Commit. 2021.

[113] Google. OSS-Fuzz. https://google.github.io/oss-fuzz/.

[114] Google. Overview of ClusterFuzzLite. https://google.github.io/clusterfuzzlite/overview/.

[115] Sinha V S, Saha D, Dhoolia P, et al. Detecting and Mitigating Secret-Key Leaks in Source Code Repositories. Working Conference on Mining Software Repositories, 2015: 396-400.

[116] Meli M, McNiece M R, Reaves B. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. Network and Distributed System Security Symposium, 2019.

[117] Garrett K, Ferreira G, Jia L, et al. Detecting Suspicious Package Updates. International Conference on Software Engineering: New Ideas and Emerging Results, 2019.

[118] Teng J, Guang Y, Shu H, et al. Automatic Detection Method for Software Upgrade Vulnerabilities based on Traffic Analysis. Chinese Journal of Network and Information Security, 2020, 6(01): 94–108.

[119] Mike Gerwitz. A Git Horror Story: Repository Integrity With Signed Commits. https://mikegerwitz.com/2012/05/a-git-horror-story-repository-integrity-with-signed-commits.

[120] Tedhudek. Trusted Publishers Certificate Store. https://docs.microsoft.com/en-us/windows-hardware/drivers/install/trusted-publishers-certificate-store.

[121] Neelakantam S, Pant T. Learning Web-Based Virtual Reality: Build and Deploy Web-Based Virtual Reality Technology. Apress, 2017: 69–79.

[122] Lamb C, Zacchiroli S. Reproducible Builds: Increasing the Integrity of Software Supply Chains. IEEE Software, 2021: 0–0.

[123] Ullah F, Raft A, Shahin M, et al. Security Support in Continuous Deployment Pipeline. arXiv preprint arXiv:1703.04277.

[124] Bass L, Holz R, Rimba P, et al. Securing a Deployment Pipeline. International Workshop on Release Engineering, 2015 .

[125] Linkedin. External Library Management. https://engineering.linkedin.com/blog/2017/08/external-library-management--making-continuous-delivery-reliable.

[126] Koishybayev I, Kapravelos A. Mininode: Reducing the Attack Surface of Node.Js Applications. International Symposium on Research in Attacks, Intrusions and Defenses, 2020.

[127] Nguyen D C, Derr E, Backes M, et al. Up2Dep: Android Tool Support to Fix Insecure Code Dependencies. Annual Computer Security Applications Conference, 2020: 263–276.

[128] Vasilakis N, Karel B, Roessler N, et al. BreakApp: Automated, Flexible Application Compartmentalization. Network and Distributed System Security Symposium, 2018.

[129] Vasilakis N, Benetopoulos A, Handa S, et al. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. ACM SIGSAC Conference on Computer and Communications Security, 2021: 1755–1770.

[130] CVSS. Common Vulnerability Scoring System SIG. https://www.first.org/cvss.

[131] Younis A A, Malaiya Y K, Ray I. Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability. International Symposium on High-Assurance Systems Engineering, 2014.

[132] Plate H, Ponta S E, Sabetta A. Impact Assessment for Vulnerabilities in Open-Source Software Libraries. International Conference on Software Maintenance and Evolution, 2015.

[133] Wu Q, He Y, McCamant S, et al. Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison. Network and Distributed System Security Symposium, 2020.

[134] Zhang H, Qian Z. Precise and Accurate Patch Presence Test for Binaries. USENIX Security Symposium, 2018: 887–902.

[135] Jiang Z, Zhang Y, Xu J, et al. PDiff: Semantic-Based Patch Presence Testing for Downstream Kernels. ACM SIGSAC Conference on Computer and Communications Security, 2020: 1149–1163.

[136] Dai J, Zhang Y, Jiang Z, et al. BScout: Direct Whole Patch Presence Test for Java Executables. USENIX Security Symposium, 2020: 1147–1164.

[137] Chen Y, Zhang Y, Wang Z, et al. Adaptive Android Kernel Live Patching. USENIX Security Symposium, 2017: 1253–1270.

[138] Duan R, Bijlani A, Ji Y, et al. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries. Network and Distributed System Security Symposium, 2019.

[139] Wang X, Sun K, Batcheller A, et al. An Empirical Study of Secret Security Patch in Open Source Software. Adaptive Autonomous Secure Cyber Systems, 2020: 269-289.

[140] Wang X, Sun K, Batcheller A, et al. Detecting “0-Day” Vulnerability: An Empirical Study of Secret Security Patch in OSS. Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2019.

[141] Apple Developer. Distribute. https://developer.apple.com/distribute/.

[142] Npm Docs. Npm-Audit. https://docs.npmjs.com/cli/v7/commands/npm-audit.

[143] Android. Download Android Studio and SDK Tools. https://developer.android.com/studio.

[144] ProGuard. Java Obfuscator and Android App Optimizer. https://www.guardsquare.com/proguard.

[145] Song L, Tang Z, Li Z, et al. AppIS: Protect Android Apps Against Runtime Repackaging Attacks. International Conference on Parallel and Distributed Systems, 2017: 25–32.

[146] Gregor F, Ozga W, Vaucher S, et al. Trust Management as a Service: Enabling Trusted Execution in the Face of Byzantine Stakeholders. International Conference on Dependable Systems and Networks, 2019.

[147] Ozga W, Quoc D L, Fetzer C. A Practical Approach for Updating an Integrity-Enforced Operating System. International Middleware Conference, 2020: 311–325.

[148] Karthik T, Kuppusamy, McCoy D. Uptane : Securing Software Updates for Automobiles. International Conference on Embedded Security in Car, 2016: 1-11.

[149] Singi K, R P J C B, Podder S, et al. Trusted Software Supply Chain. International Conference on Automated Software Engineering, 2019.

