NPM 中语义化版本控制的大规模分析

Donald Pinckney Federico Cassano Arjun Guha Jonathan Bell东北大学 东北大学 东北大学和 *Roblox* 东北大学波士顿，美国 波士顿，美国 波士顿，美国 pinckney.d@northeastern.edu cassano.f@northeastern.edu a.guha@northeastern.edu j.bell@northeastern.edu

arXiv:2304.00394v1 [cs.SE] 1 Apr 2023

摘要——**NPM 软件包仓库包含超过两百万个软件包，每周下载量达数百亿次。几乎所有 JavaScript 应用程序都会使用 NPM 软件包管理器从 NPM 软件包仓库安装软件包。NPM 依靠“语义化版本控制”（“semver”）方案来维护健康的生态系统：漏洞修复能够尽快、可靠地交付给下游软件包，而破坏性变更则需要下游软件包维护者进行人工干预。为了了解开发者如何使用 semver，我们构建了一个数据集，其中包含 NPM 上每个软件包的每个版本，并分析整个生态系统中的更新流动情况。我们为 NPM 构建了一个时间旅行依赖解析器，从而能够准确确定在不同时间解析出的每个依赖项的版本。我们对分析进行分段，以便直接比较与安全性相关的更新（即引入漏洞或修复漏洞的更新）和生态系统中的其他更新。我们发现，当开发者正确使用 semver 时，安全补丁等关键更新在大多数情况下（90.09%）都能相当迅速地传递至下游依赖，但这种情况并不总是发生，原因在于开发者对 semver 版本约束和 semver 版本号递增规则的使用并不完善。我们的研究结果对开发者和研究人员都具有重要意义。我们依据开源许可证公开提供基础设施和数据集。**

索引术语—**NPM、依赖项管理、JavaScript**

I. 引言

现代软件开发与大规模的开源软件包代码仓库密不可分。例如，NPM 代码仓库包含超过两百万个软件包，每周下载量达数百亿次；实际上，几乎每个 JavaScript 应用程序都使用 NPM 软件包管理器从 NPM 代码仓库安装软件包。随着开源软件包代码仓库规模不断扩大，软件包的维护、更新和分发为恶意行为者提供了日益扩大的攻击面，而了解软件供应链的特性至关重要。

开源生态系统中一个特别值得关注的问题是软件包经历的技术滞后 [1]–[5]，即依赖项有新更新可用与实际应用该更新之间的时间差。NPM 以及其他采用类似设计的生态系统（PyPI 等）通过语义化版本控制（“semver”）和灵活的版本约束提供了一种潜在的解决方案。在 semver 中，版本采用 major.minor.bug 的形式编号，其中 major 表示破坏性 API 变更，minor 表示添加新功能的非破坏性变更，而 bug 表示向后兼容的错误修复[1] [6]。灵活的版本约束允许下游（即依赖其他软件包的）软件包的开发者指定他们愿意自动接受哪些类型的更新。理想情况下，semver 能帮助开发者表达约束和版本号，使非破坏性的重要更新（例如安全补丁）能够快速传递到下游软件包，同时将破坏性变更延后，直到开发者选择接受这些变更。例如，开发者可以指定其依赖软件包 react，并设置约束 ^18.1.1，这意味着允许自动更新至（但不包括）19.0.0 版本。本质上，这一约束表示“接收 React 的所有不太可能造成破坏性变更的更新 ”。

然而，在实践中，语义化版本存在三个重大问题，可能导致技术滞后 [1]–[5]。首先，语义化版本的积极特性取决于两个条件：上游开发者使用正确的语义化版本增量类型标注其更新，以及下游开发者使用既不过于宽松也不过于严格的版本约束。其次，传递依赖链中间的软件包会影响最终接收到的依赖项版本。下游开发者可能列出一个允许使用某软件包最新版本的版本约束，但如果某个传递依赖项具有更严格的版本约束，下游开发者可能无法获得该软件包的最新版本。第三，允许对依赖项进行自动（漏洞）更新可能很危险，因为这会引入恶意软件的攻击向量。

在本研究中，我们旨在了解开发者如何在生态系统规模上使用依赖项、语义化版本控制和灵活的版本约束，以及这些因素如何相互作用，进而影响开发者体验和供应链安全。此前从 NPM 生态系统挖掘数据的研究，主要着眼于回答有关 NPM 在某个时间快照下的问题 [7]–[10]。在本研究中，我们首先通过分析整个 NPM 历史中灵活约束类型频率和语义化版本增量类型频率，了解开发者如何使用语义化版本控制。接下来，为了了解

本研究部分得到了美国国家科学基金会资助项目 CCF-2102288、CCF-2100037 和 CNS-2100015 的支持。

[1]本文使用“漏洞”而非标准的“补丁”语义化版本术语，以便与安全补丁这一概念区分开来。

更新在实际生态系统规模下如何流转，我们开展了大规模实验，在不同时间快照下解析软件包的依赖项，并观察下游软件包需要多久才能收到更新。为开展这些实验，我们构建了一款工具，可在整个 NPM 历史中准确地进行时间穿越式依赖项求解。与此前研究 [2], [4], [11]–[13] 对 NPM 的行为语义进行近似处理不同（这些行为语义并未得到充分明确的规范 [14]），本方法能够更精确地随时间解析依赖项。

我们构建了 NPM 上首个包含以下内容的数据集（截至2022年10月31日）：

1）NPM 上的所有软件包（2,663,681 个软件包）；2）每个软件包的所有版本（28,941,927 个版本）；3）每个软件包所有版本的元数据（压缩后为 ≈ 40 GB）和打包代码（压缩后为 ≈19 TB）；4）从 GitHub 安全公告数据库下载的、针对 NPM 软件包发布的安全公告完整数据。

该数据集已建立索引，便于查询和执行大规模分布式计算。为收集这些数据，我们设计并实现了一个分布式系统，用于从 NPM 下载、归档和检索软件包。我们依据 BSD 三条款许可证[2]发布数据抓取器和数据集。

我们使用该数据集探讨有关 NPM 生态系统的若干问题，尤其是开发者如何使用语义化版本控制，以及这对供应链安全有何影响：

- **RQ1**：开发者是否会指定依赖项版本约束，以支持自动更新？
- **RQ2**：开发者是否会在软件包发布时采用语义化版本控制，以便下游软件包能够自动更新？
- **研究问题3**：软件包是否经常包含过时的依赖项？发布更新后，下游软件包需要多久才能收到这些更新？
- **RQ4**：在各种语义化版本更新中，开发者通常会进行哪些高层次的更改？开发者仅更新依赖项的情况有多常见？

这些结果对开发者和研究人员都具有重要意义。我们表明，总体而言，NPM 生态系统能够有效地分发不引入破坏性变更的更新；但由于需要处理的依赖项数量庞大、更新频繁，大多数软件包最终仍会包含过时的依赖项。此外，我们发现，有些开发者在发布安全补丁时并未以最佳方式使用语义化版本；而次版本和主版本更新似乎更有可能引入安全漏洞。

II. 相关工作

我们的研究问题和方法建立在大量关于语义化版本控制和技术滞后的相关研究之上。

[2]请访问 https://dependencies.science，以获取最新的元数据、tarball 数据和源代码。不含 tarball 数据的原始工件可在 Zenodo [15]上获取。

1）语义化版本控制： 虽然语义化版本控制确实具有精确的语法规范 [6], ，但对于何种情况属于向后兼容，其语义并未得到形式化定义。包括 NPM 在内的工具通常不会强制开发者在实践中遵循语义化版本控制的使用方式。语义化版本控制的使用选择会影响软件包的发布速度、技术滞后、稳定性、开发者挫败感等。Bogart 等人 [16]于 2015 年在 NPM 和 CRAN 生态系统中开展的开发者访谈发现，开发者会尝试使用语义化版本控制，但并不总是了解其影响，而且通常认为依赖管理令人筋疲力尽。更具体地说，Raemaekers 等人 [17] [17] 发现，在 2006—2011年，Maven 开发者经常在所谓的非破坏性语义化版本更新中引入二进制不兼容变更。Wittern 等人 [18] 研究了 NPM 中软件包之间的依赖关系，发现软件包之间的依赖项数量随时间推移不断增加，并观察了 2016年版本约束类型的频率。Dietrich 等人 [19]随后观察了项目级别上版本约束类型频率随时间的变化。在全生态系统级别考察版本约束演变，可以基于“群体智慧”进行评估。Decan 等人 [20]对 Cargo、NPM、Packagist 和 RubyGems 的生态系统级依赖约束进行了分析。我们仅聚焦于单个生态系统（NPM），验证 Decan 等人的研究发现，并对数据集开展更深入的分析。我们的研究还考察了已发布的更新类型的频率，从而能够得出有关安全更新传播的重要结论。

