ENTITY 软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

开源软件供应链研究综述——一个整体性的视角*

高恺[1], 何昊[2], 谢冰[1,2], 周明辉[2]

[1](北京大学 软件与微电子学院,北京 100871)

[2](北京大学 计算机学院,北京 100871)

通讯作者: 周明辉, E-mail: zhmh@pku.edu.cn

摘 要: 开源软件已经成为现代社会的一项关键基础设施，支撑着几乎所有领域的软件开发。通过安装依赖、API 调用、项目 fork、文件拷贝和代码克隆等形式的代码复用，开源软件之间形成了错综复杂的供应（依赖）关系网络，被称为开源软件供应链。一方面，开源软件供应链为软件开发提供了便利，已然成为软件行业的基石。另一方面，上游软件的风险可以沿着开源软件供应链波及众多的下游软件，使开源软件供应链呈现牵一发而动全身的特点。开源软件供应链近年来逐渐成为学术界和工业界的关注焦点。为了帮助增进研究人员对开源软件供应链的认识，本文从整体性的角度，对开源软件供应链给出定义和研究框架；然后，对国内外的研究工作进行系统文献调研，总结了结构与演化、风险传播与管理、依赖管理三个方面的研究现状；最后，展望了开源软件供应链的研究挑战和未来研究方向。

关键词: 开源软件供应链;结构与演化;风险传播与管理;依赖管理

中图法分类号: TP311

中文引用格式:

英文引用格式:

Survey of Open Source Software Supply Chain – A Holistic Perspective

GAO Kai[1], HE Hao[2], XIE Bing[1,2], ZHOU Ming-Hui[2]

[1](School of Software and Microelectronics, Peking University, Beijing 100871, China)

[2](School of Computer Science, Peking University, Beijing 100871, China)

Abstract: Open source software has been a key infrastructure of modern society, supporting software development in almost every area. Through various kinds of code reuse such as install dependency, API call, project fork, file copy, and code clone, open source software forms an intricate supply (i.e., dependency) network, which is referred as open source software supply chain. On the one hand, software supply chains facilitate software development and have become the foundation of software industry. On the other hand, risks from upstream software can affect numerous downstream software depending on it along the chain, leading to the ripple effect in open source software chains. Open source software supply chains have attracted more and more attention from both the academia and the industry. To help advance researchers' knowledge on open source software supply chain, this paper provides a definition and research framework of open source software supply chain from a holistic perspective. Then, it conducts systematic literature review on worldwide research and summarizes the status quo of research from three aspects: structure and evolution, risk propagation and management, and dependency management. Finally, the paper summarizes challenges and opportunities of future research on open source software supply chain.

Key words: open source software supply chain; structure and evolution; risk propagation and management; dependency management

20 世纪末以来，开源软件取得了令人瞩目的成就。一方面，随着以 GitHub 为代表的代码托管平台的流行，开发者普遍在这些平台上进行软件开发，互联网上积累了海量的代码[1]。截至 2021 年 10 月 1 日，GitHub

* 基金项目:

收稿时间:; 修改时间:; 采用时间:

上托管了超过 2.5 亿个代码仓库[2]；截至 2022 年 9 月 26 日，编程问答社区 Stack Overflow 上发布了约 2300 万个问题和 3400 万个回答[3]。另一方面，开源软件几乎涵盖软件开发的所有环节，并在市场占有率上逐渐媲美甚至远超同类商业软件。从 Linux 操作系统 Red Hat Enterprise Linux (RHEL) 和 Android，到深度学习框架 TensorFlow 和 PyTorch，人们日常使用的软件中随处可见开源软件的影子。2021 年 Synopsys 分析了 17 个行业的 2400 个商业代码库，发现 97%的代码库中包含开源代码[4]。由此可见，开源软件已经成为现代社会的一项重要基础设施，支撑着几乎每一个领域的软件开发。

（开源）软件的开发建立在对已有软件或代码的复用的基础上。通过安装依赖、API 调用、项目 fork、文件拷贝和代码克隆等各种形式的代码复用，开源软件之间形成了供应（即依赖）关系。随着技术的发展和应用的深入，开源软件之间的供应关系网络也愈发复杂[5]。在这样的网络中，一个被其他开源软件复用的开源软件，同时还会复用其他开源软件——即一个开源软件可能既是上游供应者又是下游消费者，从而开源软件之间的供应关系网络呈现出层级的结构，逐渐形成了类似工业生产中的供应链。

随着复用开源软件在现代软件开发中越来越普遍，开源软件供应链已然成为现代软件行业的基石。然而，开源软件供应链在为软件开发带来便利的同时，也带来不容忽视的风险问题。一方面，供应链上任何一个软件的问题都可以沿着供应链影响大量的其他软件，使得影响被放大；另一方面，供应链上每一个软件都会受到它直接或间接复用的软件的影响，使得问题的定位与解决变得困难。近年来发生的一系列安全事件也印证了开源软件供应链上的这些风险问题。例如，2016 年 3 月，被整个前端开源生态依赖的 NPM 包 leftpad 被开发者删除，对海量前端项目的构建和部署造成严重影响，乃至影响了大量网站的正常运行[6]；2021 年 12 月，Log4j 2 被曝光远程代码执行漏洞，作为 Java 软件项目的主流日志库之一，其对政府和企业的信息安全造成的影响难以计量[7]。

虽然软件供应链被许多研究工作提及，直到近年来，才有学者尝试对软件供应链提出定义和研究框架。例如，Zhou 等人[5]在 2019 年提出开源供应链定义为“由于软件或软件项目之间互相依赖(如软件的构建或运行时依赖，开发者同时参与多个开源项目，软件代码的复制粘贴等)形成的复杂关系网络”；He 等人[8]把软件供应链定义为“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”，并进一步提出一个包含开发、交付、使用三个环节的软件供应链安全研究框架；Ji 等人[9]定义开源软件供应链为“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”，并进一步提出一个包含组件开发、应用软件开发、应用软件分发和应用软件使用四个环节的开源软件供应链安全研究框架。He 等人[8]和 Ji 等人[9]分别根据各自提出的研究框架，综述了软件供应链上软件制品生命周期内各个环节的安全风险及相关研究。然而，已有文献综述主要关注单个软件制品的生命周期中的软件供应链安全风险[8-12]，缺乏对软件供应链的整体结构的审视和对链上软件制品之间相互影响的相关分析。因此，为了深入理解开源软件供应链的复杂性、动态性和软件制品间的相互影响，本文将从供应链**整体**视角，对开源软件供应链提出新的研究框架，并基于该框架进行文献综述，方便研究者针对开源软件供应链开展进一步的研究工作。

本文第 1 节首先对开源软件供应链进行形式化定义，并综合考虑各种类型的软件制品和各种形式的代码复用，举出具体的开源软件供应链实例。第 2~5 节从这些定义和实例出发，从结构及演化、风险传播与管理和依赖管理三个方面对开源软件供应链的研究工作展开综述。第 6 节总结了该领域的研究挑战，并对未来研究方向进行展望。第 7 节对全文进行总结。

1 开源软件供应链定义

在工业生产中，供应链是一个很成熟的概念，是指在创造产品并将其交付给消费者的过程中涉及到的上游与下游企业所形成的网链结构[13]。而在软件工程领域，“软件供应链”这一概念出现较晚，最早在 1995 年被 Jacqueline Holdsworth 提出[14]，用来描述通过制定具体的标准来设计和编写软件的整个过程。此后，“软件

供应链”这一概念常被用来分析软件工程中的经济、管理和安全问题[15]。

本文参考现有工作[5,8,9]对软件供应链的定义，将**开源软件供应链**定义为开源软件制品之间在各自开发、构建、发布和运维过程中通过各种形式的代码复用形成的供应关系网络。从这个定义出发，可以将开源软件供应链在数学上抽象为一个图：

$$G = \langle V, E \rangle \quad (1)$$

其中V表示软件制品，E表示软件制品间的代码复用关系。V中的软件制品类型可以相同，也可以不同。在软件开发实践中，有 4 种典型的软件制品类型（如表 1 所示）和 5 种典型的代码复用关系类型（如表 2 所示）。

表 1 软件制品类型

软件制品	描述
软件包	一些开发者会把自己编写的软件经过打包工具打包后上传到包托管平台供其他开发者下载使用。大部分编程语言都有自己的包托管平台，比如 Python 的 PyPI，Java 的 Maven 和 JavaScript 的 NPM。此外，一些操作系统也提供了自己的包托管平台。在本综述中，软件包、库表达同一个意思。
代码仓库	代码仓库包含一个软件制品的开发历史，除了核心功能外，代码仓库通常还包含测试、文档等开发相关的文件。
二进制应用	二进制应用是源代码经过编译等流程产生的二进制可执行文件，比如 Google Play 上的安卓应用。
代码片段	开发者常常将包含某些功能的开源代码片段直接或经过修改后使用，其中相当一部分来源于编程问答网站 Stack Overflow 的回答。

表 2 代码复用关系类型

代码复用方式	描述
安装依赖	开发者会在软件制品的依赖声明文件里声明依赖的软件包和对应的版本，供包管理器自动安装相应的依赖。
API 调用	软件制品在代码中调用软件包提供的 API。安装依赖和 API 调用的区别是：安装依赖不一定会在代码里使用。比如，有的安装依赖是冗余的，有的安装依赖只能在命令行里使用。
Fork	完整克隆其他软件制品（尤其是代码仓库）。一个典型的例子是各大手机厂商对安卓操作系统的定制化。
文件拷贝	复制粘贴其他软件制品的代码文件。
代码克隆	复制粘贴或修改其他软件制品的代码片段。

2 文献收集与分析

为了对开源软件供应链的研究框架建立整体性视角，本文遵循系统文献调研的流程[16]收集和分析相关文献。系统文献调研的目标是找到尽可能多和尽可能全面的相关文献。常见的文献搜索策略主要有数据库检索和“滚雪球”两种。我们这里采用“滚雪球”的方法进行相关文献收集，主要出于以下两点原因：

1) 正如第 1 节所示，软件供应链的含义相当广泛，难以形成既精确又全面的检索词；
2) 软件供应链是近几年逐渐流行的概念，一些早期的重要相关工作可能不会使用“软件供应链”这一术语，因此使用数据库检索可能会遗漏掉重要的相关文献。

“滚雪球”方法需要先确定一组起始的论文，然后从这些起始论文出发，通过前向搜索（即收集论文引用的文献）和后向搜索（即收集引用该论文的文献）的方式迭代找到新的相关文献[17]。我们选择从近三年的 ICSE 和 FSE 会议论文中选出起始论文，考虑到以下三点：

1) 如前所述，很难形成精确有效的搜索词进行搜索；
2) ICSE 和 FSE 是软件工程领域的两个综合性会议，包容性高，涵盖的研究话题广泛；
3) ICSE 和 FSE 是 CCF A 类会议，论文代表性高，受到学界的广泛认可。

通过阅读近三年 ICSE 和 FSE 所有论文的标题、摘要和引言，我们筛选出 13 篇研究开源软件供应链的论文。以这 13 篇论文为起始论文，我们通过后向搜索和前向搜索迭代地进行论文收集和评估，不断找到新的相关论文。在论文评估时，一方面我们要找到研究开源软件供应链的论文，另一方面，为了保证论文的质量，我

们只保留 CCF 推荐的国际学术会议和期刊的长文。最终经过两轮滚雪球后，我们没有找到新的相关文献。第一轮滚雪球我们发现 52 篇新的论文，第二轮滚雪球我们发现 25 篇新的论文，最后一共获得 90 篇论文。图 1 展示了这 90 篇论文的发表年份分布。可以看到，最早进行开源软件供应链的研究可以追溯到 2012 年，并且开源软件供应链吸引到了越来越多研究者的关注。

