软件库维护活动分析

Alexandros Tsakpinis
fortiss - Research Institute of the Free State of Bavaria
Munich, Germany
tsakpinis@fortiss.org

摘要

工业应用如今大量集成了开源软件库。除了库带来的好处，如果一个库受到漏洞影响而其社区又不活跃于创建修复版本的话，它们也可能构成实际威胁。因此，我想介绍一种针对工业应用的自动监控方法，用于识别那些在当前或未来维护活动方面表现出负面迹象的开源依赖项。由于该领域的大多数研究受限于缺乏特征、标签和传递链接，因而不可应用于行业，我的方法旨在通过捕获直接和传递依赖项在维护活动方面的影响力来弥补这一差距。自动监控依赖项的维护活动可以减少应用程序维护人员的手动工作，并通过持续拥有良好维护的依赖项来支持应用程序的安全性。

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Software and its engineering → Maintaining software**.

KEYWORDS

维护活动，开源软件库，仓库挖掘

1 介绍

情况： 多年来，由于商业、工程和质量原因，在产品生命周期中使用开源软件（OSS）已成为标准 [15]。事实上，一个普通商用产品的代码中有大约 80%由开源组件构成，这突显了在软件行业中已经巩固的实践 [29]。OSS 的一个组成部分是库，它在现代软件开发中扮演着关键角色。软件库的思想是在为特定功能从头编写代码时，可以重用第三方库提供的经过良好测试且已确立的代码来实现所需的功能，而不是重新开始编写 [3]。一旦一个库被包含在一个项目中，它可以被定义为依赖项 [10]。OSS 依赖关系可能会引入漏洞（例如，Log4j[1]），这是软件行业中最为紧迫和重要的问题之一 [13]。如果检测到漏洞，库的社区通常会很快发布一个修复版本 [31]。

问题： 随着应用程序与其依赖项独立演化，应用程序维护者需要仔细监控依赖项的演进 [20]。例如，可能会出现库的支持被中止或暂停的情况，使得不再期望有任何扩展或对关键错误的修复 [3, 30]。如果一个系统直接依赖于一个维护不善的库，那么该库中的任何漏洞都可能对该系统本身造成危害 [32]。此外，依赖于存在漏洞的库不仅会影响所有依赖项目，还会影响通过传递链接连接的所有项目和库，进一步增加受影响项目的数量 [27]。

如果出现这样的问题场景，即一个应用程序依赖于某个存在漏洞且没有活跃社区维护的库时，有两种成本高昂的方式来解决这一安全威胁。首先，应用程序维护人员或外包软件服务提供商通过为该库贡献代码来解决问题，即开发其新版本 [27, 30] 的代码。其次，应用程序维护人员用另一个提供相同所需功能的库替换掉原来的库 [2, 30]。

另一个问题是，一旦库被安装，监控社区支持和维护活动目前主要是手动完成的 [8]。对于少量依赖项来说，这可能不是问题。然而，对于实际系统而言，各种研究表明存在大量直接和传递依赖项，导致监控它们的维护活动变得不可行 [3, 17, 19]。这些大量的直接和传递依赖项，再加上需要监控维护活动的需求，要求降低自动监控系统中依赖项维护活动的成本 [10, 27]。

[1] https://logging.apache.org/log4j/2.x/

然而，大多数研究由于缺乏特征、标签和传递链接而受到限制，因此在行业中不可应用 [28]。要么完全不考虑传递依赖项的维护活动 [9]，要么仅将第一级包含在分析中 [22]。据我所知，没有研究能够全面捕捉直接和传递依赖项在维护活动方面的影响力。除此之外，现有的方法仅根据库的当前状态对其进行分类，并错过了对未来进行预测 [9, 23]。预测一个库未来的维护活动允许应用程序维护者提前考虑对于显示出未来维护活动负面迹象的库应该采取什么行动，在其维护支持实际被取消之前。此外，它们还缺乏针对努力意识的关注度评估，这在行业中非常重要。在我的语境中，努力意识衡量了监控 OSS 库的维护活动能在多大程度上减少应用程序维护者的努力。