2) 技术滞后： 许多既有研究尝试分析更新向下游软件包的传播情况，以及项目的依赖项通常有多过时。Gonzalez-Barahona 等人 [1]定义了“技术滞后”这一度量，用于分析软件包的依赖项相较于较新发布的版本有多过时；此后，这一度量也在 NPM 的背景下得到了进一步研究 [2], [4], [5]。此外，后续研究还将技术滞后的概念专门用于分析安全补丁的传播或漏洞 [3], [11], [12]。

计算技术滞后很困难，先前的研究曾尝试模拟在不同时间点会解析出的依赖项。有些研究没有考虑传递依赖 [4], [11]，这令人担忧，因为在 NPM 中，传递依赖通常占软件包依赖项的大多数。另一些研究则进一步考虑了传递依赖 [2], [12]。Liu 等人 [13]提出了 *DTResolver*，这是一种自定义的依赖求解算法，比以往方法更贴近 NPM 的行为。然而，作者对 DTResolver 的评估发现，在为 15,673 个库构建依赖树时，它只有 90.58% 的情况下与 NPM 的行为一致 [13]。我们最近对 NPM 依赖解析语义的评估发现了各种边界情况：为统一版本，NPM 的算法会为依赖项选择出乎意料的版本 [14]。尤其是在解析传递依赖时，对 NPM 解析语义的不准确近似所引入的误差

会不断累积。就我们所知，在研究 NPM 生态系统中的技术滞后时，与所有 先前工作相比，我们是唯一 一项使用 NPM 本身来解析历史依赖关系的研究。我们公开了工具和数据集，以便其他人采用这一方法 [15]。

3) NPM 研究：最后，其他研究探讨了 NPM 数据分析中更具体的问题或应用，例如研究开发者何时会降级软件包 [21], 分析 NPM 中热门“微型”软件包的现象 [8]–[10], 以及开发用于理解和防止 NPM 中漏洞或恶意软件的方法 [7], [22]–[24]。我们将在第 VI-C 节中进一步讨论我们的发现如何为未来的研究应用提供指引。

III. 方法

总体而言，我们利用数据集和分析系统的不同方面来回答四个核心研究问题。RQ1 和 RQ2 完全通过分析抓取的元数据来回答。回答研究问题3更具挑战性，因为这需要推理依赖项如何随时间解析；对此，我们在大规模实验中使用时间旅行依赖解析器。最后，为回答 RQ4，我们计算不同软件包版本的 tarball 压缩包之间的差异。

A. RQ1：版本约束使用情况

在 NPM 用于指定依赖项版本约束的丰富语法中，开发者经常使用哪些约束类型，以及这些约束有多宽松或严格，目前尚不清楚。 [14], [25],

我们将版本约束划分为以下互斥类别：

1) 精确约束（"=1.2.3"）只接受明确列出的版本；2) 错误修复灵活约束（"~1.2.3"）接受语义化版本中错误修复部分的任意更新，例如 1.2.4 等；3) 次版本灵活约束（"^1.2.3"）接受语义化版本中次版本部分的任意更新，例如 1.3.0 等；4) 大于或等于约束（">=1.2.3"）接受大于或等于指定版本的任意版本；5) 任意版本约束（"*"）接受任意版本；6) 其他约束，例如析取、合取、GitHub 网址等。

随后，我们按年份对 NPM 中这些约束类别的频次进行考察，以观察约束的使用方式在历史上的变化。此外，分析 NPM 数据时面临的一个挑战是，有些软件包会发布大量版本（React 的版本超过 1,000 个），因此汇总所有版本的数据可能会使结果偏向版本较多的软件包。在 RQ1 中，我们只选取每个软件包在每一年内上传的最新版本。这样既能按时间进行划分，又能避免这种偏差。

B. RQ2：更新中的语义化版本

现在，我们来考察开发者在发布更新时如何递增语义化版本号。首先找出 NPM 历史上发生的所有软件包更新，并将每次更新归类为漏洞更新（例如 5.4.8 → 5.4.9）、次版本更新（例如 5.4.8 → 5.5.0）或主版本更新（例如 5.4.8 →6.0.0）。

人们可能会认为，只需将同一软件包的连续版本识别为更新即可。然而，NPM 允许以非时间顺序发布版本。此功能使并行版本分支的维护成为可能。例如，考虑以下按时间顺序排列的版本：1.0.0，然后是 2.0.0、1.0.1，最后是 2.0.1。在这个例子中，挖掘出的更新应包括：1.0.0 → 2.0.0、1.0.0 → 1.0.1 和 2.0.0 → 2.0.1，因为这些更新最接近基于源版本，同时在时间顺序和数值上保持一致。我们不会纳入更新 1.0.1 → 2.0.0，因为它在时间顺序上不一致，因此 2.0.0 不太可能是 1.0.1 的派生版本。

为确定更新集合，我们按照主版本号组件相同这一等价关系对版本分组，并假设各组内部按时间顺序排列。随后，我们得到每组内部版本之间以及不同组之间的更新。延续上面的示例，我们有两组：{1.0.0、1.0.1} 和 {2.0.0、2.0.1}。根据组内排序，我们得到 1.0.0 → 1.0.1 和 2.0.0→ 2.0.1；根据组间排序，我们得到 1.0.0 → 2.0.0。我们认为，这个算法很好地反映了开发者发布更新的方式，并在第七节讨论其他方案。计算这些更新时，我们首先筛除所有预发布版本（例如 1.2.3-beta5），最终得到 1,453,789 个至少有一次更新的软件包（共 2,869,085 个软件包）。随后，我们筛除了 52,279 个组内时间顺序不一致的软件包。

在确定了所有更新及版本递增类型后，我们首先考察三种更新类型在总体样本中的分布，然后将其与引入漏洞和修补漏洞的更新子群体进行比较。修补漏洞的更新可直接从抓取的安全公告数据库中识别，而引入漏洞的版本则被定义为包含该漏洞的最小版本。为避免某些软件包更新数量过多所造成的偏差，我们的顶层聚合以软件包而非更新为单位。对于每个软件包，我们分别确定其各类更新所占的比例（按安全影响进行划分），然后将这一百分比在所有软件包中的分布可视化。这样，我们便能得出有关软件包及软件包开发者在更新过程中通常如何处理语义化版本号递增的结论。此外需要注意的是，在按引入漏洞的更新进行划分时，我们并非试图研究恶意软件，而是研究那些（很可能是无意中）引入漏洞的更新。

C. 研究问题*3*：过时依赖项与更新流程

迄今为止考察的属性都是各个软件包的局部属性，也就是说，我们分别分析了每个软件包。现在，我们希望回答 NPM 软件包通常有多过时，以及更新需要多长时间才能传递

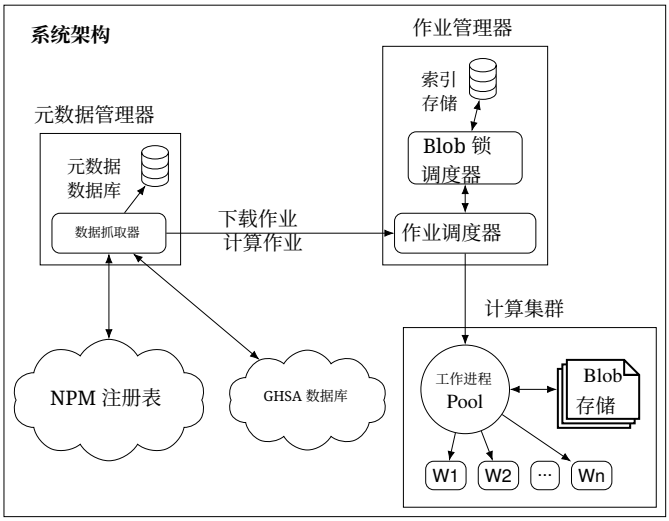


图1：系统架构概览。

到下游软件包。这两项属性都取决于下游软件包的传递依赖闭包中的所有软件包。然而，要精确推理依赖项是如何求解的颇具挑战性：一方面，NPM 的依赖求解算法十分复杂（见第 II-2 节）；另一方面，我们还希望对其随时间的变化进行参数化。

为了在不同时间点准确地计算求解结果，我们使用原生 NPM 求解器，并结合一个代理来模拟历史上任意时间点的世界状态（详见第 IV-C 节）。借助这一关键工具，我们开展了两项实验：首先，求解 NPM 中每个软件包最新版本的依赖项，并观察有多少软件包存在过时依赖；然后，通过在不同时间点求解下游软件包的依赖项，探索更新如何传递到下游软件包，直至其收到更新。