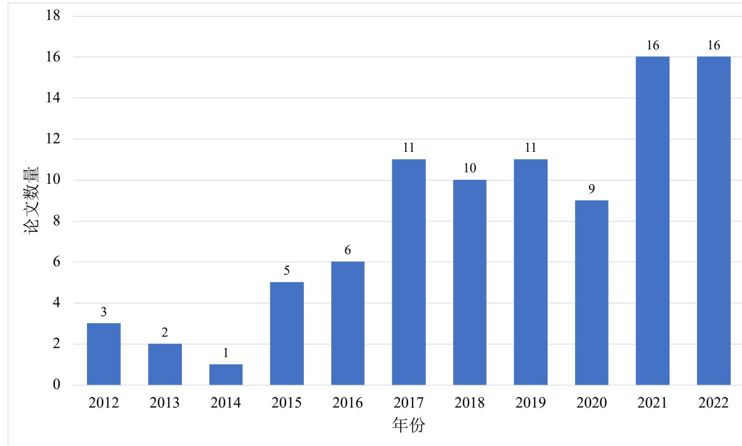

图 1　研究开源软件供应链的论文的发表年份分布

表 3　相关文献的数量分布，按开源软件供应链组成类型和研究方向划分

开源软件供应链组成类型		结构与演化	风险传播与管理	依赖管理	总计
软件制品	代码复用方式				
软件包与软件包	安装依赖	13	8	11	32
软件包与代码仓库	安装依赖	3	1	19	23
软件包与代码仓库	API 调用	2	5	7	14
软件包与软件包	API 调用	2	3	0	5
代码仓库与代码仓库	文件拷贝	4	1	0	5
代码仓库与代码仓库	Fork	0	2	1	3
代码片段与代码仓库	代码克隆	1	0	2	3
代码仓库与代码仓库	代码克隆	2	0	0	2
软件包与软件包	文件拷贝	1	0	0	1
软件包与二进制应用	API 调用	0	0	2	2
总计		28	20	42	90

我们对这些收集到的论文进行阅读，建立了一个考察供应链的整体性视角，分析它们研究的软件制品类型、代码复用方式和具体研究方向，分析结果展示在表 3 中。考虑到开源软件供应链是一个复杂、动态和相互影响的系统，我们把相关研究分为三大方向：*开源软件供应链的整体结构与演化*，*上游软件制品的风险传播与管理*，和*下游软件制品的依赖管理*，分别有 30 篇、14 篇和 46 篇论文。这三个研究方向的论文从整体到局部，由泛入微地从不同的视角对开源软件供应链展开了研究。第一个研究方向关注的是对开源软件供应链整体性质的刻画，后两个研究方向关注的是供应链上软件制品间的相互影响。

从供应链类型来看，研究得最多的供应链类型是软件包与软件包之间通过安装依赖形成的供应链（下文称为第一类供应链），软件包与代码仓库之间通过安装依赖形成的供应链（第二类供应链），和软件包与代码仓库之间通过 API 调用形成的供应链（第三类供应链），分别有 32 篇、23 篇和 14 篇论文。对于第一类供应链，研究者主要研究其整体结构与演化特征，有 13 篇论文；对于第二类和第三类供应链，研究者主要研究其依赖管理，分别有 19 篇论文和 7 篇论文。这些研究内容的区别可能与使用的数据有关。我们发现，第一类供应链的研究以各大包托管平台上软件包的元数据为数据源，这类元数据可以方便获得，其中通常包含软件包的依赖信息，使得研究供应链的整体结构与演化成为可能。第二类和第三类供应链的研究则是选择一些代码

仓库，分析这些代码仓库的依赖声明文件或代码中调用的 API 来构建供应链，因为代码仓库的依赖声明历史和 API 调用行为可以很方便地获得，所以便于分析其依赖管理行为。除了这三种类型的软件供应链外，其他类型的软件供应链研究较少，每种不超过 5 篇文献。这可能有两方面的原因：1)，一些供应链上存在的问题最近几年才被意识到；2)，大规模的代码复用检测如文件拷贝和代码克隆较为困难，需要的时间和空间资源太高。

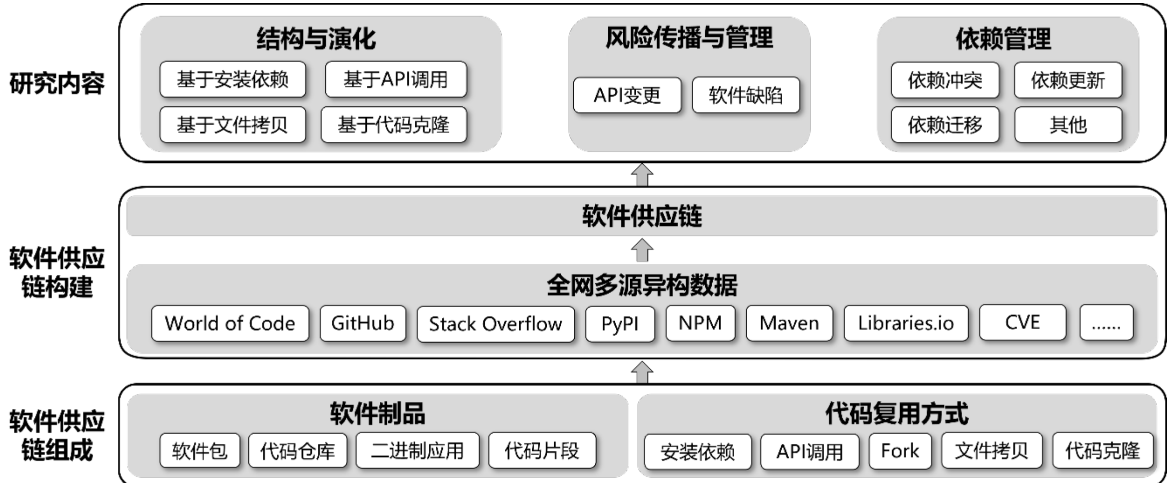

图 2 开源软件供应链研究框架

基于已有研究的软件供应链组成类型、软件供应链构建方法、和具体研究方向，我们进一步总结了如图 2 所示的开源软件供应链研究框架。对于所有开源软件供应链研究，我们首先明确其研究的软件供应链的类型，即是什么软件制品通过什么代码复用方式形成的供应链；其次，明确数据源与供应链构建方法，例如从 World of Code[18], GitHub, Stack Overflow 和/或各大包管理器平台等数据源收集数据，然后通过语言特定和代码复用方式特定的方法工具解析收集的软件制品的直接（和间接）依赖，构造相应的供应链；最后，在结构与演化、风险传播与管理和依赖管理三个方面中确定其具体的研究方向。接下来，我们将从结构与演化、风险传播与管理和依赖管理三个方面，对相关研究展开综述。

3 结构与演化

精准构建开源软件供应链并探索其结构与演化是理解开源软件供应链的第一步。通过理解开源软件供应链的结构与演化，可以帮助我们认识开源软件供应链的性质和特征，支持有效发现潜在的问题，支持核心软件的孵化和培育。形式化地看，构建开源软件供应链是通过收集一组节点（即软件制品）并确定这些节点之间的边（即代码复用）来构造图的过程。因此，构建开源软件供应链需要首先明确节点是什么，边是什么。鉴于代码复用方式是构建开源软件供应链的基础，接下来我们从代码复用方式的角度对相关研究进行综述。

3.1 基于安装依赖的开源软件供应链

安装依赖是指软件制品（一般是软件包和代码仓库）在其依赖声明文件里声明的对其他软件包及版本的依赖。对于软件包来说，其安装依赖一般在元数据里声明，可以通过其所在包托管平台的 API 获取。此外，Libraries.io[19]收集了多个包托管平台上所有软件包的元数据，方便研究人员开展软件包之间安装依赖的研究。对于代码仓库来说，其安装依赖的获取需要先把仓库克隆到本地，然后设计包管理器特定的脚本解析依赖声明文件如 packages.json，POM.xml 和 requirements.txt。因此，从构建难度上看，第一类供应链（即软件包与软件包之间通过安装依赖形成的供应链）的构建较为容易，也吸引了大量的研究。

许多编程语言都有自己的包托管平台，上面托管了大量软件包供开发者下载使用。这些软件包之间通过

安装依赖形成了（第一类）供应链。随着复用第三方软件包在现代软件开发中越来越普遍，这些供应链对各编程语言的编程社区越来越重要。研究者对多个包管理平台上第一类供应链的结构与演化进行分析和比较。Wittern 等人[20]，Kikas 等人[21]和 Decan 等人[22,23,24]分析了多个编程语言的包托管平台（包括 JavaScript 的 NPM，Ruby 的 RubyGems，Rust 的 Crates，Python 的 PyPI，Perl 的 CPAN，R 的 CRAN，.NET 的 NuGet 和 PHP 的 Packagist）上第一类供应链的结构与演化，发现这些平台上第一类供应链中软件包、直接依赖和间接依赖关系的数量都在迅速增长；其中只有一小部分软件包被大多数软件包依赖，这一小部分软件包的失效（如被维护者删除）会影响到许多其他的软件包，降低供应链的弹性。这些工作并没有考虑到库的每个版本的依赖和被依赖的情况，于是 Soto-Valero 等人[25]对 Maven 平台上库的每个版本的依赖情况进行刻画。他们分析了 Maven Central 上 7 万多个库的近 150 万个版本，发现超过 30%的库有多个版本被其他库的最新版本依赖，尤其是对于流行的库，有超过 50%的版本被其他库的最新版本依赖。他们还发现超过 17%的库有几个版本的使用量远高于其他版本。与之前工作对第一类供应链上软件制品的依赖分布进行刻画不同，Zimmermann 等人[26]分析了第一类供应链的结构可能被用来发起供应链攻击的机会。他们对 NPM 第一类供应链上软件包之间的依赖，软件包的维护者账户和公开报告的安全问题进行了系统的分析，发现单个软件包可以影响到供应链的很大一部分，其中极少数软件包的维护者账户可以被用来向大多数的软件包注入恶意代码，并且由于缺乏维护，许多软件包依赖有漏洞的软件包。

2016 年 3 月，left-pad，一个实现字符串左填充功能的只有 11 行代码的 NPM 软件包被开发者删除，而它被另一个知名的 NPM 包 Babel 使用，导致它的删除影响了包括 Facebook，Netflix 和 Airbnb 在内的大型网站的正常运行[27]。left-pad 事件引起了研究者对小包（trivial packages）的关注，并分析了这类包的普遍性、使用原因、发布原因和使用情况。Abdalkareem 等人[28,29]首先通过问卷调查的方式提出小包的定义为：如果一个 NPM 或 PyPI 软件包的代码行数不超过 35 行并且圈复杂度不超过 10，那么这个软件包就被认为是一个小包。基于这个定义，他们分析了 NPM 和 PyPI 两个平台上小包的普遍性和开发者使用小包的原因。他们发现 NPM 和 PyPI 上小包是普遍存在的，占 16.0%和 10.5%。开发者使用小包是因为他们认为小包提供了良好的实现和经过测试的代码，但是作者发现 NPM 和 PyPI 上只有 28%和 49%的小包有测试代码，且 18.4%和 2.9%的小包有超过 20 个安装依赖。Chen 等人[30]调查了开发者发布小包的原因，通过对 59 名发布小包的开发者的问卷调查，他们发现 8 个开发者发布小包的原因，包括创造可复用的组件、便于测试和记录文档、专注于具体的任务等，以及 5 种小包带来的问题，包括发布者要维护多个软件包、复杂的依赖、增加找到合适的软件包的难度和增加重复的软件包等。Chowdhury 等人[31]进一步分析了小包的使用情况。他们发现在实际 JavaScript 项目中，小包通常在核心代码文件中使用，且比非其他包被调用的次数要多。在 NPM 的第一类供应链上，小包比其他包处于更加核心的位置，移除某些小包可能影响供应链上 29%的软件包。综上，小包在软件供应链上扮演着重要的角色，但是其质量却参差不齐，开发者在使用小包时应该慎重选择。

除了对基于安装依赖的供应链的依赖结构进行刻画外，研究者还分析了这类供应链的结构对链上软件包的影响。比如 Valiev 等人[32]分析了软件包在第一类供应链上的位置对它的可持续性的影响。他们使用 Katz 中心度来度量软件包在供应链上的位置，通过拟合 Cox 回归模型，发现 Katz 中心度与软件包不活跃的机率有显著正相关关系。Dey 等人[33]利用软件包供应链的信息来预测 NPM 软件包的流行度的变化，发现一个软件包的上游或下游的数量变少或下载量的增加会提高它的下载量增加的机率。他们进一步分析了 NPM 软件包供应链上开发者的贡献行为[34]，发现开发者主要向他们直接依赖的软件包提 issue 和 PR，而向间接依赖的软件包提 issue 和 PR 较少。