解决方案： 本论文的目标是研究允许自动监控现实世界软件系统中依赖关系维护活动的新技术。为了解决这个问题，我建议将问题分为两个子任务，即分类和预测。对于分类，应根据库的项目级特征（例如提交次数、贡献者数量、星标数量等）对其进行分析，并基于当前社区支持活跃程度将其分配到一个类别中。对于预测，应对不同时间段的相关项目级特征进行预测，然后使用这些特征来分配描述未来社区支持活跃程度的标签。对于这两个子任务，必须考虑直接依赖和传递依赖的维护活动。因此，可以识别出存在问题的依赖关系，显示出当前或未来的负面维护活动迹象。

贡献： 作为科学贡献，我想创建一个包含扩展和组合维护活动特征及标签的数据集。文献中提出的标签包括*活跃*、*功能完整*、*潜伏的*和*非活动*，这些标签描述了图书馆的维护活动级别。该数据集用于构建一个模型，根据其维护活动对图书馆进行分类，并预测不同时间段内的维护活动情况，同时考虑直接和传递依赖关系。所提出的方法将在关注努力意识的案例研究中进行评估。

作为实际贡献，我想通过提供一个辅助工具来减少应用程序维护人员在监控工业应用依赖关系方面的维护工作。这样的工具支持应用程序安全，通过持续确保依赖关系得到良好维护。

2 用例动机

目标： 我的用例旨在自动监控工业应用中直接和间接 OSS 依赖的维护活动，以识别显示出可疑维护活动（例如功能完备、休眠、不活跃）的 OSS 库。

参与者： 主要参与者是安全经理和外部软件审计员，他们负责确保组织内仅使用安全的开源软件。如果一个组织中没有单独的安全经理，我的工作也可以直接帮助开发者。请注意，较大的组织将从持续生成的维护活动报告中获益更多，因为他们有更多的员工可以利用这些信息。

相关性对于参与者： 我的用例特别重要，因为目前大多数公司都没有足够的监控机制来维护 OSS 依赖项。要么这些维护活动是手动监控的，要么根本就没有进行监控，这要么成本非常高昂，要么存在安全隐患。然而，缺乏这样的监控机制以及由此导致的未知未维护依赖项是一种潜在的安全威胁。

预条件： 系统中使用的开源软件库应可通过源代码或其构建定义（例如，pom.xml）访问。我的用例仅限于那些列在公开可访问的软件仓库（例如，GitHub）中的开源软件库，以便能够收集所需的特征数据。

触发器： 我的用例的主要触发因素是一个新的 CI 运行。我的方法将作为 CLI 工具集成到一个 CI 步骤中，该工具为每次 CI 运行创建一份关于每个 OSS 依赖的当前和未来维护活动状态的报告。或者，可以通过向 CLI 工具提供单个库字符串或包含多个库的文件来手动触发报告生成。

时间适用性： 我的用例适用于应用程序生命周期中的任何阶段。对于没有持续集成管道的原型和早期应用，可以将命令行工具作为手动触发器来评估选定库是否有足够的维护活动，然后再进行安装。对于使用持续集成管道的所有其他应用程序，我的工具可以作为一个独立的 CI 步骤进行集成，无论它们处于应用程序生命周期的哪个阶段。

基本流程： 基本流程包括两个步骤。首先，每个库应根据当前的维护活动进行分类。其次，预测所有库在接下来几个月的不同时间段内的维护活动。因此，可以识别出当前或未来的可疑 OSS 依赖。

重要的是，分类和预测任务不仅应应用于直接依赖项，还应考虑传递依赖项。这些关于传递依赖项的附加信息可能对那些由于自身的维护活动已被归类为可疑的库来说并不感兴趣，因为传递依赖项的聚合结果无论如何都不会改变总体结果。然而，设想一个场景：某个库的维护活动看起来没有问题，但是存在会被识别为可疑的传递依赖项。这样的库应该被标记为可疑，并由负责人进行更详细的分析，因为在传递依赖项中的漏洞可能与在根库中的漏洞一样危险。这意味着只有当一个库的所有传递依赖项也没有任何负面迹象时，这个库才会被标记为不可疑。

后置条件：在报告了可疑依赖关系作为基本流程的一部分之后，接下来需要描述负责任的人可能会如何处理这些信息。如果一个库被标记为可疑的，则可以采取三种行动，即忽略警告、如果有替代品则替换该库或继续开发。继续开发可以在私有分叉项目中进行，也可以将代码贡献给 OSS 项目。根据组织对其应用程序的安全要求，当一个库被报告为可疑时，可能会出现不同的情况，如图 1 所示。