D. RQ4：分析更新中的代码变更

在分别考察开发者如何使用版本约束和版本号之后，我们接下来将这些发现与对更新实际变更内容的总体描述相结合。对于每个已识别的更新，我们都会解压两个版本中的打包代码，并查找文件变更。随后，我们将变更分类为修改依赖项、修改代码（.js、.ts、.jsx、.tsx）、两者均有修改，或两者均未修改。最后，我们按语义化版本增量类型对这些变更类型的分布进行分析，并再次按软件包归一化，以避免更新次数较多的软件包造成偏差。

借助我们的数据集，我们可以进行更深入的分析，但这超出了本文的范围。请注意，许多软件包上传的是编译后的或压缩后的 JavaScript 代码，因此即使只是查看简单的逐行差异也很困难。此外，我们本可以将其他文件类型（如 .sh 等）也计为代码，但我们选择专注于 JavaScript 代码。

IV. 系统架构

为了实施我们的方法，我们需要一个能够抓取并存储所有元数据和 tarball 数据的系统，并允许我们对元数据和 tarball 数据进行分析和实验。该系统需要能够在我们由 Slurm 支持的学术高性能计算集群 [26], [27] 上运行。为了解决这个问题，我们设计了自己的系统，并将其划分为 3 个主要组件（图 1）：

1）元数据管理器会持续从 NPM 抓取数据，并定期从 GitHub 安全公告数据库抓取数据；2）作业管理器接收来自元数据管理器的作业请求（通常是 tar 包下载或计算作业），随后协调作业执行和分布式文件系统锁；3）计算集群可用于启动工作节点并访问网络文件系统。

现在，我们将说明如何完成方法所要求的主要任务。

A. 元数据采集

NPM 将元数据存储在 CouchDB 数据库中。CouchDB 是一种面向文档的 JSON 数据库，很适合 NPM，因为它没有固定架构，并允许 JSON 对象（例如 package.json 文件）任意嵌套。我们发现，由于其结构极其松散，它并不适合用于数据分析。CouchDB 中的 package.json 文件几乎没有经过任何验证，因此若不先清理数据，就难以用于分析。

元数据管理器（图 1 左上方）持续通过 NPM 的变更 API [28], 接收元数据变更，验证这些变更，并将数据插入 PostgreSQL [29]。此外，元数据管理器还会定期抓取 GitHub 安全公告数据库，并将安全元数据一并导入 PostgreSQL。RQ1 和 RQ2 完全可以通过对元数据管理器执行 PostgreSQL 查询来回答。

收到包含新软件包 tarball URL 的元数据变更后，元数据管理器会将一个 tarball 下载任务加入队列，随后由作业管理器处理。

B. Tarball 数据获取与计算集群

为了抓取和存储软件包 tarball，我们需要能够存储数千万个 tarball，同时支持对存储进行并发写入（因为新的 tarball 会持续下载），以及在执行分析时对存储进行并发读取。

计算集群中的工作节点通过网络文件系统连接。一个有趣的方法是使用 Hadoop [30] 构建在网络文件系统之上的技术来实现这一点。不过，考虑到 Hadoop 在存储大量小文件方面的可扩展性 [31]（这是我们的使用场景），我们没有探索这种方法。此外，我们不确定 Hadoop 是否能在网络文件系统之上正确且高效地运行。

相反，我们将 tarball 数据存储在一个构建于网络文件系统之上的定制 Blob 存储系统中（图 1 右下方）。作业管理器（图 1 右上方）控制对 Blob 存储的访问，跟踪字节偏移量并协调写入锁，而计算集群中的各个工作节点则执行网络磁盘 I/O。

作业管理器从元数据管理器收到下载作业请求后，便会将下载作业分配给一个工作节点，并下载 tarball 压缩包。类似地，元数据管理器也可以发送计算作业请求，作业管理器会将其分发给多个工作节点，并可选择允许每个工作节点从 Blob 存储中执行无锁只读操作。

该系统使我们能够持续抓取并存储数千万个 tarball 压缩包，并在回答研究问题3时高效地检索这些压缩包以进行计算。此外，虽然研究问题3不会从 Blob 存储中读取数据，但它遵循相同的计算工作流。

C. 时间旅行依赖解析器

为了开展第 III 节中针对研究问题3所述的实验，我们需要观察在 NPM 历史上的任意时间点，软件包的依赖项将如何得到解析。我们构建了一个代理服务器，可与原版 NPM 配合使用，从而实现时间旅行依赖解析。

NPM 的命令行工具允许用户指定一个自定义包注册表，以替代 npmjs.com。要使用我们的时间旅行解析器，我们指定一个指向代理服务器的注册表基础 URL，并在 URL 中包含要回溯到的时间戳。随后，代理服务器会接收该时间戳，并重写来自 npmjs.com 的响应，移除时间戳之后发布的软件包版本。由于这不依赖我们系统的其他部分，因此设置和使用都极为简单。不过，为了扩展跨整个数据集的计算规模，我们使用了上文第 IV-B 节讨论的计算能力。

V. 结果

从总体上看，如果一个包生态系统能够让积极的更新（性能改进、漏洞修复、安全补丁等）迅速、轻松地被下游依赖采用，同时让具有破坏性的变更（安全漏洞、恶意软件等）传播得较慢，那么我们会认为其更新分发是健康的。在 NPM 中，更新的流动取决于两个因素：下游开发者通常如何为依赖项指定版本约束（RQ1），以及上游开发者在发布更新时通常如何递增版本号（RQ2）。我们首先说明数据集的整体结构和一般属性，然后分别讨论 RQ1 和 RQ2，最后考察 RQ1 和 RQ2 在生态系统实践中如何交织（研究问题3），以及它们与更新的实际内容有何关联（RQ4）。

A. 数据集结构与一般属性

如第 I 节所述，我们收集的数据分为两部分：1）生态系统元数据：包括完整的软件包列表（2,663,681 个软件包）、每个软件包的所有版本（28,941,927 个版本），以及每个版本的元数据，包括版本上传时间、版本号、依赖项、描述、代码仓库链接等。我们还完整抓取了所有 NPM 软件包的安全公告，其中包括哪些版本存在漏洞，以及哪些版本修复了漏洞的数据。2）已发布软件包的 tar 包：我们的系统已下载每个软件包每个版本的完整源代码压缩包[3] 。

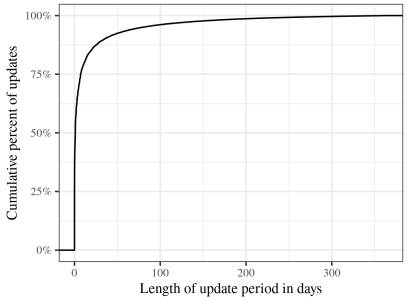
在深入探讨核心研究问题之前，我们先讨论数据集的一般属性。图 2 展示了关于我们主要关注对象——更新和依赖项的三种分布。

图2a 展示了软件包更新间隔分布的经验累积分布函数（ECDF），该函数基于 1,401,510 个软件包和 16,547,653 次挖掘出的更新计算得出（第III-B节）。一个令人意外的发现是，许多情况下更新发布得非常迅速：25% 的更新间隔仅为 39.87分钟 或更短，50% 的更新间隔为 22.71 小时或更短。不过，更新间隔也存在较长的尾部：间隔最长的 25% 更新为 7.78 天或更长，10% 为 40.12 天或更长。平均而言，更新间隔为 21.03 天。对数据的人工检查表明，更新行为具有明显的突发性：开发者会在很短时间内连续发布多个更新，随后又长时间保持沉默；不过，这一假设仍有待更深入的研究。

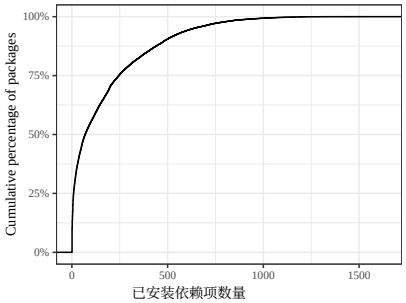
图 2b 和图 2c 分别展示了（传递）依赖项数量和下游软件包数量（即传递反向依赖）的分布经验累积分布函数。我们选取了每个至少有过一次更新的软件包中最新的非预发布版本（以排除已被弃用的软件包），共得到 1,401,510 个软件包。随后，我们使用 NPM 的时间旅行变体解析这些软件包的依赖项，并收集软件包之间的传递依赖关系，不考虑版本。由于 NPM 实际解析失败（例如缺少依赖项）以及计算集群中的暂时性系统故障（第七节将对此作更多讨论），部分软件包的依赖解析失败。总的来说，我们的实验成功运行 NPM 依赖解析器，处理了 696,419 个软件包。数据显示，软件包平均有 167.87 个依赖项；95% 的软件包解析结果规模不超过 636 个依赖项，而规模最大的解析结果则多达 1,641 个依赖项。