3.2 基于API调用的开源软件供应链

除了在依赖声明文件里声明安装依赖外，软件制品还可以通过调用软件包的 API 来复用软件包提供的功能。API 调用与安装依赖的区别在于，软件制品的代码中不一定会调用安装依赖的 API，因为安装依赖可能是冗余的，或者只能在命令行中使用。从这个角度看，分析 API 调用可以获得比分析安装依赖得到更精细的代码复用关系。然而相比分析安装依赖，分析软件制品间的 API 调用更加复杂。一方面，需要对下游软件制品

的代码进行分析，另一方面，还需要构建上游软件包提供的 API 的数据库。研究者尝试对两种静态语言 Java 和 Rust 语言的软件包构建基于 API 调用的供应链。Hejderup 等人[35]实现了一个原型 PRÄZI 构建了 Rust 的包托管平台 Crates 上软件包之间基于 API 调用的供应链，并与基于安装依赖的供应链进行比较，发现平均每个软件包只调用 78.8%的直接依赖的 API 和 40%的间接依赖的 API。Harrand 等人[36]则分析了 Java 的 Maven 平台上库之间的 API 调用关系。他们从 Maven Central 上收集了 94 个流行库和（安装）依赖这些库的 829410 个下游软件包。通过分析下游软件包的字节码，他们发现 41.3%的安装依赖的 API 并没有出现在字节码中，而对于所有的库来说，只有一小部分 API 会被其绝大多数的下游使用。这些结果表明尽管都是对软件包的代码复用，基于 API 调用的供应链与基于安装依赖的供应链有很大区别，后者会在分析漏洞等供应链风险传播时产生过高估计。

为了帮助研究者进行 API 调用层次的供应链研究，Ma 等人[18]构建了 World of Code。World of Code 收集了全网 1.7 亿多个公开的 Git 代码仓库，并对这些仓库中使用主流编程语言编写的代码文件里的依赖导入语句（如 Python 代码中的 import 语句）进行解析，构建了代码文件与调用的软件包的映射。基于 World of Code，研究者针对通过 API 调用产生的软件供应链展开研究，比如 Ma 等人[37]分析了这种类型的供应链对软件包采用的影响，Tan 等人[38]研究了 Python 深度学习供应链。Tan 等人通过 World of Code 提供的代码文件与调用的软件包的映射和 Libraries.io 记录的软件包的仓库地址，找到直接和间接依赖 TensorFlow 和 PyTorch 的软件包和代码仓库，构建了 TensorFlow 和 PyTorch 供应链。她们分析了这两条供应链的结构、应用领域和流行因素，发现虽然这两条供应链包含数十万个仓库，但是都只有五六层，且约 90%的项目在第二层，即直接依赖 TensorFlow 或 PyTorch。

3.3 基于文件拷贝的开源软件供应链

在软件开发中，从其他代码仓库或软件包中拷贝完整的代码文件到自己的项目中也是很普遍的。基于文件拷贝形式的代码复用的检测可以通过比较文件的哈希实现。一些研究者对代码仓库和软件包之间文件拷贝的普遍性和维护情况进行分析。Lopes 等人[39]分析了 GitHub 代码仓库中文件拷贝的普遍性。他们分析了 450 万个非 fork 仓库中的 4.28 亿个 Java，C++，Python 和 JavaScript 代码文件，发现其中只有 8500 万个唯一的文件。不同语言的文件拷贝情况有很大区别，其中 JavaScript 代码文件的重复率最高，只有 6%的文件是独一无二的，而 Java 代码文件的重复率最低，有 60%的文件是独一无二的。Hata 等人[40]分析了相同的文件在不同的项目里是如何被维护的。他们分析了 32007 个 GitHub 仓库中的约 2800 万个重复的代码文件，发现这些重复的文件一搬是库、驱动文件和配置文件等，但只有不到 40%的文件会被所在的项目维护。不同于这两篇工作分析单个文件在仓库间的重复情况，Wyss 等人[41]研究了 NPM 软件包之间的重复情况，即一个软件包完全复制或近乎完全复制另一个软件包的代码。他们提出一个工具 Unwrapper 来检测 NPM 上软件包的重复情况。Unwrapper 基于目录树的相似度和文件内容的相似度来度量两个包之间的差异，可以在 72.85s 内快速比较一个软件包和 NPM 已有的所有软件包。利用 Unwrapper，他们估计 NPM 上约有 10.4%的软件包是重复的。

除了代码文件与调用的软件包的映射，World of Code 还提供了代码文件到代码仓库的映射。通过其提供的 API，可以快速找到一个代码文件存在于哪些项目中，便于开展基于文件拷贝的开源软件供应链的研究。比如 Imam 等人[42,43]分析了黑客马拉松项目中的代码文件复用情况。他们分析了 22183 个黑客马拉松项目，通过 World of Code，他们发现这些项目中 85.56%的代码文件和 78%的代码行数是在黑客马拉松之前创建的，其中 69.54%的代码文件和 38.2%的代码行数是由黑客马拉松队伍外的开发者创建的。这表明黑客马拉松项目会大量复用其他项目的代码，这可能与黑客马拉松项目要在有限的时间里完成有关。

3.4 基于代码克隆的开源软件供应链

代码克隆形式的代码复用在软件开发中也很普遍，开发者会复用其他项目或者 Stack Overflow 上的代码片段。目前已经有很成熟的代码克隆检测工具，如 CCFinder[44]，Deckard[45]和 SourcererCC[46]等。已有研究利用这些工具研究了代码克隆在软件制品中的普遍性。比如 Abdalkareem 等人[47]分析了安卓应用克隆 Stack

Overflow 上代码片段的情况。他们分析了 22 个开源的安卓应用代码仓库，发现这些仓库中或多或少都使用 Stack Overflow 上代码片段，平均 1.06%的代码（按代码行数衡量）克隆自 Stack Overflow。开发者使用 Stack Overflow 上代码片段的主要原因是实现新功能和增强已有功能。然而，作者也发现文件在使用 Stack Overflow 上代码片段后出现 bug 的比例会更高，所以开发者应该注意使用的 Stack Overflow 上代码片段的质量和后期维护。Gharehyazie 等人[48,49]分析了 GitHub 上代码仓库间的代码克隆情况，发现仓库间代码克隆占仓库内所有代码克隆的 10%到 30%，且相当一部分的仓库间代码克隆是把整个文件、文件夹甚至整个项目拷贝过来并进行微小的改动。他们还发现仓库间代码克隆现状像一个洋葱模型：大部分代码克隆都是克隆自同一个仓库，然后是克隆自同一领域的仓库，很少是克隆自不同领域的仓库。

3.5 小结

相关工作对基于安装依赖、API 调用、文件拷贝和代码克隆等代码复用方式形成的开源软件供应链的结构与演化开展了不同程度的研究。尽管不同类型的软件供应链（如基于 API 调用的供应链和基于安装依赖的供应链）的图结构不尽相同，但是这些图都具有如下相同的特点：每个节点普遍与多个节点相连，体现了图的复杂性；小部分节点可达图中大部分节点，体现了图的脆弱性。得益于数据的容易获得，基于安装依赖的开源软件供应链的结构与演化被广泛研究。而基于其他代码复用方式的供应链，要么由于代码复用检测复杂，要么由于大规模数据收集复杂，目前的研究仍然较少，且或者集中在某些编程语言如 Java，或者是在小规模数据上进行。所有研究建立在数据基础之上，构造面向开源软件供应链的数据基础设施是当下开展供应链相关研究的迫切需要。基于这些数据基础设施，对不同编程语言、不同领域和不同类型的供应链的结构与演化进行横向比较，探究其共性和差异，对供应链的管理与维护也很有意义。

4 风险传播与管理

在开源软件供应链中，一个软件制品可能同时扮演着上游和下游的角色，这使得一个软件制品的风险可以沿着供应链传播产生连锁反应，影响大量的下游软件制品。为了避免或控制供应链中的风险影响，许多工作探索风险在开源软件供应链上的传播和管理实践。从形式上看，风险传播是在构建的图中，识别节点的变动（如 API 变更，软件缺陷等），并找到该节点可达的所有节点（即下游软件制品）中受该变动影响的节点。相关研究主要对 API 变更和软件缺陷两种类型的风险展开研究。

4.1 API变更

软件包会不断地发布新版本（亦即，持续演化），在新版本中往往会加入新的特性、软件缺陷（例如软件故障和安全漏洞）的修复、以及代码重构等。在这个过程中，软件包的 API 会发生变更，影响到下游使用这些 API 的项目。已有研究主要从上游 API 变更的引入情况、API 变更的影响范围和 API 变更的应对实践三方面展开。

4.1.1 API 变更的引入情况

在软件包演化过程中，会引入各种各样的 API 变更，其中会影响下游代码正常运行的 API 变更被称为不兼容变更（breaking change）。不兼容更改会使得已有代码无法兼容新版本，影响下游用户的升级。已有研究主要分析了不兼容变更的普遍性、是否遵守 Semantic Versioning 和引入的原因，且主要是在 Java 语言的软件制品中展开。

关于不兼容变更的普遍性，已有研究在不同规模的数据集上的分析结果都证实了不兼容变更在软件包演化过程中是普遍存在的。比如 Wu 等人[50]分析了 22 个 Apache 和 Eclipse 项目中类、接口和方法的变更情况，发现移除类和方法在这些框架里经常发生；Xavier 等人[51]分析了 317 个 Java 库中不兼容变更的数量，发现 14.78%的 API 变更为不兼容变更，且项目引入不兼容变更的频率随时间推移而增加；Raemaekers 等人[52]分析了 Maven Central 上 22205 个软件库的约 15 万个版本，发现约 1/3 的版本引入至少 1 个不兼容变更。

为了使用户了解一个版本是否包含不兼容变更，Tom Preston-Werner 提出了 Semantic Versioning[53]规范。

Semantic Versioning 规定了版本号为 MAJOR.MINOR.PATCH 的格式，并约定：若新版本引入了不兼容变更，则应该增加主版本号（MAJOR）；若新版本没有不兼容变更，且引入了向下兼容的新功能，则应该增加次版本号（MINOR）；若新版本仅仅引入了向下兼容的问题修正，则应该增加修订版本号（PATCH）。然而已有研究发现相当一部分软件包并不会遵守这一规范。比如 Raemaekers 等人[52]使用 Clirr 工具[54]来分析一个软件包相邻两个版本的 jar 包中包含的不兼容变更。他们分析了 Maven Central 上 22205 个软件库的约 15 万个版本是否遵守 Semantic Versioning 规范，发现主版本和次版本中不兼容变更的数量几乎没有不同。然而 Clirr 工具检测不兼容变更的准确性不高[55]，比如无法识别异常和继承相关的不兼容变更，Ochoa 等人[56]使用更先进的不兼容变更检测工具 japicmp[55]对 Raemaekers 等人[52]的工作进行了一项外部和差异化的复现研究。他们发现尽管 83.4%的版本是遵守 Semantic Versioning 规范的，但是仍有 20.1%的次版本和修订版本包含不兼容变更。他们还发现遵守 Semantic Versioning 规范的版本比例随时间上升，包含不兼容变更的次版本和修订版本的比例从 2005 年的 67.7%下降到 2018 年的 16.0%。尽管 japicmp 可以解决 Clirr 的一些不足，但是它仍不能识别泛型相关的不兼容变更。此外，japicmp 在识别修饰符相关的不兼容变更上仍有改进空间。

在引入不兼容变更的原因方面，Brito 等人[57]总结了 Java 库的开发者引入不兼容变更的三大原因：支持新的特性、简化 API 和改善可维护性，并且开发者主要使用版本发布纪要（release note）和更改日志（changelog）记录重大更改。研究还发现软件生态系统的规范也会影响开发者引入不兼容变更的行为[58,59]，比如 Eclipse 社区非常重视长期稳定性，因此 Eclipse 社区的开发者会尽量避免引入不兼容变更。