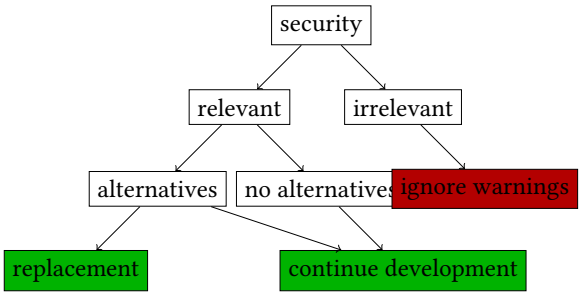

图 1: 可能的场景：库被报告为可疑

维护活动报告可以被视为通过启动标记为绿色的安全增强措施之一来确保应用程序安全的关键因素。采取何种行动取决于具体情境，必须由负责人决定。安全经理可以使用报告的信息来计划库的替换/开发任务，或者在没有资源应对安全威胁的情况下，评估报告中的库与应用程序集成的风险。外部软件审计员可以将关于当前或未来可疑库的报告信息作为其评估的一部分。

预期收益：自动监控系统中 OSS 库的维护活动有两个主要好处。首先，它减少了应用程序维护人员的工作量，因为他们只需要分析整个库集中的一小部分，而直接和间接依赖关系会使得这部分呈指数级增长。其次，在可疑库被识别并相应地采取了应对措施后，可以持续拥有维护良好的库，从而防止库因缺乏活跃的社区支持而受到漏洞影响的情况。

3 研究问题

以下四个研究问题（RQ）来自上述情况和问题：

研究问题 1: 哪些特征和标签能够表征 OSS 库在维护活动方面的特性？**研究问题 2:** 性能如何分类 OSS 库的维护活动，考虑到直接和间接依赖项，在精度和召回率方面？

研究问题 3: 不同时间范围内考虑直接和传递依赖关系时，针对 OSS 库的维护活动预测在精确率和召回率方面的性能如何？

研究问题 4: 维护人员在监控工业应用程序所使用的开源软件库的维护活动方面所付出的努力可以在多大程度上通过使用工具来减少？

4 相关工作

以下，对每个 RQ 的相关文献进行了总结。

4.1 RQ1: 维护活动的特征描述

最近的两项研究表明，社区支持是开发者在选择和采用第三方库和开源软件时最重要的前三项因素之一 [11] 或者最重要 [23] 因素。考虑这些因素基于维护活动已经在库的选择和采用过程中扮演重要角色的前提，并且必须在集成后继续监控。由于篇幅限制，具体维护活动特征在这里未被提及，因为初步文献回顾发现有超过 30 个来源。作为维护活动标签，已识别的状态包括活跃、功能完整、休眠和不活跃 [9, 10, 12, 18, 22, 36]。为了更好地理解底层数据，Choi et al.提出了一个无监督聚类方法（k-均值）[7]。因此，根据已识别的维护活动标签，k 可以设置为四。RQ1 的主要差距是对现有维护活动特征和标签的经验评估，以便更好地了解如何表征开源软件库。

4.2 RQ2: 维护活动的分类

为了识别未维护的开源软件库，Coelho et al.使用了基于随机森林算法的 13 个项目级 GitHub 特征 [9]。然而，分类仅基于项目级别特征，并没有考虑直接和间接依赖关系，这在工业环境中是一个重要的差距。为了解决这一差距，通过计算每个直接依赖及其直接依赖的风险，来计算系统中每一个开源软件库的累积废弃风险。这意味着第一级的间接依赖被包括进来，但是所有间接链接仍然缺失 [22]。整合间接依赖影响的一种方式是使用不同的影响测量方法，例如度中心性、入度中心性、出度中心性和特征向量中心性 [1]。同样地，应用了 PageRank 算法来根据软件包的中心性识别正在衰退的软件包 [25]。这种方法可以通过将每个库的维护活动作为 PageRank 算法的测量标准来进行调整。一个类似的与间接开源软件依赖相关的问题是许可证违规，这可以自动检测 [14] 并通过在公司中实施开源治理和合规的最佳实践来解决 [16]。有趣的是，有明确的呼吁要求未来的工作构建一个精确模型来自动识别未维护的库 [27, 28]，这激发了我的工作动机，但却忽略了区分已识别的不同维护活动状态 [9, 10, 12, 18, 22, 36]。我计划通过从二分类方法转向多类方法来解决这一差距。