[150] Samuel J, Mathewson N, Cappos J, et al. Survivable Key Compromise in Software Update Systems. ACM Conference on Computer and Communications Security, 2010: 61–72.

[151] Kuppusamy T K, Torres-Arias S, Diaz V, et al. Diplomat: Using Delegations to Protect Community Repositories. USENIX Symposium on Networked Systems Design and Implementation, 2016: 16.

[152] Kuppusamy T K, Diaz V, Cappos J. Mercury: Bandwidth-Effective Prevention of Rollback Attacks Against Community Repositories. USENIX Annual Technical Conference, 2017: 17.

[153] Brown F, Mirian A, Jaiswal A, et al. SPAM: A Secure Package Manager. USENIX HotSec 2017, 2017: 7.

[154] Cappos J, Samuel J, Baker S, et al. Package Management Security. University of Arizona Technical Report, 2018:08-02.

[155] Nikitin K, Kokoris-Kogias E, Jovanovic P, et al. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. USENIX Security Symposium, 2017.

[156] Stengele O, Baumeister A, Birnstill P, et al. Access Control for Binary Integrity Protection Using Ethereum. ACM Symposium on Access Control Models and Technologies, 2019: 3–12.

[157] Guarnizo J, Alangot B, Szalachowski P. SmartWitness: A Proactive Software Transparency System Using Smart Contracts. ACM International Symposium on Blockchain and Secure Critical Infrastructure, 2020: 117–129.

[158] Boyens J M, Paulsen C, Moorthy R, et al. Supply Chain Risk Management Practices for Federal Information Systems and Organizations. NIST Special Publication, 2015: 32.
[159] Cooper D, Regenscheid A, Souppaya M, et al. Security Considerations for Code Signing. NIST Cybersecurity White Paper, 2018.
[160] CISA. Defending Against Software Supply Chain Attacks. https://www.cisa.gov/publication/software-supply-chain-attacks.
[161] UK National Cyber Security Centre. Supply Chain Security Guidance. https://www.ncsc.gov.uk/collection/supply-chain-security.
[162] UK National Cyber Security Centre. Secure Development and Deployment Guidance. https://www.ncsc.gov.uk/collection/developers-collection.
[163] UK National Cyber Security Centre. Defending Software Build Pipelines from Malicious Attack. https://www.ncsc.gov.uk/blog-post/defending-software-build-pipelines-from-malicious-attack.
[164] National Security Science and Technology Evaluation Center. GB/T 36637-2018 Standard Interpretation. http://www.gjbmj.gov.cn/n1/2020/0115/c411145-31550085.html.
[165] Coughlan S. OpenChain 2.1 Is ISO/IEC 5230:2020, the International Standard for Open Source Compliance. https://www.openchainproject.org/featured/2020/12/15/openchain-2-1-is-iso5230.
[166] Clancy D C, Ferraro J, Martin R A, et al. Deliver Uncompromised: Securing Critical Software Supply Chains.
[167] Microsoft Cybersecurity. Cyber Supply Chain Risk Management. https://www.microsoft.com/en-us/cybersecurity/content-hub/cyber-supply-chain-risk-management.
[168] Huawei. Huawei Released 2016 Cyber Security White Paper. 2016. https://www.huawei.com/cn/news/2016/6/2016-Cyber-Security-White-Paper.
[169] Red Hat. How to use a trusted software supply chain to adopt DevSecOps. https://www.redhat.com/en/resources/ve-trusted-software-supply-chain-brief

附中文参考文献:

[2] 武振华,张超,孙贺,等.程序逆向分析在软件供应链污染检测中的应用研究综述. 计算机应用, 2020, 040(001):103-115.
[3] 周振飞.软件供应链污染机理与防御研究.北京邮电大学, 2018 (in Chinese with English abstract).
[4] 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.
[74] 王蕾,李丰,李炼,冯晓兵.污点分析技术的原理和实践应用.软件学报, 2017, 28(04):860-882.
[80] 任泽众,郑晗,张嘉元,王文杰,冯涛,王鹤,张玉清.模糊测试技术综述.计算机研究与发展, 2021, 58(05):944-963.
[118]腾金辉,光焱,舒辉等.基于流量分析的软件升级漏洞自动检测方法.网络与信息安全学报, 2020, 6(01): 94–108.
[164]国家保密科技测评中心.GB/T 36637-2018《信息安全技术 ICT 供应链安全风险管理指南》标准解读.2020.http://www.gjbmj.gov.cn/n1/2020/0115/c411145-31550085.html
[168]华为.华为发布 2016 年网络安全白皮书.2016.https://www.huawei.com/cn/news/2016/6/2016-Cyber-Security-White-Paper

纪守领(1986－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为人工智能与安全,数据驱动安全,软件与系统安全,大数据分析与多媒体理解.

王琴应 (1996－),女,博士生,主要研究领域为物联网安全, 软件与系统安全.

陈安莹(1996－),女,硕士生,主要研究领域为数据驱动安全,软件与系统安全.

赵彬彬(1996－),男,博士生,主要研究领域为物联网安全.

叶童(1998－),男,博士生,主要研究领域为机器学习,漏洞检测.

吴敬征(1982－),男,博士,研究员,博士生导师,CCF 专业会员,主要研究领域为系统安全,漏洞挖掘,操作系统安全.

尹建伟(1974－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为医疗信息化,分布式计算,服务计算,云计算.

张旭鸿(1988－),男,博士,研究员.博士生导师,主要研究领域为分布式大数据系统,人工智能与安全,数据驱动安全,软件与系统安全,计算机视觉,大数据挖掘与分析,多媒体理解.

李昀(1974－),女,博士,首席技术专家,主要研究领域为人工智能与安全,认知智能,大数据安全,知识工程.

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