4.1.2 API 变更的影响范围

已有研究主要分析了两种类型的 API 变更对下游项目造成的影响：不兼容变更和 API 弃用（deprecation）。与不兼容变更会破坏已有代码不同，API 弃用并不会破坏已有代码，它只是作为一种信号，提醒开发者该 API 可能会在未来的某个版本被删除。因此，为了避免未来升级时软件会出现异常行为，开发者应及时对弃用的 API 进行调整。

关于不兼容变更的影响，尽管软件包引入不兼容变更很常见，但是已有研究均发现不兼容变更涉及的 API 很少被下游项目使用，因此受影响的下游项目较少。如 Wu 等人[50]设计了工具 ACUA 以分析 Apache 和 Eclipse 生态中 22 个框架的不兼容变更对 213 个下游项目的影响，发现下游项目中 35%的类和接口会使用上游框架提供的 API，但是只有 11%的使用会受到上游 API 变更的影响。然而 ACUA 无法识别基于反射机制的 API 使用方式，这种情况可能需要复杂的静态分析或动态分析才能识别。Ochoa 等人[56]设计了工具 Maracas 分析了上游不兼容变更对近 30 万个 Java 项目的影响，发现大多数不兼容变更涉及的 API 并不会被下游项目使用，只有 7.9%的下游项目会受到上游不兼容变更的影响。尽管 Maracas 可以以非常高的准确率和召回率识别不兼容变更对下游项目的影响，但是它仍存在几点不足，比如不能识别通过方法覆写的方式调用发生不兼容变更 API 的影响，不能识别 strictfp 和 native 修饰符相关的不兼容变更的影响等。

已有工作主要分析 API 弃用对 Smalltalk 和 Java 项目的影响。对于 Smalltalk 语言，Robbes 等人[60]分析了 Squeak 和 Pharo 两个软件生态中上游 API 的弃用对下游的影响。他们识别出 577 个被弃用的方法和 185 个被弃用的类，发现只有很少一部分 API 弃用（14%的方法弃用和 7%的类弃用）会对下游项目造成影响，这可能由于大部分弃用的 API 只在上游项目内部使用。对于 Java 语言，Sawant 等人[61,62]分析了 Java 项目是如何对 5 个流行的第三方库的 API 弃用和 JDK 中的 API 弃用做出反应的。他们发现不同第三方库的下游项目受 API 弃用影响的比例变化很大，比如 Spring 的下游项目受影响的比例几乎为 0，而 Easymock 和 Guava 的下游项目受影响的比例超过 20%。

4.1.3 API 变更的应对实践

研究者主要研究了 Smalltalk 和 Java 下游项目是如何应对 API 弃用的。研究结果发现两种语言的下游项目大部分都不会对上游 API 的弃用作出反应，而做出反应的项目处理弃用 API 的方式也不同。这可能由于 API 弃用并不会影响项目的运行，或者处理弃用的 API 需要的成本和复杂度较高[63]。对于 Smalltalk 项目，Robbes 等人[101]发现只有约 1/5 使用弃用 API 的项目会对弃用的 API 做出调整，然而开始做出调整的时间和调整完成

的时间可能会非常长，Hora 等人[64]发现下游项目做出的反应可能不同，比如使用不同的 API 替换变更的 API。对于 Java 项目，Sawant 等人[63]发现下游项目应对上游 API 弃用的 5 种模式：不处理、删除弃用的 API、用另一个 API 替换弃用的 API、用内部实现的方案替换弃用的 API 和回滚到之前的 API 版本。他们分析了 20 万个 GitHub 上的 Java 项目，发现约 88%的项目选择不处理这些弃用的 API，只有 0.02%的项目会用推荐的 API 替换弃用的 API。

4.2 软件缺陷

上游软件制品的缺陷（如安全漏洞、软件故障、恶意代码等）会沿着供应链传播，对大量下游项目造成影响。如何评估并控制上游软件缺陷在软件供应链上的影响，是保障软件供应链安全的关键。目前研究主要从缺陷的传播和上下游对缺陷的应对实践两个方面展开。

4.2.1 缺陷的传播

软件缺陷可以通过不同形式的代码复用被引入到下游项目。目前关于缺陷传播的研究大都是针对安装依赖形式的代码复用，还有一些文献研究了通过文件拷贝和 API 调用等代码复用形式发生的缺陷传播。此外，软件供应链攻击也是近年的研究热点，

基于安装依赖的缺陷传播分析。研究发现，通过直接或间接依赖，一个有漏洞的软件包很容易进入大量代码仓库或软件包的依赖树。比如 Decan 等人[65]分析了 269 个 NPM 软件包中存在的 399 个安全漏洞对 NPM 上第一类供应链的影响，发现这些漏洞可以影响到 7 万多个软件包。然而这篇工作并没有考虑到 NPM 特有的依赖解析规则，Liu 等人[66]在考虑 NPM 的依赖解析规则后，构建了更准确的 NPM 上第一类供应链，发现 20%的软件包直接或间接依赖存在有漏洞的软件包。除了对其他软件包的影响，Zerouali 等人[67]还分析了有漏洞的软件包对下游代码仓库的影响，发现约有 2/3 的 JavaScript 和 Ruby 项目会间接依赖有漏洞的 NPM 软件包和 RubyGems 软件包。不同于这些工作分析具体的漏洞在供应链的传播，Massacci 等人[68]借用金融中杠杆的概念提出了软件工程中的技术杠杆，即软件项目的直接和间接依赖的库的代码行数和使用的编程语言标准库的代码行数与软件项目本身的代码行数的比值。Massacci 等人分析了 Maven 项目的技术杠杆与有漏洞风险的关系，发现高杠杆率会导致代码带有漏洞的风险的概率增加 60%。前面这些工作都只关注了同一个语言生态内的漏洞传播，但是漏洞可能会跨语言生态传播，比如 Python 和 Java 项目可能会依赖用 C 语言编写的第三方软件包。因此，如果 C 软件包发现漏洞，依赖它的 Python 和 Java 项目也会受到漏洞的影响。Xu 等人[69]设计了工具 Insight 来检测依赖有漏洞的 C 软件包的 Python 和 Java 项目。通过对 PyPI 和 Maven 生态的分析，他们识别出 8 万多个直接或间接依赖有漏洞的 C 软件包的 Python 和 Java 项目。

尽管软件会依赖各种各样的软件包，但是这些软件包的作用是不同的，有的只在构建的时候用到，有的只在测试的时候用到，有的只是运行时需要。比如，通过对 100 个 JavaScript 项目的分析，Latendresse 等人[70]发现只有不到 1%的依赖是在生产环境中需要的。Pashchenko 等人[71]分析了 SAP 公司使用最多的 200 个 Java 软件库，发现约 20%受漏洞影响的软件库实际并没有被部署到生产环境。这些发现说明在分析漏洞传播时，考虑依赖的实际使用场景可以大大减少分析的工作量。

基于 API 调用的缺陷传播分析。（直接或间接）依赖有缺陷的软件包并不意味着会受到缺陷的影响，比如受缺陷影响的上游 API 并没有被下游项目调用，即使下游项目调用了有缺陷的上游 API，其调用方式并不会触发缺陷。因此，基于 API 调用对缺陷传播进行可达性分析和可触发性分析可以帮助我们更清楚地了解到哪些下游项目会真正受到缺陷的影响。可达性是指下游项目调用了有缺陷的上游 API，可触发性是指下游项目对有缺陷的上游 API 的调用方式可能触发缺陷。Ma 等人[72]提出一种方法来估计上游 API 的 bug 对下游项目的影响，该方法通过分析下游项目对有 bug 的上游 API 的调用路径，并结合 bug 的触发条件，利用 SAT 求解器来判断下游项目的模块是否会触发上游 bug。他们采用这个方法分析了 31 个有 bug 的上游 API 在 22 个流行 Python 项目的 121 个版本中的影响。这 121 个版本中有 25490 个模块调用了有 bug 的上游 API，但是只有 1132（4.4%）个模块可能会触发上游 bug。此方法可以准确定位出有 bug 的上游 API 影响的下游模块，适用于上游发现 bug 时，下游项目评估上游 bug 的影响。但是此方法需要人工从问题报告中分析 bug 触发条件，

导致该方法有两点局限：此方法需要问题报告中有明确的 bug 触发条件描述；此方法依赖大量的人工分析，容易出错。未来可以探索从问题报告中自动提取 bug 触发条件以解决上述局限。

基于文件拷贝的缺陷传播分析。如果下游软件制品拷贝了上游软件制品中有漏洞的代码文件，那么下游软件制品就会受该漏洞的影响。基于文件拷贝的缺陷传播可以通过比较文件的哈希是否在项目的目录树中来识别。Reid 等人[73]基于 World of Code 设计了工具 VDiOS 来识别有漏洞的代码文件在开源代码仓库中的普遍性和维护状态。VDiOS 工具以一个包含漏洞的代码文件作为输入，基于 World of Code 提供的 API 找到该文件在 World of Code 收集的所有代码仓库中的完整拷贝和后续修改，然后根据上游修复漏洞后的代码文件，将找到的文件分为三个状态：未修复漏洞、已修复漏洞和未知状态。通过对四个已知漏洞（包括 OpenSSL 的 Heartbleed 漏洞）的案例研究，他们发现即使是活跃和很流行的项目也存在大量未修复的漏洞。VDiOS 只能找到完整拷贝有漏洞的代码文件的项目，如果项目对有漏洞的代码文件进行修改或者只拷贝部分代码片段，VDiOS 就无法识别出来。此外，由于 World of Code 每半年更新一次，VDiOS 检测的结果会有延迟。

软件供应链攻击分析。Java，Python，JavaScript 等语言都有中心化的软件包托管平台，负责软件包的分发。攻击者常在这些平台上分发恶意软件包实施软件供应链攻击，损害用户的安全，比如盗取开发者的证书。一些研究者通过实际的软件供应链攻击事件，总结软件供应链攻击的特征，并设计工具识别软件供应链攻击。Duan 等人[74]设计了一个框架，从安全机制、涉众和攻击消解技术等方面比较分析了 PyPI，NPM 和 RubyGems 三个软件包托管平台，总结了它们的攻击向量。基于这些发现，他们综合元数据分析、静态分析、动态分析等技术设计了面向包托管平台的恶意软件包扫描工具 MalOSS，并在分析的三个平台上发现 339 个新的恶意软件包。Vu 等人[75]设计了工具 Lastpymile 来检测软件包的代码仓库和分发之间的文件差异，以减少恶意代码扫描工具要扫描的文件数量。Gu 等人[76]分析了软件包生命周期中涉及的官方软件包托管平台、软件包镜像站和软件包用户，识别了 12 个潜在的攻击向量，比如跳转劫持攻击、针对镜像站特有软件包的覆盖攻击等。在此基础上，他们设计了工具 Rscouter，对软件包的元数据和事件进行分析，识别潜在的安全问题。Ladisa 等人[77]通过对科学文献和灰色文献进行分析，总结了一个包含 107 个攻击向量的通用开源软件供应链攻击分类树，以及应对这些攻击的 33 个措施。

4.2.2 缺陷的应对实践

关于缺陷的应对实践，研究者主要从缺陷的修复时间和缺陷的修复方式两个方面展开。

研究者分析了不同供应链上缺陷的修复时间，发现上下游软件制品修复漏洞往往需要数月甚至数年的时间，这增加了漏洞被攻击者利用的机会。比如 Alfadel 等人[78]发现 Python 上游软件包中的漏洞一般需要三年的时间才会被披露，且一半的漏洞是在漏洞披露后才被修复的；Prana 等人[79]发现 Java、Python 和 Ruby 的下游项目中漏洞持续时间较长，一般要 3 到 5 个月才会被修复；Zhang 等人[80]分析了安卓内核供应链上的补丁传播时间。安卓内核供应链的最上游是 Linux 内核，然后安卓开源项目（AOSP）基于 Linux 内核增加许多针对移动设备的新特性，形成了 AOSP 内核，接着芯片供应商如高通基于 AOSP 内核增加额外的硬件特定的更改，最后 OEM 供应商如三星和小米基于芯片提供商的内核进行个性化的定制。Zhang 等人发现安卓内核供应链上近一半的漏洞是在上游修复 200 天后甚至更长时间才在 OEM 设备上修复，10%-30%的漏洞是在上游修复一年后才在 OEM 设备上修复。漏洞修复的延迟主要是由于目前打补丁的实践和 OEM 供应商缺乏对安全相关的上游提交的了解。Hou 等人[81]对分析了来自 153 个供应商的 6261 个安卓固件中的漏洞修复时间，发现 24.2%的固件需要至少一个月来修复漏洞，平均每个固件修复漏洞的时间为 3.2 个月。