4.3 RQ3：预测维护活动

不同的研究为不同主题预测了特定特征，例如与健康相关的特征 [22, 37] 或流行度衡量指标 [4, 35]。针对多变量维护活动特征的预测目前仍然缺失。统计算法如逻辑回归、k 最近邻、支持向量回归、线性回归和回归树 [22, 37]，以及基于神经网络的算法，如 LSTM RNNs[4, 35] 已被应用。预测周期范围从 1 到 30 天 [4, 22, 35, 37]，1 到 6 个月 [4, 22, 37]，以及长达 12 个月或更长时间 [4, 37]。RQ3 的主要差距在于将传递链接整合进预测方法中，因为当前的方法仅基于项目级别的特征。

4.4 RQ4: 努力意识的案例研究

可用的案例研究与行业合作伙伴评估了识别可疑 OSS 库 [22, 27, 28] 的可行性和性能，但尚未考察在比较手动和基于工具监控方法时可以减少多少工作量。事实上，适用于工业环境的公开可用的用于评估 OSS 库的工具 [21] 缺乏。尽管有多种工具 [2][3][4][5] 管理 OSS 库，在生成 SBOM、识别漏洞或进行安全报告方面，但在维护活动方面的考量要么有限，要么过于简单化，或者不够透明。为了可视化直接和传递依赖的影响，Chen and German构建了一个工具，该工具利用公开的 NPM 特性高亮显示通过依赖网络中的传递链接引发的问题依赖关系使用 [6]。对于持续监控，我的方法可以集成到自动化构建过程中，类似于审计 JS[6] 或依赖机器人 [7]。

5 研究计划

以下是每个研究问题的贡献和方法的总结。

5.1 RQ1: 维护活动的特征描述

我计划通过采用结合定量和定性研究的混合方法来扩展表征 OSS 库维护活动的状态-of-the-art 特征和标签，这增加了结果的有效性，并提供了对研究主题更深更广的理解 [24]。

作为定量研究，将进行文献回顾，分析最新的文献以寻找特征来表征开源软件库的维护活动（例如，提交次数、贡献者（核心）数量、平均问题响应时间）。为了扩展现有的特征，还从相关论文的限制或未来工作部分中提取了附加特征，这些特征有望对表征开源项目在维护活动方面的特性产生积极影响，但尚未经过评估。据我所知，没有研究尝试过现有特征的组合是否能提高分类性能。除此之外，多项研究也表明额外的特征可以进一步提升它们方法的性能 [9, 22]。

[2] https://debricked.com/
[3] https://tidelift.com/
[4] https://snyk.io/
[5] https://endorlabs.com/
[6] https://www.npmjs.com/package/auditjs
[7] https://github.com/dependabot

此外，我计划在定量研究中扩展现有方法的标签，因为据我所知，目前只有带有二元标签的数据集存在，例如维护中的/活跃 vs. 未维护的/不活跃的/废弃的/存档的 [9, 12, 18, 22]。然而，在文献中提到了另外两种应单独处理的状态，即休眠和功能完整。一个处于休眠状态的项目暂时暂停了其维护活动但尚未被定义为未维护的 [36]。一个功能完整的项目可能会表现出类似未维护或休眠项目的迹象，但由于该项目在功能上已经完成，并且可能永远不会需要再次修改（例如 NPM 的 escape-string-regexp[8]）[10]。尽管功能完整项目像未维护或休眠项目一样显示出开发活动，但此类项目通常有一个社区作为后盾，在出现紧急情况时（如发现漏洞）会采取行动。

需要以不同于维护或未维护项目的方式来处理这两个额外状态的项目。一个功能完备的依赖项可以安全保留，即使它显示出未维护的迹象。使用现有的二元分类方法，功能完备的项目可能会被错误地计为未维护，因为它们通常一年以上没有提交活动。基于最后一次提交活动设置这样的阈值常用于识别未维护项目，导致假阳性 [18]。相比之下，以前的方法会将休眠项目错误地标记为正在维护，因为他们常常认为被归档或在 README 中明确定义为未维护的项目是未维护的，从而导致假阴性 [9]。

为了能够根据维护活动对每个仓库进行标记，文献中应检查诸如活跃、功能完整、休眠和不活跃等标签的定义。除了已经在文献中识别出的四个标签外，在定量研究的文献回顾过程中可能会发现更多的标签。由于大多数标签定义基于简单且有限的假设，因此应该详细说明一种方法，该方法提供现实的标记策略并解决这些限制。