然而，转向下游软件包（图 2c）时，情况相当不对称：有大量下游软件包依赖的软件包构成了长得多的尾部。我们观察到，被依赖次数最多的 3 个软件包是：1）supports-color（用于检测终端是否支持颜色，624,883 个下游软件包）；2）debug（日志库，571,547 个下游软件包）；3）ms（时间转换库，515,684 个下游软件包）。另一方面，许多软件包除了被少数几个下游软件包使用外，几乎无人使用；50% 的软件包只有 2 个或

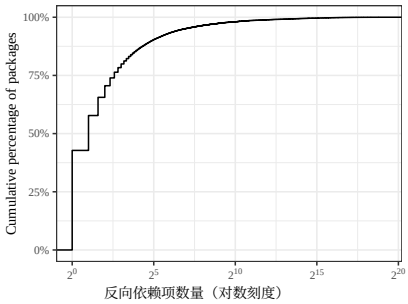
[3]不包括已删除内容，相关内容将在第七节中介绍



（a）软件包两个版本发布之间相隔天数的经验累积分布函数。请注意，此图特别排除了软件包非预发布版本的更新。



（b）各软件包（传递）依赖项数量的经验累积分布函数。该数据是在图 5 的实验中，通过解析 NPM 上每个软件包的最新版本收集的。



(c) 每个软件包的反向（传递）依赖项数量的经验累积分布函数。请注意，x 轴采用对数刻度。此数据是在图 5 的实验中，通过解析 NPM 上每个软件包的最新版本收集的。

图 2：一般属性的经验累积分布函数图 展示 NPM 生态系统在版本管理和依赖项方面的一般属性 s.

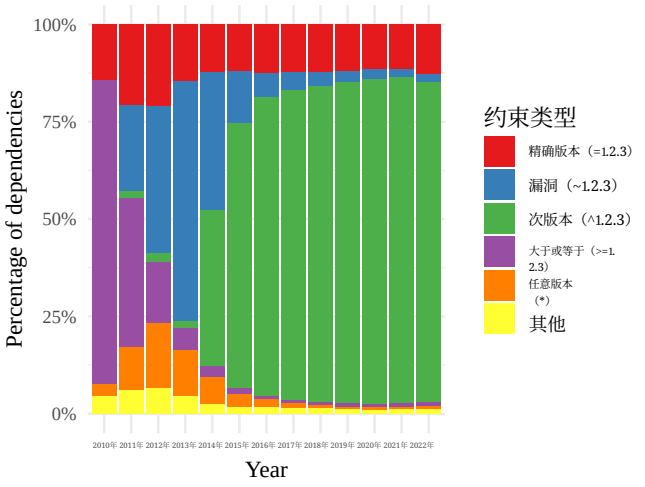


图 3：各类版本约束随时间变化的相对流行度。百分比不跨年份累积，仅反映每年发布的依赖项。

更少的下游软件包，而 90% 的软件包仅被 30 个或更少的下游软件包使用。

B. RQ1：版本约束使用情况

如第 III-A 节所述，开发者可以通过不同方式指定版本约束，从而控制这些依赖项是否安装较新的版本。图3展示了自 NPM 于 2010 年推出以来，各年份发布的每种主要版本约束类型的使用频率。对于每一年，我们仅统计至少发布过一个版本的软件包；若某个软件包在当年发布了多个版本，则仅统计该年最新发布的非预发布版本。2022年，共有 429,265 个软件包至少发布过一个版本，而所有年份中共有 1,678,681 个不同的软件包。

随着时间推移，版本约束的使用呈现出几种有趣的趋势。首先，约 78.36% 的初始依赖项被指定为接受高于某个特定版本的任意版本（大于等于，紫色柱），例如 "react" : ">= 1.2.3"。随后，开发者在 NPM 推出后的头 3 至 4 年内便弃用了大于等于约束，这很可能是因为随着库开始引入破坏性变更，大于等于约束会自动应用这些变更，使得维护变得困难。其次，尽管次版本号组件灵活的约束（次版本，绿色柱）目前占依赖项的大多数，但这种仅使用次版本灵活约束的做法直到 2014年 才开始出现，之后迅速普及。次版本灵活约束的扩展与漏洞组件灵活约束（漏洞，蓝色柱）使用率的下降同时发生。第三，开发者近来几乎只倾向于使用两种约束：精确版本约束（精确版本，红色柱）和次版本号组件灵活约束。这两类约束在 2022 年 合计占比超过 94.85%。最后，可能自动接收更新的依赖项所占比例（红色柱以下的所有部分）在 NPM 的整个发展历程中一直相对稳定，目前约占所有依赖项的 87.32%。

C. RQ2：更新中的语义化版本

RQ1 考察了指定依赖项时语义化版本控制的使用情况，而 RQ2 考察了部署这些依赖项的发布时语义化版本控制的使用情况。图 4 展示了箱线图，其中每个观测值表示某个软件包的更新中属于三种语义化版本增量类型之一的比例，并按安全影响进行归一化。此分析涵盖 1,401,510 个软件包和 16,547,653 次更新，如第III-B节所述。

我们发现，在无安全影响类别（占更新的绝大多数）中，最常见的更新远远是漏洞语义化版本增量：75% 的软件包中，至少 66% 的更新属于此类（最左侧红框的下四分位数）。其次最常见的是

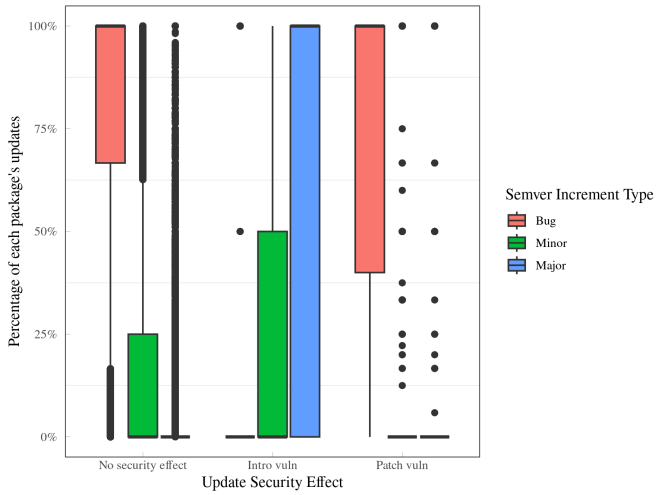


图4：箱线图展示了按语义化版本增量类型划分的软件包更新百分比分布，并按安全影响进行细分。对于每种安全影响，各语义化版本增量类型的百分比均经过归一化。

次版本语义化增量，而最不常见的是主版本语义化版本增量。

然而，当我们考察引入漏洞的更新时，情况就有所不同。大多数软件包通过主版本语义化增量引入漏洞，这表明漏洞往往是在软件包开发者发布新的主版本时引入的，这些版本可能包含许多新功能，并对代码库进行重大结构性更改。不过，我们确实发现有 29 个离群软件包至少在一次缺陷更新中引入了漏洞。一个特别有意思的例子是 ssri 软件包（一个用于检查加密子资源完整性的库，每周下载量为 2300 万次）从 5.2.1 版本更新到 5.2.2 版本。此次更新试图修补一个正则表达式拒绝服务漏洞，却无意中加重了该漏洞的严重程度：将最坏情况下的行为从二次复杂度改成了指数复杂度 [32]。这凸显了软件包开发者面临的挑战：既要迅速发布漏洞补丁，又必须在处理安全相关代码并通过缺陷更新发布这些代码（这些更新会轻易传播到下游软件包）时格外谨慎。

最后，在漏洞得到修复的情况下，几乎所有补丁都以补丁版本语义化增量的形式发布，这意味着图3所示的87.32%的非精确约束理论上都能够自动接收这些补丁。然而，少数异常软件包将漏洞补丁作为非缺陷更新发布（我们在298个软件包中发现了358次此类更新）。通过人工检查可以看出，其中许多更新都将安全漏洞修复与大量其他更改混在一起，而不是单独发布漏洞修复。例如，xmlhttprequest软件包（每周下载量为120万次）从1.6.0更新到1.7.0时，修复了一个高严重性的代码注入漏洞 [33]。该更新中与安全性相关的部分只有1行，但更新共修改了892行代码。由于缺乏进一步调查，我们无法确定为什么一些开发者选择将安全补丁作为较大更新的一部分，而不是作为独立更新发布。