研究者从更新依赖版本、上下游合作等角度分析了在软件供应链中软件缺陷是如何被修复的。Decan 等人[65]发现 NPM 上 44%的下游软件包可以通过声明宽松的版本约束与上游软件包同时修复安全漏洞，超过 40%受影响的下游软件包则因为使用的依赖版本约束不能自动更新到上游软件包修复安全漏洞的版本，这种情况需要下游软件包在上游软件包发布修复漏洞的版本后尽快升级。但是 Chinthanet 等人[82]发现下游软件包并不一定会很快更新到上游修复漏洞的版本，这可能与更新版本需要花费很多努力来消除版本间的不兼容有关。Ma 等人[83]分析了下游开发者是如何处理因上游 bug 引起的 bug 的。她们发现下游开发者定位到上游 bug 是

困难的，而堆栈记录信息、与上游开发者沟通和熟悉引起 bug 的上游项目是对定位上游 bug 最有帮助的三个因素。下游开发者在定位到上游 bug 后，会采用不同的处理方法，比如本地提出一个临时的解决方案、限制上游项目的版本和等待上游项目修复等。Lin 等人[84]发现 Debian 和 Fedora 两个 Linux 发行版中，13.3%的上游软件包 bug 是由发行版的开发者修复的。

4.3 小结

相关工作对开源软件供应链上 API 变更和软件缺陷两种类型风险的传播与控制展开研究，分析了其引入情况、影响范围和应对实践。研究结果表明风险可以沿着供应链对大量下游软件制品造成影响，而由于不清楚这些风险或处理这些风险的成本和复杂度较高，很多下游软件制品并不会处理这些风险或者很晚才会处理。形式上看，图（即软件供应链）中上游节点的变动可以影响到大量的下游节点，这与软件供应链的图结构是一致的，但是下游节点很少根据上游节点的变动而变动。

总的来说，开源软件供应链上的风险可以分为良性风险和恶意风险。良性风险是由上下游软件制品的异步演化造成的，即上游软件制品（通常是软件包）在演化过程中会引入 API 变更，代码缺陷等，这些风险是不可避免的。当良性风险出现时，它会立即沿着软件供应链传播，对下游软件制品的运行和维护产生影响。增强上下游之间的沟通和协作是降低良性风险的有效措施。下游用户对良性风险的应对实践取决于风险种类、处理成本等因素。恶意风险是攻击者通过分发包含恶意代码的软件制品来危害开发者的系统和隐私安全。恶意风险需要利用各种手段吸引开发者下载使用进行传播，因此恶意风险的传播是可以阻断的。通过分析攻击向量，识别出恶意风险的传播途径，进而设计安全机制可以从源头阻断恶意风险的传播。

然而我们也注意到目前研究仍存在两方面的问题。一方面，相关研究主要关注某些编程语言。比如关于 API 变更的研究主要关注 Java 语言，而缺少对其他编程语言的分析。这主要是因为其他编程语言的代码分析工具的缺乏。另一方面，相关研究分析风险传播的方式比较单一。比如相关工作主要研究了漏洞通过安装依赖的传播，缺乏对漏洞通过其他代码复用方式传播的分析，可能会导致漏洞影响的误报和漏报。考虑不同代码复用方式下漏洞的传播，将更精确地评估漏洞在供应链上的影响。

5 依赖管理

一个软件制品可以通过安装依赖、文件拷贝、代码克隆等方式复用其他软件制品，这些软件制品又可以进一步复用其他软件制品，进而导致一个软件制品有大量直接或间接依赖，并受其影响。因此，如何管理这些依赖是一个重要问题。研究者主要从依赖冲突、依赖更新、依赖迁移三个方面展开研究。此外，依赖降级、许可证合规、变更移植等主题等也被相关研究涉猎到。

5.1 依赖冲突

相关工作主要从依赖冲突的类型和依赖冲突的检测与修复两个方面展开。

相关文献主要研究了以下四种类型的依赖冲突：

- **同一依赖的不同版本约束的冲突。**通过直接依赖或间接依赖的方式，下游项目可能存在对同一软件包的多个版本约束，如果没有一个版本可以同时满足这些版本约束，冲突就会发生[85-89]。根据依赖管理工具的不同，该类型的冲突会有不同的表现形式，比如对于 Java，JVM 会根据某种版本仲裁策略选择其中一个版本，然后遮蔽其他版本。如果加载的版本中不包含软件项目需要调用的类或方法，则会引发软件崩溃[86,87]；对于 Python，如果没有一个可以同时满足多个版本约束的版本，软件制品则会构建失败[88]。
- **同一依赖的不同版本的兼容性冲突。**软件包在演化过程中会不可避免地引入不兼容变更，进而导致一个软件包的不同版本之间存在兼容性冲突。比如 Jia 等人[90]分析了软件包相邻版本之间移除一个 API 和增加一个 API 两种类型的不兼容变更，Wang 等人[91]分析了 Java 库相邻两个版本同一个 API 的语义冲突。

- **不同依赖的资源冲突。**不同依赖在全局共享资源如配置文件、文件格式、命名空间上也可能产生冲突[85,92]。比如，JavaScript 并不提供显式的命名空间，这意味着所有库共享一个全局命名空间。因此，一个库的值或函数可能很容易被其他库覆盖、修改、删除，进而导致下游应用出现意外行为。Patra 等人[92]通过分析真实的案例，总结了四种类型的 JavaScript 不同库的资源冲突：导入冲突、类型冲突、值冲突和行为冲突。
- **不同依赖管理模式的冲突。**Go 语言在演化过程中产生两种依赖管理模式。在 Golang 1.11 之前，Go 项目通过 GOPATH 模式引用其他库。GOPATH 模式通过 go get 命令把指定 URL 的库的最新版本下载安装到本地。为了克服 GOPATH 只能安装最新版本的限制，Golang 1.11 引入了更加灵活的 Go Modules 模式，允许一个库使用不同的引入路径来引用一个库的不同版本。两种模式对引入路径的解析不同，进而产生冲突。Wang 等人[93]发现三种发生依赖冲突的场景：使用 GOPATH 模式的项目依赖使用 Go Modules 模式的项目、使用 Go Modules 模式的项目依赖使用 GOPATH 模式的项目和使用 Go Modules 模式的项目依赖使用 Go Modules 模式的项目。

关于依赖冲突的检测与修复，目前工作主要首先从依赖冲突报告中总结冲突的表现形式和开发者采取的修复策略，然后基于冲突的表现形式和修复策略，设计自动化的冲突检测工具。冲突检测工具的设计主要有两种思路，一种是构建项目的依赖树检测冲突，然后利用 SAT 求解器等工具找到满足所有约束条件的依赖版本集合；另一种是通过构造测试样例触发冲突。下面，本文针对不同类型的依赖冲突介绍相关工作。

- **同一依赖的不同版本约束的冲突。**该类型的依赖冲突主要是通过构造依赖树来检测，然后利用 SAT 求解器等工具进行修复。Abate 等人[94]提出一个工具 distcheck，基于软件包的依赖配置文件中声明的依赖关系，利用 SAT 求解器检测软件包是否可以被成功安装；Wang 等人[88]实现了一个工具 Watchman 对 PyPI 上由于库更新导致的依赖冲突进行持续监控，该工具通过分析每个软件包的直接和间接依赖图，检测依赖冲突，并提供诊断信息帮助开发者理解冲突的导致原因，促进冲突的修复；Li 等人[89]设计了工具 NuFix，在.NET 项目的依赖图上利用二进制整数线性优化模型找到符合开发者配置偏好的软件包与平台版本；Huang 等人[87]设计了工具 LibHarmo 来修复 Java 项目子模块间依赖版本不一致的问题，该工具首先分析 POM.xml 文件识别版本不一致的库，然后通过库的 API 调用情况以及与开发者的交互，推荐修复成本最小（比如要删除和改变的 API 调用数量最少）的版本。LibHarmo 同样局限于静态分析的缺点，不能处理如反射等形式的 API 调用，会影响修复成本的估计。
- **同一依赖的不同版本的兼容性冲突。**Jia 等人[90]设计了工具 DepOwl 来检测同一 C/C++软件包不同版本因为 API 移除或 API 增加导致的兼容性冲突。DepOwl 首先检测一个软件包相邻版本间移除和增加的 API，然后检测这些 API 是否在下游项目中使用，进而确定下游项目可以使用的软件包版本。DepOwl 依赖带调试符号（debug symbols）的二进制文件来收集软件包版本间的不兼容变更和下游项目中的兼容性冲突，然而大部分软件包和下游项目的二进制往往不带调试符号。为了解决这个局限，DepOwl 采用默认编译指令将源代码编译成带调试符号的二进制，但由于实际编译指令可能与默认编译指令不同，DepOwl 会产生误报。Wang 等人[91]提出基于构造测试用例的工具 Sensor 来触发 Java 库不同版本间的具有相同签名的 API 语义不一致冲突。
- **不同依赖的资源冲突。**Patra 等人[92]提出了工具 ConflictJS 来自动检测 JavaScript 库之间在全局命名空间上的冲突。ConflictJS 包括两步，第一步动态分析每个库在全局命名空间写的位置识别潜在的存在冲突的库，第二步合成客户端并比较它在不同潜在冲突的库的配置下的行为，如果行为不同，那这两个库是冲突的。
- **不同依赖管理模式的冲突。**Wang 等人[93]分析了 Go 项目两种依赖管理模式发生依赖冲突的原因：GOPATH 和 Go Modules 解释导入路径的方式不同和 Golang 生态并不严格要求执行语义版本发布规则，以及 8 种修复策略，如使用 Go Modules 的项目严格遵守语义版本发布规则、更新改变仓库位置

的库的导入路径等。基于上述发现的原因和修复策略，她们设计了基于项目依赖树的工具 Hero 来自动检测依赖冲突并建议修复策略。

此外，还有一些工具对多种类型的依赖冲突进行评估。比如 Wang 等人[86]设计了一个静态分析工具 Decca 来帮助开发者诊断软件项目中依赖冲突问题的严重等级和维护成本；她们还设计了工具 Riddle[95]，结合条件变异、搜索策略和条件恢复等技术自动生成测试用例来触发依赖冲突引发的软件崩溃，收集软件崩溃的堆栈记录信息，帮助开发者了解冲突。

总的来说，依赖冲突检测是一个检测供应链上某节点（即下游软件制品）与其所有祖先节点（即上游软件制品）构成的子图中边的属性要求（如版本约束、API 调用）能否同时满足的过程。修复依赖冲突的思路是调整边的某些属性，以使子图中所有边的属性要求自洽。

5.2 依赖更新

依赖更新可抽象为图中边的版本约束属性与其连接的上游节点的最新版本保持一致的过程。上游软件制品在不断演化，增加新的特性，修复 bug，满足行业标准或者改善性能等。从这个角度看，下游软件制品应该及时更新到新的版本[96]。但是上游软件制品的更新往往会伴随着不兼容变更，导致下游软件制品的更新有很高的成本，很多下游软件制品仍在使用旧的上游版本。相关工作主要从下游软件制品更新的行为、(不）更新的原因和自动化更新工具的效果三个方面展开研究。

相关工作主要对安卓和 Java 项目的依赖更新行为展开研究，发现大部分项目不会更新它们使用的依赖，即使更新，也是在上游依赖新版本发布数月后进行。对于安卓项目，已有研究发现大部分开发者不会更新所有依赖的安卓 API[97]或第三方软件库[98]，发生更新的第三方库主要是 GUI 相关的，为了使应用与最新的设计趋势同步[99]。对于 Java 项目，已有研究发现约 80%的项目在使用过时的依赖[100-102]，当依赖的项目新版本包含大量 bug 修复时，下游项目更倾向于更新依赖，而当依赖的项目新版本有大量 API 变更时，由于升级需要进行相当多的代码更改，开发者不会轻易升级依赖[103,104]。此外，也有研究工作提出量度对基于安装依赖的供应链上依赖的过时程度进行度量[105-109]。Zerouali 等人[106]提出的两种度量供应链依赖过时程度的量度被广泛使用：

- 滞后时间：一个软件包的依赖的版本约束允许的最新版本与包托管平台上该依赖的最新版本的发布时间之差。
- 滞后版本：一个软件包的依赖的版本约束允许的最新版本与包托管平台上该依赖的最新版本的版本号之差。

已有研究[98,99]通过对开发者进行问卷调查的方式，发现开发者不更新依赖主要有 4 个原因：项目使用旧的依赖版本也能正常运行、害怕出现不兼容、不了解依赖的第三方库发生更新和更新的成本过高。比如 Derr 等人[98]分析了 89 个安卓第三方库的 1971 个版本，发现 58%的版本变化不符合 Semantic Versioning 规范，因此应用开发者无法正确评估升级库的预期成本，选择继续使用过时的版本；Huang 等人[101,102]同样发现 35.3%修复安全 bug 的库版本中删除了超过 300 个 API，为库的更新带来很大的成本与风险.