定量研究完成后，将使用两个目标群体进行定性研究，以生成文献中未提及的功能和标签的新想法。因此，我想采访在日常工作中使用 OSS 库的研究和行业中的有经验的人士。通过采访来自研究和行业的类似数量的人，我试图减少最终功能和标签的偏差。为了支持受访者，我计划系统地准备并在文献回顾中展示所有发现的功能和标签。在访谈过程中，我还旨在询问在研究和行业中应用于软件开发的 OSS 库监控维护活动的最新机制、方法和自动化程度。

回答 RQ1 的最后一步是创建一个多类标签数据集。因此，我计划专注于那些将库代码公开存储（如 GitHub、GitLab）的常用包管理器，例如 PyPI、Maven 和 NPM[38]。对于每个包管理器，我希望选择流行度低、中、高的库以减少数据集中与之前主要关注高度流行的库的方法相比的偏差 [9, 22, 28]。扩展的数据集大小也可能对第 5.2 和 5.3 节中提到的分类和预测产生积极影响。所选库的所有直接和间接依赖项都应包含在数据集中，以便能够在后期计算传递性影响。对于每个选定的库，我计划提取已识别的维护活动特征并根据建议的标签策略分配一个标签。为了更好地理解所得的数据集，将使用无监督统计方法如多类聚类或主成分分析（PCA）进行描述性分析。

5.2 RQ2: 维护活动的分类

基于前一个研究问题的标注数据集，我计划通过考虑直接和传递依赖关系来扩展和改进现有的开源软件库维护活动标签分类方法。因此，我打算进行不同的研究活动。首先，我计划进行特征重要性分析，并扩展现有方法，使用随机森林分类器从二元分类转向多类分类方法，类似于 [9]。这使得我的结果可以与仅考虑项目级特征的先前工作进行比较。

其次，我计划通过整合直接和传递依赖关系对其维护活动的影响来扩展最先进的分类方法。因此，我想遵循 Mujahid et al.的想法，应用谷歌的页面排名算法，该算法适用于捕捉依赖网络中的传递影响 [33]。在我的情况下，页面排名算法应该在根据维护活动对开源软件库进行分类时捕获每个依赖关系在网络中的汇总维护活动特征。此外，还可以评估不同的影响测量方法来捕捉传递效应，例如度中心性、入度中心性、出度中心性和特征向量中心性 [1]。

5.3 RQ3: 维护活动的预测

我计划扩展和改进现有的预测方法，以考虑直接和传递依赖关系来预测 OSS 库的维护活动。在预测维护活

[8] https://www.npmjs.com/package/escape-string-regexp

动时有两个重要的方面。首先，我想测试不同的机器学习方法（例如 LSTM RNNs），这些方法已被证明能够成功捕捉时间序列数据。其次，我想评估不同时间段如何影响预测性能和置信水平（例如 1 个月、3 个月、6 个月、9 个月、12 个月），类似于 Xia et al. [37] 。

除了预测一个库的项目级维护活动特征外，还应考虑直接和传递依赖的影响，而这一点目前被现有方法所忽视。相比之下，我计划为通过直接和传递链接与正在研究的库相连的所有库预测一组维护活动特征。随后，可以按照第 5.2 节所述进行分类，使用预测的特征来对未来维护活动状态进行分类。

5.4 RQ4: 努力意识的案例研究

我计划在一个案例研究中评估所述方法，重点关注应用程序维护人员的努力意识使用一种工具。该工具必须在工业环境中适用，并提供两种主要功能：首先，它应将 OSS 依赖项分类为混合方法研究中确定的多种维护活动状态之一。其次，它应预测不同时间段内 OSS 依赖项的维护活动。这两种功能都应考虑直接和传递依赖项。

我将尝试从行业内寻找一个愿意进行案例研究的合作伙伴。或者，如果没有行业合作伙伴愿意合作，可以使用应用于行业的 OSS 项目中的依赖数据来进行案例研究，例如 Tensorflow、Jenkins 或 Angular。因此，研究设计独立于数据来源。

案例研究旨在评估使用工具可以将应用程序维护者监控 OSS 依赖项维护活动的努力减少到什么程度。作为基线，我将计算理论上在手动监控所有 OSS 依赖项的维护活动时所需分析的库的数量，包括直接和传递依赖项。该基线可以与基于工具方法的结果进行比较，后者描述了节省工作量的度量标准。