D. 研究问题*3*：过时依赖项与更新流程

1）软件包的依赖项有多过时？：版本约束与语义化版本更新类型相互配合，共同控制更新流向下游软件包的过程，这一过程涉及许多传递依赖链。下游软件包能否获得最新依赖项，不仅取决于该软件包的版本约束和反向依赖项的语义化版本增量类型，也取决于传递依赖链中间的软件包。

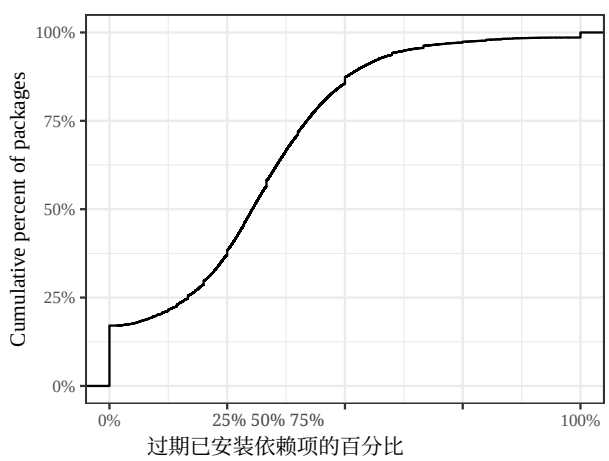
在本实验中，我们选取了至少有一次更新的每个软件包的最新版本（共 1,401,510 个软件包）。随后，我们使用 NPM 的时间旅行变体，在该软件包最新版本上传时（T_P）解析其依赖项。然后，我们观察其已安装依赖项中哪些已经过时：如果某依赖项的版本为 V_D，上传时间为 T_D，且该依赖项的另一个版本 V'_D 的上传时间为 T'_D，满足 $T_D < T'_D < T_P$ 和 $V_D < V'_D$，那么该依赖项即为过时。接着，我们将过时时间定义为满足上述条件的最大 T'_D 所对应的 $T'_D - T_D$。排除暂时性系统故障的影响后，共有 696,419 个软件包成功完成解析。

图5a展示了各软件包中过时依赖项所占比例分布的经验累积分布函数。有一部分软件包（约 17.08%）的依赖项全部是最新的。不过，这些软件包几乎都只有很少的依赖项，平均仅有 3.17 个，而整个样本平均有 167.87 个依赖项。换句话说，这些依赖项全部是最新的软件包主要位于图 2b 中经验累积分布函数的最左侧。

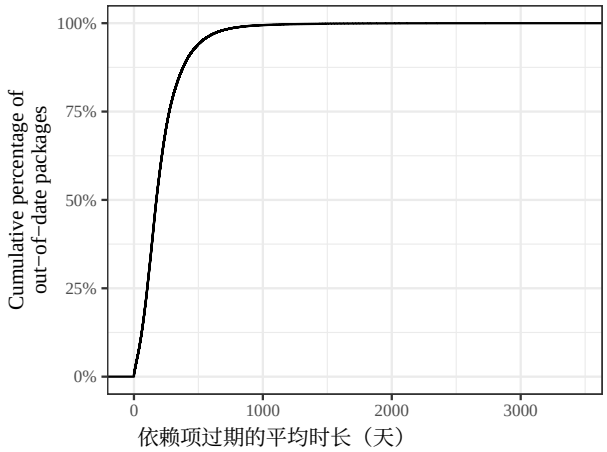
除了最新软件包形成的峰值之外，大多数软件包都至少有一些过时依赖项，其中 62.94% 的软件包有 25% 或更多的依赖项已经过时。这些软件包不仅经常存在依赖项过时的情况，而且过时的时间往往也很长。在至少有一个过时依赖项的软件包中，图5b展示了各软件包依赖项平均过时时长的经验累积分布函数。所有存在过时依赖项的软件包中，有一半的依赖项平均已有 173.87 天或更久未更新；此外，长尾部分有 5% 的软件包，其依赖项平均已有 527.38 天或更久未更新。相比之下，更新平均在 21.03 天内发布，且 50% 的更新在短短 22.71 小时内发布（图 2a）。

软件包的依赖项可能因多种原因而过时，其中一些是有意为之，例如开发者选择继续使用较旧版本的库，而不是重写代码以处理破坏性变更。

2）更新如何快速流向下游？：我们现在希望了解更新如何流向下游软件包，以及在需要人工干预时开发者如何响应。对于每个软件包在 2021 年之前最近的一次更新，我们随机选取了 50 个在更新前刚好与上游软件包保持同步的下游软件包，这些软件包



(a) 每个软件包中过期依赖项所占比例的经验累积分布函数。



(b) 对于至少有一个过期依赖项的每个软件包，依赖项过期天数平均值的经验累积分布函数。

图 5：NPM 生态系统中技术滞后分布的经验累积分布函数图。

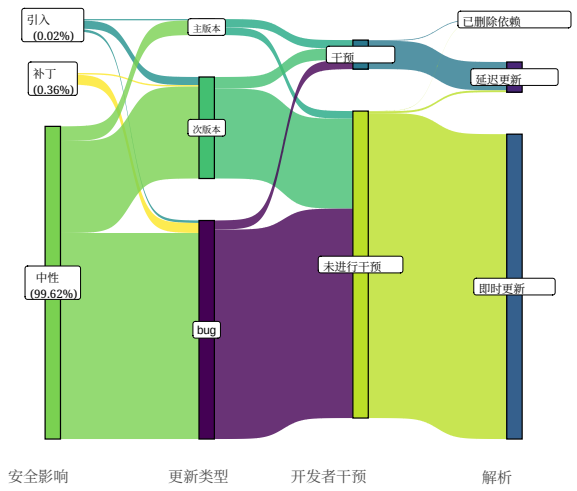


图 6：更新流路径的可视化。

随后，我们使用时间旅行解析器，在更新完成后立即解析下游软件包，并在此后每隔 1 天再次解析，直到对旧版本依赖项的依赖被更新或删除。计入暂时性系统故障后，共成功解析了 888,294 个更新流。

图 6 展示了更新如何流向下游软件包，以及需要开发者干预的频率。更新流包含多个步骤。首先，上游（依赖项）开发者会发布带有特定语义化版本增量类型（主版本、次版本或漏洞）的更新。更新被标记为漏洞、次版本或主版本并上传到 NPM 后，下游软件包便可以接收该更新，包括通过传递依赖链接收。这一过程可以自动完成，也可以由下游开发者手动更新或移除依赖项，或者由传递依赖链中间的开发者更新或移除依赖项。

最常见的情况是，下游软件包会立即收到更新，无需人工干预（$\cdots \rightarrow$无需干预 $\rightarrow$ 即时更新）。当声明了对已更新软件包的约束的软件包，所采用的约束至少与语义化版本增量类型一样灵活时，就会出现这种情况。请注意，声明约束、因而决定是否允许或阻碍更新流的软件包，既可能是最终的下游软件包，也可能是依赖链中间的软件包。这种更新流类型在大多数漏洞和次版本更新中都会出现，这是由约束类型的分布所导致的（图3）。这种更新流占所有已分析更新流的 90.09%，远远高于其他类型，是最常见的情况，表明我们随机抽取的 NPM 生态系统更新流整体上健康状况良好。

第二常见的更新流包括需要下游软件包开发者（也可能包括其他软件包的开发者）进行干预、因而导致延迟的更新（$\cdots \rightarrow$需要干预 $\rightarrow$延迟更新）。这类更新占所有已分析更新流的 9.01%；其中，主版本更新占 28.11%，次版本更新占 40.27%，缺陷更新占 31.62%。需要干预的更新源于约束比语义化版本增量类型更严格。因此，干预涉及开发者改用更灵活的约束类型，或提高约束版本。

少数更新（0.60%）并非由下游软件包的开发者通过干预来解决，而是由中间层的开发者解决（$\cdots \rightarrow$无干预 $\rightarrow$ 延迟更新）。要出现这种情况，中间层软件包的开发者必须指定了过于严格的约束，而