针对上述开发者不更新依赖的原因（例如不了解依赖的第三方库出现更新，害怕出现不兼容），一些自动化工具被提出来帮助解决上述问题。比如 David-DM 会自动检查一个项目的依赖是否过时，如果过时了就会展示红色的徽章提醒开发者。greenkeeper 会自动检查项目的依赖声明文件中的依赖是否有更新，如果有更新可用，它会在项目内创建一个 branch，在这个分支上进行更新，并运行 CI 测试，如果 CI 测试通过，它会提交一个更新依赖的 PR；如果 CI 测试不通过，它会在项目内提交一个 issue 报告，通知开发者哪个依赖的更新没有通过测试。研究发现这些自动化工具的确可以提高开发者更新依赖的频率[110]，但是其有效性仍然需要改进，比如只有 1/3 的 greenkeeper 创建的 PR 会被合并[110]，很多 greenkeeper 创建的 issue 占所有 issue 数量的一半，且很多 issue 实际上并不是因为依赖更新造成的[111]。

5.3 依赖迁移（库迁移）

形式化地看，依赖迁移是图中某下游节点移除与上游节点之间的边并与其他类似节点建立新的边的过程。复用第三方库在软件开发中非常常见。随着下游软件的演化，其对上游第三方库的需求可能会发生改变，同时第三方库在演化过程中也可能出现不被维护、出现安全漏洞等问题。这时，下游软件的开发者会选择迁移到另一个库，即移除对现在的第三方库的依赖，然后用另一个第三方库替代，这种现象被称为*依赖迁移*，在很多文献中也被称为*库迁移*。库迁移在软件维护中很常见，但是这个过程通常是由开发者通过网络搜索等方式收集多方面的信息然后分析和讨论来决定的，耗时耗力。根据库迁移的流程，本文从迁移原因、迁移目标库推荐、迁移 API 映射三方面对相关工作进行总结。

研究人员主要通过挖掘版本开发历史如 commit、issue 和 PR 来找到开发者公开讨论的进行库迁移的原因。Kabinna 等人[112]分析了 223 个 ASF 项目的 JIRA 中的 issue 报告，总结了它们进行日志库迁移的 5 个原因：可以在一个项目里使用不同的日志库、改善性能、减少代码维护的成本、降低对其他库的依赖和使用目标日志库的新特性。He 等人[113]总结了开发者进行库迁移（不仅仅是库迁移）的 14 个原因，其中被弃用的库不被维护、目标库的特性和可用性更好、为了与项目更好地集成和简化依赖是开发者提及的最频繁的原因。

开发者在决定进行库迁移后，需要决定迁移到哪个库。一些工作构建自动化工具为开发者推荐迁移的目标库。这些工具首先通过分析依赖配置文件[114,115]或 API 调用[116]的方式识别出项目依赖的第三方库，然后挖掘项目的历史，获取项目的依赖变更历史，最后设计算法如关联规则挖掘和排序[115]来识别迁移规则。如 He 等人[115]从依赖配置文件 POM.xml 中识别项目的依赖，然后挖掘项目的依赖变更历史，识别出候选的迁移目标库，最后设计了 4 个量度对候选库进行排序。这些库迁移推荐工具都依赖于实际发生的库迁移，因此都不可避免地受限于冷启动问题。利用这些工具，研究者还分析了库迁移发生的领域和迁移趋势，发现库迁移主要发生在日志、JSON、测试和网络服务等几个领域，并呈现出非常高的单向性，即一个库要么被大多数项目弃用，要么被大多数项目选择[113]。

在确定迁移到的目标库后，还需要进行功能的替换，即找到迁移库和目标库之间的 API 映射。这些工具基于发生库迁移的项目中相关的代码修改片段识别迁移 API 映射。Teyton 等人[117]从同一个 commit 的同一个代码修改片段中增加和删除的 API，构建迁移的库和目标库之间的 API 映射，然后通过人工检查的方式确定最终的映射关系。Alrubaye 等人[118]进一步提出一个自动化挖掘迁移 API 映射的方法。给定一个被迁移的库，该方法首先挖掘项目的提交历史，识别出开发者修改调用它的 API 的 commit 集合，然后提取所有包含移除方法和增加方法的代码更改片段，最后结合 API 的签名和 API 文档的相似性，生成要迁移的库和目标库之间的 API 映射。Chen 等人[119]提出一种无监督的深度学习方法。该方法通过学习 API 使用和 API 描述（API 名字和文档）的嵌入模型，来推断迁移库与目标库之间相似 API 的映射。此方法只能产生一对一的 API 映射，然而实际进行库迁移的时候，迁移库和目标库之间的 API 映射可能是多对一或一对多的，甚至是多对多的。

5.4 其他

除了从上述六个方面研究软件供应链中下游软件制品的依赖管理外，研究者还从变更移植、依赖选择、许可证合规和依赖降级等角度分析下游项目的依赖管理。Ray 等人[120]分析了 FreeBSD，NetBSD，和 OpenBSD 三个项目间的变更移植行为，发现这三个项目的每个版本中约 10%-15%的代码行数是移植自另外两个项目的，变更移植是定期发生的且主要由一小部分开发者完成。同时移植的变更比非移植的变更出现缺陷的可能性更低，这表明开发者更有可能选择其他项目中经过良好测试的变更进行移植。Larios 等人[121]对来自 11 个不同行业的 16 名开发人员进行访谈，并对 115 名参与选择第三方库的开发者进行调查，来分析产业界开发者选择第三方库时考虑的因素。基于访谈和调查结果，他们总结了一套包含技术、人力和经济三个方面的 26 个因素供开发者在选择第三方库时参考。Wu 等人[122]总结了开发者复用 SO 代码的三大障碍：需要太多的代码修改才能在项目里运行、代码的实现不全面和代码质量低；Baltes 等人[123]研究了开发者在使用 Stack Overflow (SO) 上代码时是否遵守了 CC BY-SA 系列许可证的要求，即注明出处和许可证兼容，他们估计 GitHub 上最多只有

1/4 的对 SO 上 Java 代码片段的使用注明了出处，且大多数注明 SO 代码片段出处的方式并不符合 CC BY-SA 系列许可证的要求。Cogo 等人[124]发现开发者进行依赖降级要么是为了处理上游版本的问题如软件缺陷、意外的特性变更或与其他依赖不兼容，要么是主动降级以避免上游依赖未来版本潜在的问题。

5.5 小结

综上所述，相关工作主要从依赖冲突、依赖更新和依赖迁移三个方面对基于安装依赖的软件供应链中下游软件制品的依赖管理问题进行研究。这些工作揭示了下游软件制品在管理依赖时面临着问题多样性、对上游缺乏了解和管理成本高等挑战，并且这些挑战会随着软件包的增加和演化进一步加剧。尽管已经有一些工具帮助开发者应对上述挑战，但目前仍存在两方面的不足。一方面，目前的工具涵盖的编程语言和问题类型较少，比如同一依赖不同版本约束的冲突被广泛研究，而其他类型的依赖冲突研究则较少，再比如 Java 语言的依赖管理（尤其是依赖更新和依赖迁移）问题被广泛研究，其他编程语言的依赖管理问题很少被研究；另一方面，工具的可用性还有待进一步提高，比如 greenkeeper 会产生大量依赖更新失败的误报，给开发者带来额外的负担。

6 开源软件供应链研究的挑战与展望

开源软件供应链已成为软件行业的基石，具有深刻的复杂性、动态性和低可见性。现有文献从结构与演化、风险传播与管理，以及依赖管理三个方面，从整体到局部、由泛入微地对各种类型的开源软件供应链展开了广泛的研究。本文分析与总结相关文献，提出下述挑战。

6.1 开源软件供应链的构建及其数据基础设施

开源软件供应链的构建是控制和管理开源软件供应链的基础。厘清开源软件供应链的结构，一方面需要形成系统性的软件供应链构建理论，另一方面需要相关开源大数据基础设施的支持。

关于软件供应链的构建，本文将软件供应链抽象为一个图，图中的节点代表软件制品，边代表代码复用。我们总结了 4 种典型的软件制品类型，包括软件包、代码仓库、二进制应用和代码片段，以及 5 种典型的代码复用方式，包括安装依赖、API 调用、fork、文件拷贝和代码克隆。这为软件供应链的构建提供了模型基础。未来工作可以针对不同制品类型和代码复用关系，或者定位不同领域，进行供应链构建，并以此为基础研究开源软件供应链的基本性质和规律。

关于开源大数据基础设施，目前比较流行的数据基础设施有 Libraries.io、GitHub Dependency Graph、Open Source Insights 和 World of Code，前三个基于安装依赖构建，World of Code 提供了代码文件的修改历史与项目的交叉引用以及代码文件与导入的软件包的映射，便于研究者构建基于文件拷贝和 API 调用的代码复用形成的供应链。未来工作需要对这些数据基础设施中的数据内容进行交叉验证，保障数据的正确性和全面性，同时构建数据基础设施间数据实体的映射，便于结合多个数据基础设施开展供应链的研究；另一方面，需要保障数据基础设施的及时性和高效更新，提高其可用性。此外，面向二进制应用的供应链和面向代码克隆的供应链的数据基础设施仍是空白，这需要大规模高效的二进制应用分析技术和代码克隆检测技术的支持。

6.2 软件供应链的风险评估与控制

软件供应链中上游软件制品的风险（如 API 变更和软件缺陷）可以沿着供应链广泛传播，对软件供应链造成巨大影响。为了降低软件供应链中的风险传播的影响，未来工作需要研究既系统又精细的风险评估与控制机制。对于上游的 API 变更来说，需要研究如何提高上游软件制品对下游软件制品复用情况的感知能力，比如 API 调用频率和调用方式等。这可以帮助上游了解引入 API 变更对下游造成的影响，做出对下游影响最小的 API 变更方式，例如影响的下游项目数量较少、下游项目升级依赖版本时需要进行的修改最少、甚至可以通过自动化升级工具实现平稳升级等。对于软件缺陷来说，需要研究：如何保障链上高风险节点（如被直接和间接依赖最多的软件制品）的安全，从根源上避免漏洞的产生；另一方面，在漏洞发现后，从可达性和可触发性的角度精确找到受漏洞影响的下游软件制品，并及时把解决方案推送给下游。最后，优化软件供应链

的结构，控制节点在供应链的影响范围，也是一种有效手段，比如降低上游软件制品的不可替代性等。

6.3 面向开源软件供应链的依赖管理

通过各种形式的代码复用以及直接和间接代码复用，一个软件制品包含对大量其他软件制品的依赖，为依赖管理带来了困难。已有研究发现供应链上软件制品的依赖管理面临着问题多样化、对上游了解不充分、管理成本高等挑战，并且这些挑战还会随着软件制品的增加和演化而加剧。因此，需要研发更好的面向开源软件供应链依赖管理的自动化工具来帮助开发者应对这些挑战。

针对问题多样化的挑战，尽管相关工作已经提出一些工具自动化检测和修复这些问题，但是这些工具主要是针对 Java 项目，对于其他编程语言的项目则研究较少。考虑到不同编程语言遵循不同的设计模式和开发实践，不同语言的供应链上面临的依赖管理问题和对应的解决方案也会有所差异。有必要对其他编程语言的供应链上的依赖管理问题进行广泛分析，优化其依赖管理机制，并寻找通用的解决方案。

关于对上游了解不充分和管理成本高的挑战，首先需要结合相关数据基础设施，从不同代码复用方式的角度检测软件制品中的上游成分，然后设计自动化工具监控上游的演化，评估上游演化对下游软件制品的影响，然后及时报告给下游开发者。已有一些工具可能囿于项目本身数据质量的问题，可能会产生大量误报，为开发者带来额外的负担。因此，提高工具的可用性也是需要解决的重要问题。

7 相关工作

随着面向软件供应链的研究越来越多，一些研究者尝试对相关文献进行梳理、总结，以促进进一步的软件供应链研究。这些综述主要从软件供应链安全的角度展开。360 威胁情报中心的研究人员基于实际的软件供应链安全事件，从软件生命周期的角度将软件供应链抽象为一个包含开发、交付和使用等三个环节的模型，被研究者广泛使用。Zhou[10]基于典型的软件供应链安全事件，总结了三个环节的攻击向量，构建软件供应链威胁模型。He 等人[8]总结了软件供应链各环节风险防范技术的研究现状，然后从基本规范、风险评价和安全标准三个方面介绍了目前的供应链风险管理手段。Ji 等人[9]考虑到软件中越来越多地使用开源组件，在三环节模型的基础上，增加了组件开发环节，并基于近 10 年的开源软件供应链攻击事件分析各环节的攻击面，然后从风险识别和加固防御两个方面总结研究现状。Wu 等人[11]介绍了静态分析、动态分析、符号执行和污点分析等程序分析技术在软件供应链攻击检测任务上的应用，并分析了现有技术在软件供应链攻击检测任务上的不足与挑战。Ohm 等人[125]分析了 npm，PyPI 和 RubyGems 等三个软件包托管平台上 174 个被用来实施开源软件供应链攻击的恶意软件包，总结了这些恶意软件包的攻击方式和恶意行为。

可以看出，现有综述主要关注软件供应链的安全，抽象出基于软件生命周期的软件供应链模型，以此为基础总结研究现状。然而，这些综述忽视了软件供应链的整体结构和链上软件制品之间的相互影响。本文提出基于图的开源软件供应链模型，从结构与演化、风险传播与管理和依赖管理三个方面总结研究现状，提供一个了解软件供应链复杂性、动态性和软件制品间相互影响的整体视角，弥补了现有软件供应链综述的不足。

8 总　结

开源软件供应链在现代软件开发，乃至现代社会中，扮演着越来越重要的角色。开源软件供应链牵一发而动全身的特点亟需我们从整体性视角去看待和分析。本文把开源软件供应链定义为各种软件制品之间通过各种代码复用方式形成的供应关系网络，并进一步总结 4 种典型的软件制品和 5 种典型的代码复用方式。基于这些定义和实例，本文对相关文献进行收集和总结，从结构与演化、风险传播与管理和依赖管理三个方面总结相关文献，并展望了开源软件供应链的研究挑战与未来研究方向。我们希望本文工作可以帮助人们增加对开源软件供应链的认识，促进未来的开源软件供应链研究。

References:

[1] Zhou M H, Zhang W, Yin G. Qualitative analysis of open source software (in Chinese). Communications of The CCF,2016, 12: 24-29.

[2] GitHub. Supporting sustainable communities | The State of the Octobers'. https://octoverse.github.com/sustainable-communities/

[3] Stack Exchange. Stack Exchange Data Explorer. https://data.stackexchange.com/

[4] Synopsys. 2022 Open Source Security and Analysis Report. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

[5] Zhou M H, Zhang Y X, Tan X. Software digital sociology (in Chinese). Sci Sin Inform, 2019, 49: 1399–1411, do: 10.1360/N112018-00319.

[6] Quartz. How one programmer broke the internet by deleting a tiny piece of code. https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code/

[7] Log4j. Apache Log4j Security Vulnerabilities. https://logging.apache.org/log4j/2.x/security.html

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

[9] 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, (in Chinese). http://www.jos.org.cn/1000-9825/6717.htm

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

[11] 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.

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

[13] Wikipedia contributors. Supply chain. https://en.wikipedia.org/w/index.php?title=Supply_chain&oldid=1111876757

[14] Holdsworth J. Software Process Design[M]. McGraw-Hill, Inc., 1995.

[15] Farbey B, Finkelstein A. Exploiting software supply chain business architecture: a research agenda[J]. 1999.

[16] Keele S. Guidelines for performing systematic literature reviews in software engineering[R]. Technical report, ver. 2.3 ebse technical report. ebse, 2007.

[17] Wohlin C. Guidelines for snowballing in systematic literature studies and a replication in software engineering[C]//Proceedings of the 18th international conference on evaluation and assessment in software engineering. 2014: 1-10.

[18] Ma Y, Bogart C, Amreen S, et al. World of code: an infrastructure for mining the universe of open source VCS data[C]//2019 IEEE/ACM 16th international conference on mining software repositories (MSR). IEEE, 2019: 143-154.

[19] Tidelift. Libraries.io. https://libraries.io/.

[20] Wittern E, Suter P, Rajagopalan S. A look at the dynamics of the JavaScript package ecosystem[C]//Proceedings of the 13th International Conference on Mining Software Repositories. 2016: 351-361.

[21] Kikas R, Gousios G, Dumas M, et al. Structure and evolution of package dependency networks[C]//2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, 2017: 102-112.

[22] Decan A, Mens T, Claes M. An empirical comparison of dependency issues in OSS packaging ecosystems[C]//2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER). IEEE, 2017: 2-12.

[23] Decan A, Mens T, Grosjean P. An empirical comparison of dependency network evolution in seven software packaging ecosystems[J]. Empirical Software Engineering, 2019, 24(1): 381-416.

[24] Decan A, Mens T, Claes M. On the topology of package dependency networks: A comparison of three programming language ecosystems[C]//Proccedings of the 10th European Conference on Software Architecture Workshops. 2016: 1-4.

[25] Soto-Valero C, Benelallam A, Harrand N, et al. The emergence of software diversity in maven central[C]//2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, 2019: 333-343.

[26] Zimmermann M, Staicu C A, Tenny C, et al. Small world with high risks: A study of security threats in the npm ecosystem[C]//28th USENIX Security Symposium (USENIX Security 19). 2019: 995-1010.

[27] MacDonald Fiona. How a programmer nearly broke the internet by deleting just 11 lines of code. https:// www.sciencealert.com/how-a-programmer-almost-broke-the-internet-by-deleting-11-lines-of-code.

[28] Abdalkareem R, Nourry O, Wehaibi S, et al. Why do developers use trivial packages? an empirical case study on npm[C]//Proceedings of the 2017 11th joint meeting on foundations of software engineering. 2017: 385-395.
[29] Abdalkareem R, Oda V, Mujahid S, et al. On the impact of using trivial packages: an empirical case study on npm and PyPI[J]. Empirical Software Engineering, 2020, 25(2): 1168-1204.
[30] Chen X, Abdalkareem R, Mujahid S, et al. Helping or not helping? Why and how trivial packages impact the npm ecosystem[J]. Empirical Software Engineering, 2021, 26(2): 1-24.
[31] Chowdhury M A R, Abdalkareem R, Shihab E, et al. On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages[J]. IEEE Transactions on Software Engineering, 2021.
[32] Valiev M, Vasilescu B, Herbsleb J. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem[C]//Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018: 644-655.
[33] Dey T, Mockus A. Are software dependency supply chain metrics useful in predicting change of popularity of npm packages?[C]//Proceedings of the 14th international conference on predictive models and data analytics in software engineering. 2018: 66-69.
[34] Dey T, Ma Y, Mockus A. Patterns of effort contribution and demand and user classification based on participation patterns in npm ecosystem[C]//Proceedings of the fifteenth international conference on predictive models and data analytics in software engineering. 2019: 36-45.
[35] Hejderup J, Beller M, Triantafyllou K, et al. Präzi: from package-based to call-based dependency networks[J]. Empirical Software Engineering, 2022, 27(5): 1-42.
[36] Harrand N, Benelallam A, Soto-Valero C, et al. API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages[J]. Journal of Systems and Software, 2022, 184: 111134.
[37] Ma Y, Mockus A, Zaretzki R, et al. A Methodology for Analyzing Uptake of Software Technologies Among Developers[J]. IEEE Transactions on Software Engineering, 2020.
[38] Tan X, Gao K, Zhou M, et al. An exploratory study of deep learning supply chain[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 86-98.
[39] Lopes C V, Maj P, Martins P, et al. DéjàVu: a map of code duplicates on GitHub[J]. Proceedings of the ACM on Programming Languages, 2017, 1(OOPSLA): 1-28.
[40] Hata H, Kula R G, Ishio T, et al. Same File, Different Changes: The Potential of Meta-Maintenance on GitHub[C]//2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021: 773-784.
[41] Wyss E, De Carli L, Davidson D. What the fork? finding hidden code clones in npm[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 2415-2426.
[42] Imam A, Dey T, Nolte A, et al. The Secret Life of Hackathon Code Where does it come from and where does it go?[C]//2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). IEEE, 2021: 68-79.
[43] Mahmoud A S I, Dey T, Nolte A, et al. One-off events? An empirical study of hackathon code creation and reuse[J]. Empirical Software Engineering, 2022, 27(7): 1-49.
[44] Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code[J]. IEEE Transactions on Software Engineering, 2002, 28(7): 654-670.
[45] Jiang L, Misherghi G, Su Z, et al. Deckard: Scalable and accurate tree-based detection of code clones[C]//29th International Conference on Software Engineering (ICSE'07). IEEE, 2007: 96-105.
[46] Sajnani H, Saini V, Svajlenko J, et al. Sourcerercc: Scaling code clone detection to big-code[C]//Proceedings of the 38th International Conference on Software Engineering. 2016: 1157-1168.
[47] Abdalkareem R, Shihab E, Rilling J. On code reuse from stackoverflow: An exploratory study on android apps[J]. Information and Software Technology, 2017, 88: 148-158.
[48] Gharehyazie M, Ray B, Filkov V. Some from here, some from there: Cross-project code reuse in github[C]//2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, 2017: 291-301.

[49] Gharehyazie M, Ray B, Keshani M, et al. Cross-project code clones in github[J]. Empirical Software Engineering, 2019, 24(3): 1538-1573.

[50] Wu W, Khomh F, Adams B, et al. An exploratory study of api changes and usages based on apache and eclipse ecosystems[J]. Empirical Software Engineering, 2016, 21(6): 2366-2412.

[51] Xavier L, Brito A, Hora A, et al. Historical and impact analysis of API breaking changes: A large-scale study[C]//2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2017: 138-147.

[52] Raemaekers S, van Deursen A, Visser J. Semantic versioning and impact of breaking changes in the Maven repository[J]. Journal of Systems and Software, 2017, 129: 140-158.

[53] Semantic Versioning. https://semver.org.

[54] K¨uhne L, Massol V, Kitching S. The Clirr Maven Plugin. https://www.mojohaus.org/clirr-maven-plugin/.

[55] Jezek K, Dietrich J. API Evolution and Compatibility: A Data Corpus and Tool Evaluation[J]. J. Object Technol., 2017, 16(4): 2:1-23.

[56] Ochoa L, Degueule T, Falleri J R, et al. Breaking bad? Semantic versioning and impact of breaking changes in Maven Central[J]. Empirical Software Engineering, 2022, 27(3): 1-42.

[57] Brito A, Valente M T, Xavier L, et al. You broke my code: understanding the motivations for breaking changes in APIs[J]. Empirical Software Engineering, 2020, 25(2): 1458-1492.

[58] Bogart C, Kästner C, Herbsleb J, et al. How to break an API: cost negotiation and community values in three software ecosystems[C]//Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016: 109-120.

[59] Bogart C, Kästner C, Herbsleb J, et al. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2021, 30(4): 1-56.

[60] Robbes R, Lungu M, Röthlisberger D. How do developers react to API deprecation? The case of a Smalltalk ecosystem[C]//Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. 2012: 1-11.