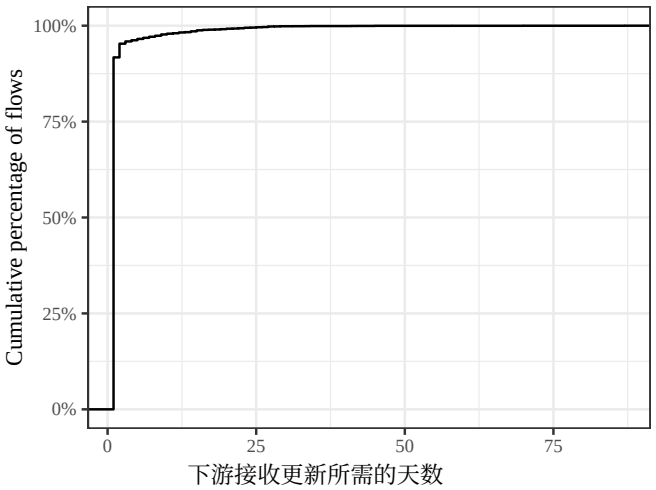


图 7：展示受阻更新流得到解决所需时间的经验累积分布函数图。

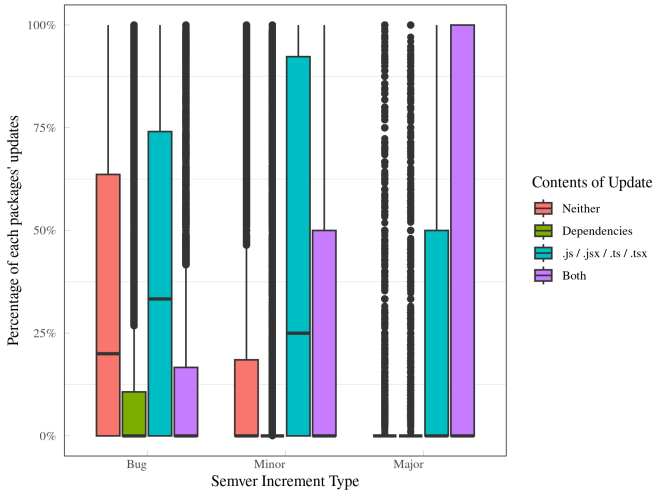


图 8：箱线图展示了按语义化版本增量类型分组的软件包更新中，仅变更代码（.js、.ts、.jsx、.tsx）、仅变更依赖项、两者均变更或两者均未变更的更新所占比例的分布。

下游软件包的开发者则指定了足够灵活的约束，允许采用中间层软件包的干预。由于这种流转极少发生，这表明下游软件包通常采用与其（传递性）依赖项同等严格或更严格的版本约束。从软件工程的角度来看，这很合理，因为更深层的软件包（更接近库而非应用程序）更有动机采用灵活的约束，因为它们很可能会在其他约束相冲突的环境中被重复使用。

最后一种更新流是过期依赖项最终被删除而非更新的情况（$\cdots \rightarrow$ 干预 $\rightarrow$已删除依赖项）。这只占所有分析过的更新流的 0.29%，表明开发者通常不会删除依赖项。未来可以进一步调查，了解开发者为何在少数情况下选择删除依赖项。

在因限制性版本约束而受阻的更新流中，几乎所有更新流都通过人工干预迅速解除阻塞。图 7 展示了每个更新流解除阻塞所需天数分布的经验累积分布函数。大多数受阻更新流（91.74%）在 1 天内解除阻塞，但仍有少数需要 25 天或更久。更新流解除阻塞的速度快得令人惊讶，这主要是因为许多相互依赖的软件包由同一批贡献者开发，他们往往会几乎同时提升版本号并更新其软件包的依赖项。

我们的结果表明，大多数更新都能有效传递到下游软件包，而图 5 表明，大多数下游软件包至少存在一些过时的依赖项。还应对这一现象展开更多研究，但我们怀疑，这在一定程度上是由于每个软件包的依赖项数量（图2b）以及更新速率（图2a）所致。鉴于软件包平均拥有 167 个依赖项，且更新平均每

21 天发布一次，我们预计，对于一个普通软件包而言，每天都会有多个依赖项发布更新，并可能因此过时。即使许多更新会被立即或迅速采用，仍有一些依赖项会变得陈旧。这一现象也可能由我们针对该实验所采用的方法解释，因为我们只选择了在分析时已经处于最新状态的软件包。

E. RQ4：分析更新中的代码变更

现在，我们转而考察软件包更新的内容，而非元数据分析。只有当软件包开发者发布的更新符合下游软件包对漏洞修复、次版本或主版本语义化版本增量的预期时，语义化版本控制才有用。本文着重对 NPM 生态系统中不同更新类型通常包含的内容进行高层次描述。更细致的分析及相关应用虽然有趣且实用，但超出了本文的范围；我们将在第六节讨论正在开展的工作和未来工作。不过，我们认为，该数据集可作为活跃研究领域中评估更新分析系统的有用基础。

图 8 展示了一个箱线图，其中每个观测值表示某个软件包在各语义化版本增量类型中的更新所占百分比，这些更新仅修改代码（.js、.ts、.jsx、.tsx）、仅修改依赖项、同时修改两者，或两者皆未修改。请注意，归类为「两者皆未修改」的更新可能还包含其他变更，例如修改其他类型的文件（README 文件、CSS 等），或除依赖项之外的其他元数据变更。这里使用的软件包和更新与图 4 中相同，并与我们成功下载其 tar 包的软件包和更新取交集，最终共包含 1,339,684 个软件包和 14,903,021 次更新。

首先，我们发现错误修复更新通常不会更改代码文件或依赖项。约 50% 的软件包在大约 20% 或更多的错误修复更新中，代码和依赖项均未更改。

代码和依赖项均未更改；另有 25% 的软件包在大多数（64%）错误修复更新中，两者均未更改。手动检查数据后发现，其中一些更新只涉及元数据（列出的贡献者、描述、README 文件）或配置文件（.json、.yaml 等），而其他更新确实没有任何改动。不过，我们还可以进一步研究这些数据，以便更精确地量化这一现象。其次，虽然并不常见，但有 25% 的软件包偶尔会发布仅修改依赖项的错误修复更新（占错误修复更新的 11% 或更多）。观察次版本和主版本更新可以发现，既不修改代码也不修改依赖项，或只修改其中一项的软件包比例有所下降；而在主版本更新中，大多数软件包会同时修改代码和依赖项。

VI. 讨论

根据我们在第五节中呈现的结果，我们发现这对软件开发者、生态系统维护者和研究人员有诸多启示。使用依赖项的开发者持续面临安全性、可靠性与技术滞后之间的权衡。我们指出，生态系统维护者可以采取措施减少其中一些阻碍，并提出了长期研究方向，以应对软件包生态系统中的一些根本性挑战。

A. 面向开发者

我们针对 RQ1 的研究结果表明，NPM 基本已普遍采用精确版本或允许次版本灵活变动（^）的约束，其中依赖项以允许次版本灵活变动的方式指定的比例最高。实际上，这意味着次版本更新几乎可以像漏洞更新一样轻松地流向下游软件包；我们在 RQ3.2 中通过实验确认了这一点：在抽样的漏洞更新和次要版本更新中，分别有 95.42% 和 86.55% 自动流向下游软件包。这一发现对库维护者很重要，因为他们可能会认为下游软件包会手动检查次版本更新的兼容性。

总体而言，版本发布方式与依赖方式之间存在不一致：以次版本更新和缺陷更新形式发布的版本通常具有不同特征（图4和图8），而下游软件包中的依赖项很少区分次版本更新和缺陷更新（图3）。具体来说，我们发现，81.19% 的更新以缺陷更新形式发布，但 84.01% 的依赖约束同时接受缺陷更新和次版本更新。虽然缺陷更新和次版本更新都应该保持向后兼容性，但由于次版本更新可能更容易包含（无意造成的）破坏性变更，开发者使用可灵活接受缺陷更新的约束（~），而非次版本灵活约束，或许能提高稳定性；前者仍可获得 81.19% 的更新。对于谨慎看待安全性的开发者来说，这一理由可能更充分，因为我们的结果表明，次版本更新比缺陷更新更常引入漏洞；不过，他们仍须保持谨慎，因为缺陷更新有时也会引入漏洞。

B. 面向生态系统维护者

RQ2 的研究结果表明，一些开发者会通过递增次版本号，有时甚至递增主版本号来发布安全补丁。这一发现令人担忧，因为这会使下游软件包更难获得安全修复。这表明，生态系统或许能从生态系统维护者与软件包开发者就安全补丁加强沟通中受益，并有助于确保安全补丁及时发布、改动尽可能少，且以符合语义化版本规范的缺陷更新形式发布。

我们在 RQ3.2 中的发现表明，只有少数更新流会因中间依赖项而受阻。这或许是最令开发者沮丧的情况，因为很难解决。一个办法是使用 NPM 的覆盖功能 [34], ，该功能允许下游软件包强制覆盖传递依赖项的版本，即使这会违反版本约束。虽然这在短期内可能有效，但开发者现在需要承担维护负担：在覆盖不再必要时将其移除，否则未来可能会导致构建失败。为了改善开发者体验，生态系统维护者可以：1）将我们的分析与中心性分析相结合，找出经常阻碍更新流的关键软件包，并与其合作解决问题，从而减少更新传播阻塞的发生频率；2）改进围绕覆盖功能的生态系统工具，帮助开发者在覆盖不再必要时自动将其移除。