[61] Sawant A A, Robbes R, Bacchelli A. On the reaction to deprecation of 25,357 clients of 4+ 1 popular Java APIs[C]//2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2016: 400-410.

[62] Sawant A A, Robbes R, Bacchelli A. On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK[J]. Empirical Software Engineering, 2018, 23(4): 2158-2197.

[63] Sawant A A, Robbes R, Bacchelli A. To react, or not to react: Patterns of reaction to API deprecation[J]. Empirical Software Engineering, 2019, 24(6): 3824-3870.

[64] Hora A, Robbes R, Anquetil N, et al. How do developers react to api evolution? the pharo ecosystem case[C]//2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2015: 251-260.

[65] Decan A, Mens T, Constantinou E. On the impact of security vulnerabilities in the npm package dependency network[C]//Proceedings of the 15th international conference on mining software repositories. 2018: 181-191.

[66] Liu C, Chen S, Fan L, et al. Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem[C]// Proceedings of the 44th International Conference on Software Engineering. 2022: 672-684.

[67] Zerouali A, Mens T, Decan A, et al. On the impact of security vulnerabilities in the npm and RubyGems dependency networks[J]. Empirical Software Engineering, 2022, 27(5): 1-45.

[68] Massacci F, Pashchenko I. Technical leverage in a software ecosystem: Development opportunities and security risks[C]//2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021: 1386-1397.

[69] Xu M, Wang Y, Cheung S C, et al. Insight: Exploring Cross-Ecosystem Vulnerability Impacts[C]//37th IEEE/ACM International Conference on Automated Software Engineering. 2022: 1-13.

[70] Latendresse J, Mujahid S, Costa D E, et al. Not All Dependencies are Equal: An Empirical Study on Production Dependencies in NPM[C]//37th IEEE/ACM International Conference on Automated Software Engineering. 2022: 1-12.

[71] Pashchenko I, Plate H, Ponta S E, et al. Vulnerable open source dependencies: Counting those that matter[C]//Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. 2018: 1-10.

[72] Ma W, Chen L, Zhang X, et al. Impact analysis of cross-project bugs on software ecosystems[C]//2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). IEEE, 2020: 100-111.

[73] Reid D, Jahanshahi M, Mockus A. The Extent of Orphan Vulnerabilities from Code Reuse in Open Source Software[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 2104-2115.

[74] Duan R, Alrawi O, Kasturi R P, et al. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages[C].// Network and Distributed Systems Security (NDSS) Symposium, 2021.

[75] Vu D L, Massacci F, Pashchenko I, et al. Lastpymile: identifying the discrepancy between sources and packages[C]//Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021: 780-792.

[76] Gu Y, Ying L, Pu Y, et al. Investigating Package Related Security Threats in Software Registries[C]//2023 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 2023: 1151-1168.

[77] Ladisa P, Plate H, Martinez M, et al. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains[C]//2023 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 2023: 167-184.

[78] Alfadel M, Costa D E, Shihab E. Empirical analysis of security vulnerabilities in python packages[C]//2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2021: 446-457.

[79] Prana G A A, Sharma A, Shar L K, et al. Out of sight, out of mind? How vulnerable dependencies affect open-source projects[J]. Empirical Software Engineering, 2021, 26(4): 1-34.

[80] Zhang Z, Zhang H, Qian Z, et al. An investigation of the android kernel patch ecosystem[C]//30th USENIX Security Symposium (USENIX Security 21). 2021: 3649-3666.

[81] Hou Q, Diao W, Wang Y, et al. Large-scale Security Measurements on the Android Firmware Ecosystem[C] //Proceedings of the 44th International Conference on Software Engineering. 2022: 1257-1268.

[82] Chinthanet B, Kula R G, McIntosh S, et al. Lags in the release, adoption, and propagation of npm vulnerability fixes[J]. Empirical Software Engineering, 2021, 26(3): 1-28.

[83] Ma W, Chen L, Zhang X, et al. How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem[C]//2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017: 381-392.

[84] Lin J, Zhang H, Adams B, et al. Upstream bug management in Linux distributions[J]. Empirical Software Engineering, 2022, 27(6): 1-41.

[85] Artho C, Suzaki K, Di Cosmo R, et al. Why do software packages conflict?[C]//2012 9th IEEE Working Conference on Mining Software Repositories (MSR). IEEE, 2012: 141-150.

[86] Wang Y, Wen M, Liu Z, et al. Do the dependency conflicts in my project matter?[C]//Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. 2018: 319-330.

[87] Huang K, Chen B, Shi B, et al. Interactive, effort-aware library version harmonization[C]//Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020: 518-529.

[88] Wang Y, Wen M, Liu Y, et al. Watchman: Monitoring dependency conflicts for python library ecosystem[C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020: 125-135.

[89] Li Z, Wang Y, Lin Z, et al. Nufix: escape from NuGet dependency maze[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1545-1557.

[90] Jia Z, Li S, Yu T, et al. DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures[C]//2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021: 86-98.

[91] Wang Y, Wu R, Wang C, et al. Will Dependency Conflicts Affect My Program's Semantics?[J]. IEEE Transactions on Software Engineering, 2021, 48(7): 2295-2316.

[92] Patra J, Dixit P N, Pradel M. Conflictjs: finding and understanding conflicts between javascript libraries[C]//Proceedings of the 40th International Conference on Software Engineering. 2018: 741-751.

[93] Wang Y, Qiao L, Xu C, et al. Hero: On the Chaos When PATH Meets Modules[C]//2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021: 99-111.

[94] Abate P, Di Cosmo R, Gesbert L, et al. Mining component repositories for installability issues[C]//2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, 2015: 24-33.

[95] Wang Y, Wen M, Wu R, et al. Could i have a stack trace to examine the dependency conflict issue?[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 572-583.

[96] Cox J, Bouwers E, Van Eekelen M, et al. Measuring dependency freshness in software systems[C]//2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, 2015, 2: 109-118.

[97] McDonnell T, Ray B, Kim M. An empirical study of api stability and adoption in the android ecosystem[C]//2013 IEEE International Conference on Software Maintenance. IEEE, 2013: 70-79.

[98] Derr E, Bugiel S, Fahl S, et al. Keep me updated: An empirical study of third-party library updatability on android[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2187-2200.

[99] Salza P, Palomba F, Di Nucci D, et al. Do developers update third-party libraries in mobile apps?[C]//Proceedings of the 26th Conference on Program Comprehension. 2018: 255-265.

[100] Kula R G, German D M, Ouni A, et al. Do developers update their library dependencies?[J]. Empirical Software Engineering, 2018, 23(1): 384-417.

[101] Wang Y, Chen B, Huang K, et al. An empirical study of usages, updates and risks of third-party libraries in java projects[C]//2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2020: 35-45.

[102] Huang K, Chen B, Xu C, et al. Characterizing usages, updates and risks of third-party libraries in Java projects[J]. Empirical Software Engineering, 2022, 27(4): 1-41.

[103] Bavota G, Canfora G, Di Penta M, et al. The evolution of project inter-dependencies in a software ecosystem: The case of apache[C]//2013 IEEE international conference on software maintenance. IEEE, 2013: 280-289.

[104] Bavota G, Canfora G, Di Penta M, et al. How the apache community upgrades dependencies: an evolutionary study[J]. Empirical Software Engineering, 2015, 20(5): 1275-1317.

[105] Gonzalez-Barahona J M, Sherwood P, Robles G, et al. Technical lag in software compilations: Measuring how outdated a software deployment is[C]//IFIP International Conference on Open Source Systems. Springer, Cham, 2017: 182-192.

[106] Zerouali A, Constantinou E, Mens T, et al. An empirical analysis of technical lag in npm package dependencies[C]//International Conference on Software Reuse. Springer, Cham, 2018: 95-110.

[107] Decan A, Mens T, Constantinou E. On the evolution of technical lag in the npm package dependency network[C]//2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018: 404-414.

[108] Zerouali A, Mens T, Gonzalez‐Barahona J, et al. A formal framework for measuring technical lag in component repositories—and its application to npm[J]. Journal of Software: Evolution and Process, 2019, 31(8): e2157.

[109] Zerouali A, Mens T, Decan A, et al. A multi-dimensional analysis of technical lag in Debian-based Docker images[J]. Empirical Software Engineering, 2021, 26(2): 1-45.

[110] Mirhosseini S, Parnin C. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?[C]//2017 32nd IEEE/ACM international conference on automated software engineering (ASE). IEEE, 2017: 84-94.

[111] Rombaut B, Cogo F R, Adams B, et al. There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the Greenkeeper dependency bot in npm[J]. ACM Transactions on Software Engineering and Methodology, 2022.

[112] Kabinna S, Bezemer C P, Shang W, et al. Logging library migrations: A case study for the apache software foundation projects[C]//2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR). IEEE, 2016: 154-164.

[113] He H, He R, Gu H, et al. A large-scale empirical study on Java library migrations: prevalence, trends, and rationales[C]//Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021: 478-490.

[114] Teyton C, Falleri J R, Blanc X. Mining library migration graphs[C]//2012 19th Working Conference on Reverse Engineering. IEEE, 2012: 289-298.

[115] He H, Xu Y, Ma Y, et al. A multi-metric ranking approach for library migration recommendations[C]//2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2021: 72-83.

[116] Teyton C, Falleri J R, Palyart M, et al. A study of library migrations in java[J]. Journal of Software: Evolution and Process, 2014, 26(11): 1030-1052.

[117] Teyton C, Falleri J R, Blanc X. Automatic discovery of function mappings between similar libraries[C]//2013 20th Working Conference on Reverse Engineering (WCRE). IEEE, 2013: 192-201.

[118] Alrubaye H, Mkaouer M W, Ouni A. On the use of information retrieval to automate the detection of third-party java library migration at the method level[C]//2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). IEEE, 2019: 347-357.

[119] Chen C, Xing Z, Liu Y, et al. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding[J]. IEEE Transactions on Software Engineering, 2019, 47(3): 432-447.

[120] Ray B, Kim M. A case study of cross-system porting in forked projects[C]//Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. 2012: 1-11.

[121] Larios Vargas E, Aniche M, Treude C, et al. Selecting third-party libraries: The practitioners' perspective[C]//Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. 2020: 245-256.

[122] Wu Y, Wang S, Bezemer C P, et al. How do developers utilize source code from stack overflow?[J]. Empirical Software Engineering, 2019, 24(2): 637-673.

[123] Baltes S, Diehl S. Usage and attribution of Stack Overflow code snippets in GitHub projects[J]. Empirical Software Engineering, 2019, 24(3): 1259-1295.

[124] Cogo F R, Oliva G A, Hassan A E. An empirical study of dependency downgrades in the npm ecosystem[J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2457-2470.

[125] Ohm M, Plate H, Sykosch A, et al. Backstabber's knife collection: A review of open source software supply chain attacks[C]//Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17. Springer International Publishing, 2020: 23-43.

附中文参考文献:

[1] 周明辉, 张伟, 尹刚. 开源软件的量化分析. 中国计算机学会通讯, 2016, 12: 24-29

[5] 周明辉, 张宇霞, 谭鑫. 软件数字社会学. 中国科学: 信息科学, 2019, 49: 1399 – 1411, doi: 10.1360/N112018-00319

[8] 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.

[9] 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. http://www.jos.org.cn/1000-9825/6717.htm

[10] 周振飞.软件供应链污染机理与防御研究.北京邮电大学, 2018.

[11] 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, 40(1):103-115.

附本课题组研究成果:

[1] Zhou M H, Zhang W, Yin G. Qualitative analysis of open source software (in Chinese). Communications of The CCF,2016, 12: 24-29.

[5] Zhou M H, Zhang Y X, Tan X. Software digital sociology (in Chinese). Sci Sin Inform, 2019, 49: 1399–1411, do: 10.1360/N112018-00319.

[38] Tan X, Gao K, Zhou M, et al. An exploratory study of deep learning supply chain[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 86-98.

[113] He H, He R, Gu H, et al. A large-scale empirical study on Java library migrations: prevalence, trends, and rationales[C]// Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021: 478-490.

[115] He H, Xu Y, Ma Y, et al. A multi-metric ranking approach for library migration recommendations[C]//2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2021: 72-83.