C. 面向研究人员

我们在 RQ2 中的发现表明，尽管 NPM 开发者通常会努力遵循语义化版本规范，但他们并非总能始终如一地做到这一点。因此，下游软件包的开发者无法完全确信更新依赖项时究竟会获得什么（尤其是恶意开发者发布恶意软件的情况！）。这表明，静态或动态程序分析工具存在一个有益且广阔的设计空间，可帮助了解更新实际带来了哪些变化。这类工具可以用于检查语义化版本合规性 [35], [36], 、确认更新是否正确修补了所声称的漏洞、检查行为中可能存在的漏洞变更 [37],，或检测恶意软件。研究语义化版本合规性随时间变化的趋势或许尤其有意义，因为我们的分析显示，2010—2022 年间依赖约束的流行程度发生了明显变化。

在其中一些方向上，已经有颇具前景的持续研究，尤其是通过元数据和轻量级语法特征进行恶意软件检测。 [7], [22], [38] 在研究问题4中，我们发现相当一部分软件包发布的漏洞更新既不会修改 js、ts、jsx、tsx 文件，也不会修改依赖项。这表明，很大一部分更新可能修改的是其他类型的文件，而恶意行为者或许会利用这些变更来隐藏恶意更改。无论这类工作的目标是恶意软件检测、缺陷检测还是其他分析，我们的结果都表明，这类工具应致力于处理多种文件类型，例如代码、配置文件、嵌入式二进制文件、Shell 脚本等。

研究问题3.1中的分析发现，NPM 软件包存在大量技术滞后，因此帮助开发者减少技术滞后的工具可能会产生显著影响。在我们此前的工作中，我们构建了一个名为 MAXNPM 的工具， [14], 它允许开发者在满足当前版本约束的同时，以尽量减少技术滞后（或实现其他目标）的方式解决依赖项问题，但无法帮助处理已经过时的约束本身。Jayasuriya 提出的补充性未来研究 [39], 或许能通过协助开发者应对破坏性变更、进行代码迁移，帮助他们完成这些手动更新。

VII. 效度威胁

A. 外部效度

我们无法可靠地抓取已从 NPM 删除（因恶意软件、版权侵权等原因）或已取消发布（由开发者主动取消）的软件包，因此在分析中将其排除。因此，我们的结果可能无法推广到经常被删除的恶意软件或其他类型的软件包。

除了已删除的软件包，我们考虑了整个生态系统，包括所谓的“琐碎”软件包 [8], [9] 以及看似无关紧要的软件包（例如反向依赖很少的软件包）。我们认为，很难判断一个软件包是否真的无关紧要，因为即使反向依赖很少的软件包，也可能实际上是一个已发布到 NPM 上的应用程序。此外，“影响力较低”的开发者仍然很重要，因为他们使用 NPM 的体验关系到生态系统的未来。

我们只从 NPM 获取软件包，不考虑 GitHub 或其他来源。因此，本研究的结果可能无法推广到 JavaScript 应用程序（而非库），因为只有部分开发者会选择将其应用程序发布到 NPM。此外，一些开发者可能会将依赖项的源文件直接复制到自己的软件包中，而我们无法检测到这种情况。最后，在推广我们关于安全漏洞的研究结果时务必谨慎，因为我们只能获取已知 漏洞的信息，而这可能只是所有漏洞中的一小部分。

B. 内部效度

第 IV 节介绍的系统包含许多环节，系统中可能存在影响实验结果的漏洞。例如，我们可能在抓取过程中遗漏了一些软件包，或者下载了错误的软件包。我们认为这种情况不太可能发生，因为我们为系统编写了单元测试，并在一小部分软件包上进行了测试，而且没有发现任何漏洞。

在针对研究问题3（第五节-D）运行数百万次软件包安装时，由于 /tmp 溢出，我们的计算集群间歇性地发生了故障。由于这些故障由系统状态而非软件包导致，我们认为这不会使结果产生偏差。为验证这一点，我们计算了成功包和失败包的直接依赖数量的均值和中位数，发现成功包的直接依赖数量均值为 9.91，中位数为 5；而失败包的直接依赖数量均值为 10.93，中位数为 6。这表明失败包的规模稍大，但还不足以使成功包缺乏代表性。成功包和失败包中都存在直接依赖数量较多的异常值。

C. 构念效度

在 RQ2–RQ4 中，我们使用第III-B节所述的算法来计算更新。由于没有正确挖掘更新的真实值，人们或许会考虑改进该算法。具体而言，可以将次版本号组件也纳入考量，以构建更细粒度的等价类。然而，这不会改变算法已成功处理的结果；而且拒绝率已经相当低（1.8%），因此我们认为，采用更复杂的算法并不值得冒分析错误的风险。

在 RQ4 中，我们将代码变更定义为扩展名为 .js、.ts、.jsx 或 .tsx 的文件。这是因为我们希望聚焦于 JavaScript 和 TypeScript 代码，但这可能导致我们遗漏了使用其他扩展名的部分 JavaScript 或 TypeScript 代码。根据研究目的，未来的工作可以考虑采用更广泛的代码定义，例如将 Shell 脚本也纳入其中。

VIII. 结论

我们对 NPM 中的语义化版本控制进行了大规模分析，并构建了一个完整且可复用的数据集，其中包含 NPM 的完整包元数据和 tarball 数据。我们发现，与错误修复（即补丁）语义化版本更新相比，次版本更新引入安全漏洞的风险更高，这说明开发者使用错误修复灵活约束（~）可能有其必要性，尽管 NPM 生态系统已基本弃用这类约束，转而采用次版本灵活约束（^）。我们发现，大多数安全补丁都是通过错误修复更新引入的，但也发现了一组令人担忧的异常值：它们作为次版本甚至主版本更新发布，可能导致安全补丁的采用速度变慢。未来对 NPM 生态系统的研究可以基于我们的数据集和工具，考察更新中与漏洞和/或安全漏洞相关的内容，以及缓解技术滞后的机制。

IX. 数据可用性

我们永久存档在 Zenodo [15] 中的研究成果包含我们的工具以及数据集的元数据。在 https://dependencies.science 上，我们发布：1）持续更新的数据快照（包括所有软件包的内容），以及 2）我们的抓取系统和数据分析脚本的完整实现。我们希望该网站能为其他研究 NPM 的人员提供有用的资源。

致谢

我们感谢 Northeastern Research Computing，尤其是 Greg Shomo，为我们提供计算资源和技术支持。

参考文献

[1] J. M. Gonzalez-Barahona、P. Sherwood、G. Robles 和 D. Izquierdo，“软件编译中的技术滞后：衡量软件部署的过时程度”，载于 *Open Source Systems: Towards Robust Practices*，F. Balaguer、R. Di Cosmo、A. Garrido、F. Kon、G. Robles 和 S. Zacchiroli 编。Cham：Springer International Publishing，2017年，第182–192页。[2] A. Zerouali、T. Mens、J. Gonzalez-Barahona、A. Decan、E. Constantinou 和 G. Robles，“衡量组件仓库中技术滞后的形式化框架及其在 npm 中的应用”，*Journal of Software: Evolution and Process*，第31卷，第8期，第e2157页，2019年，e2157 smr.2157。 [在线]。可访问：https://onlinelibrary.wiley.com/doi/abs/ 10.1002/smr.2157[3] B. Chinthanet、R. G. Kula、S. McIntosh、T. Ishio、A. Ihara 和 K. Matsumoto，“npm 漏洞修复的发布、采用和传播滞后”，*Empirical Software Engineering*，第26卷，第3期，第47页，2021年3月。 [在线]。可访问：https://doi.org/10.1007/ s10664-021-09951-x[4] A. Decan、T. Mens 和 E. Constantinou，“npm 软件包依赖网络中技术滞后的演变”，载于 *2018 IEEE Interna-tional Conference on Software Maintenance and Evolution (ICSME)*，2018年，第404–414页。[5] A. Zerouali、E. Constantinou、T. Mens、G. Robles 和 J. Gonz´alez- Barahona，“npm 软件包依赖项中技术滞后的实证分析”，载于 *New Opportunities for Software Reuse*，R. Capilla、B. Gallina 和 C. Cetina 编。Cham：Springer International Publish- ing，2018年，第95–110页。[6] T. Preston-Werner 和贡献者，“语义化版本 2.0.0”，https: //semver.org。访问日期：2023年3月9日，2023年。[7] N. Zahan、T. Zimmermann、P. Godefroid、B. Murphy、C. Maddila 和 L. Williams，“npm 供应链中的薄弱环节是什么？”，载于 *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*，系列：ICSE-SEIP ’22。纽约，纽约州，美国：计算机协会，2022年，第 331 340 [ ] 10 1145 3510457 3513044 –页。在线。可访问：https://doi.org/ . / . R. G. Kula、A. Ouni、D. M. German 和 K. Inoue，“微型软件包的影响：npm JavaScript 生态系统的实证研究”，2017 [ ] 1709 04638 。在线。可访问：https://arxiv.org/abs/ . M. A. R. Chowdhury、R. Abdalkareem、E. Shihab 和 B. Adams，“简单软件包并非无足轻重：npm JavaScript 软件包的实证研究”，*IEEE Transactions on Software Engineering*，第48卷，第8期，第2695–2708页，2022年。[10] R. Abdalkareem、O. Nourry、S. Wehaibi、S. Mujahid 和 E. Shihab，“开发者为何使用简单软件包？ npm 的实证案例研究”，载于 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*，系列：ESEC/FSE 2017。纽约，纽约州，美国：计算机协会，2017年，第385–395页。 [在线]。可访问：https://doi.org/ 10 1145 3106237 3106267 。 / . A. Decan、T. Mens 和 E. Constantinou，“npm 软件包依赖网络中安全漏洞的影响”，载于*Proceedingsof the 15th InternationalConference onMining Software Repositories*，系列：MSR ’18。纽约，纽约州，美国：计算机协会，2018年，第181–191页。 [在线]。可访问：https://doi.org/ 10 1145 3196398 3196401 。 / . A. Zerouali、T. Mens、A. Decan 和 C. De Roover，“npm 和 RubyGems 依赖网络中安全漏洞的影响”，*EmpiricalSoftware Engineering*，第27卷，第5期，第107页，2022年5月。在线。可访问：https://doi.org/ . /s - - - C. Liu、S. Chen、L. Fan、B. Chen、Y. Liu 和 X. Peng，“通过 npm 生态系统中的依赖树揭示漏洞传播及其演变”，载于 *Proceedings of the 44th International Conference on Software Engineering*，系列：ICSE ’22。纽约，纽约州，美国：计算机协会，2022年，第672–684页。 [ ] 10 1145 3510003 3510142 在线。可访问：https://doi.org/ . / . D. Pinckney、F. Cassano、A. Guha、J. Bell、M. Culpo 和 T. Gamblin，“通过最大可满足性模理论实现灵活且最优的依赖管理”，载于 *Pro-ceedings of the2023International Conference on Software Engineering*，系列：ICSE，2023年。[15] D. Pinckney、F. Cassano、A. Guha 和 J. Bell，“Artifact For A Large Scale Analysis of Semantic Versioning in NPM”，2023年1月。 [在线]。可访问：https://doi.org/ 10 5281 7552551 。 / zenodo。C. Bogart、C. Kastner 和 J. Herbsleb，“出问题时就是出问题：生态系统开发者如何推理依赖项的稳定性”，载于 2015年第30届IEEE/ACM自动化软件工程国际会议研讨会（ASEW），2015年，第86–89页。[17] S. Raemaekers、A. van Deursen 和 J. Visser，“语义化版本控制与破坏性变更：对 Maven 仓库的研究”，载于《2014年IEEE第14届源代码分析与处理国际工作会议》，2014年，第215–224页。[18] E. Wittern、P. Suter 和 S. Rajagopalan，“探究 JavaScript软件包生态系统的动态”，载于《第13届软件仓库挖掘国际会议论文集》，丛书编号 MSR ’16。美国纽约州纽约：计算机协会，2016年，第 351 361 [ ] 10 1145 2901739 2901743 –页。在线。可访问：https://doi.org/ 。/。J. Dietrich、D. J. Pearce、J. Stringer、A. Tahir 和 K. Blincoe，“实际环境中的依赖项版本控制”，载于《第16届软件仓库挖掘国际会议论文集》，丛书编号 MSR ’19。IEEE出版社，2019年，第349–359页。 [在线]。可访问：https://doi.org/ 10 1109 2019 00061 。/MSR。A. Decan 和 T. Mens，“软件包依赖关系能告诉我们什么有关语义化版本控制的信息？”《IEEE Transactions on Software Engineering》，第47卷，第6期，第1226–1240页，2021年。[21] F. R. Cogo、G. A. Oliva 和 A. E. Hassan，“npm 生态系统中的依赖项降级实证研究”，《IEEE Transactions on Software Engineering》，第47卷，第11期，第2457–2470页，2021年。[22] A. Sejfia 和 M. Sch¨afer，“恶意 npm 软件包的实用自动化检测”，载于《第44届国际软件工程会议论文集》，丛书编号 ICSE ’22。美国纽约州纽约：计算机协会，2022年，第1681–1692页。 [在线]。可访问：https://doi.org/ 10 1145 3510003 3510104 。/。M. Zimmermann、C.-A. Staicu、C. Tenny 和 M. Pradel，“高风险的小世界：npm 生态系统中的安全威胁研究”，载于《第28届USENIX安全研讨会论文集》，丛书编号 SEC’19。美国：USENIX协会，2019年，第995–1010页。[24] M. Ohm、F. Boes、C. Bungartz 和 M. Meier，“监督式机器学习用于检测恶意软件包的可行性研究”，载于《第17届可用性、可靠性与安全性国际会议论文集》，丛书编号 ARES ’22。美国纽约州纽约：计算机协会，2022年。 [在线]。可访问：https://doi.org/ 10 1145 3538969 3544415 。/。“package.json”，https://docs.npmjs.com/cli/v9/configuring-npm/ package-json# dependencies。访问日期：2023年1月20日，2023年。[26] SchedMD 和贡献者，“Slurm工作负载管理器——文档”，https://slurm.schedmd.com。访问日期：2023年3月12日，2023年。[27] A. B. Yoo、M. A. Jette 和 M. Grondona，“Slurm：简单的 Linux 资源管理实用程序”，载于《并行处理作业调度策略》，D. Feitelson、L. Rudolph 和 U. Schwiegelshohn 编。柏林、海德堡：施普林格柏林海德堡出版社，2003年，第44–60页。[28] NPM 和贡献者，“registry-follower-tutorial”，https://github.com/ npm/registry-follower-tutorial。访问日期：2023年3月12日，2023年。[29] T. P. G. D. Group，“PostgreSQL：全球最先进的开源关系数据库”，https://www.postgresql.org。访问日期：2023年3月12日，2023年。[30] T. A. S. Foundation，“Apache Hadoop”，https://hadoop.apache.org。访问日期：2023年3月13日，2023年。[31] S. Balint，“小文件问题”，https://blog.cloudera.com/ the-small-files-problem/。访问日期：2023年3月13日，2009年。[32] B. Caller，“安全公告：npm/ssri 中的正则表达式拒绝服务（正则表达式拒绝服务）”，2021年。 [在线]。可访问：https://doyensec.com/ resources/Doyensec Advisory ssri redos.pdf [33] “CVE-2020-28502 详细信息”，2021年。 [在线]。可访问：https://nvd.nist.gov/vuln/detail/CVE- 2020 28502 - NPM 和贡献者，“package.json”，https://docs.npmjs.com/cli/v9/configuring-npm/package-json#overrides。访问日期：2023年3月10日，2023年。[35] P. Lam、J. Dietrich 和 D. J. Pearce，“将语义融入语义化版本控制”，载于《2020年ACM SIGPLAN编程与软件新思想、新范式及反思国际研讨会论文集》，丛书编号 Onward! 2020。美国纽约州纽约：计算机协会，2020年，第157–179页。 [ ] 10 1145 3426428 3426922 在线。可访问：https://doi.org/ 。/。L. Ochoa、T. Degueule 和 J.-R. Falleri，“Breakbot：分析破坏性变更的影响以辅助软件库演化”，载于《ACM/IEEE第44届软件工程国际会议：新思想与新兴成果论文集》，丛书编号 ICSE-NIER ’22。美国纽约州纽约：计算机协会，2022年，第26–30页。[在线]。可访问：https://doi.org/10.1145/3510455.3512783

[37] J. Yang 和 D. Evans，“自动推断程序演化的时间属性”，载于第*15*届国际软件可靠性工程研讨会，2004年，第340–351页。[38] “Snyk 开源项目”，https://snyk.io/product/ open-source-security-management/。访问日期：2023年1月20日，2023年。[39] D. Jayasuriya，“迈向软件依赖项的自动更新”，载于*2022*年*ACM SIGPLAN*系统、编程、语言与应用：软件服务人类国际会议伴随论文集，丛书编号 SPLASH Companion 2022。美国纽约州纽约：计算机协会，2022年，第29–33页。[在线]。可访问：https://doi.org/10.1145/3563768.3565548