例如，一个包含 30 个直接依赖的系统在收集所有直接和传递依赖时会导致基线为 150 个库。设想，在这 150 个库中有 20 个是可疑的，并且应该由应用程序维护者进行更仔细的分析。如果我的工具报告了其中 20 个可疑库中的 15 个，则提议的工具与手动方法相比可以减少 90%的工作量，召回率为 75%。

我认为，召回率作为评估指标比精确率更重要，因为错过一个正例可能会导致严重的后果。相反，有人可能认为精确率更为重要，以避免给用户带来过多的信息。最终，这取决于具体情境，可以从两个角度来考虑。

为了扩展对 RQ4 的评估，将应用来自其他软件工程主题中更为复杂的努力意识指标，如及时缺陷预测 [5]。此外，努力意识指标可以转化为经济方面，以使我的工作的影响力对公司更加透明。

5.5 有效性威胁及缓解策略

有效性的威胁分为以下四个方面 [34]：

构建效度：为了自动分类开源软件库的维护活动，我将构建一个包含来自不同包管理器的开源软件库维护活动数据的数据集。为了避免在创建数据集时选择相关特征对构念效度构成潜在威胁，我计划定义一个适当的终止条件来添加额外特征，遵循 Nickerson et al. [26] 。

内部有效性：由于采用了混合方法，我相信大多数相关的特征将会被发现，并显示出与维护活动标签的关系。然而，可能存在一些超出范围的方面，它们与维护活动有隐藏的关系，而这些关系以前从未被人想到过。

外部有效性：由于我将仅分析选定包管理器的库，因此研究结果可能不直接适用于其他编程语言。然而，如果在这些生态系统中存在识别到的功能，所应用的方法可以推广到其他编程语言的包管理器。

尽管我的重点是工业应用，这些方法也适用于其他使用 OSS 库进行软件开发的场景（例如，研究）。

可靠性：为了降低对可靠性的威胁，所有必要的复现实验结果的构件都将发布，并且所需的数据将从公开访问的来源（例如 GitHub）收集。

6 当前状态

目前，我正在研究第一个研究问题（RQ1），这包括一项混合方法研究，以收集描述开源软件库维护活动的特征和标签。因此，我在使用文献回顾进行定量研究

时收集了最先进的特征和标签。之后，在定性研究中通过访谈应该产生新的特征和标签的想法。在这些访谈期间，我还旨在探究在软件开发的研究和行业中应用于监控开源软件依赖项维护活动的最新机制、方法和自动化水平。

在确定了一组最终的特征和标签后，我计划定义将用于收集所识别功能维护活动数据的相关流行包管理器库（例如 NPM、PyPI、Maven）。RQ1 的结果，包括特征、标签、数据集的描述性分析、数据集本身以及访谈结果应在单独的出版物中发表。

7 结论

本文旨在总结我的博士学位项目愿景，以识别应用程序中可疑的开源软件依赖项及其维护活动。我相信无论是研究界还是业界都能从我的项目中受益，因为如今开源软件库的采用非常广泛，但对潜在的安全威胁关注却远远不足。我的工作可以减少应用程序维护者的努力，因为只需分析整个库集中的很小一部分。此外，我的工作通过持续使用维护良好的库来支持应用安全，防止出现因缺乏活跃社区支持而导致库受到漏洞影响的情况。为了为其他研究者创造最大的价值，我计划让未来发表的所有成果均可访问。

REFERENCES

[1] Derek Banks, Camille Leonard, Shilpa Narayan, Nicholas Thompson, Brandon Kramer, and Gizem Korkmaz. 2022. Measuring the Impact of Open Source Software Innovation Using Network Analysis on GitHub Hosted Python Packages. In *2022 Systems and Information Engineering Design Symposium (SIEDS)*. IEEE, 110–115.

[2] Veronika Bauer and Lars Heinemann. 2012. Understanding API usage to support informed decision making in software maintenance. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 435–440.

[3] Veronika Bauer, Lars Heinemann, and Florian Deissenboeck. 2012. A structured approach to assess third-party library usage. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 483–492.

[4] Neda Hajiakhoond Bidoki, Gita Sukthankar, Heather Keathley, and Ivan Garibay. 2018. A cross-repository model for predicting popularity in github. In *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 1248–1253.

[5] Peter Bludau and Alexander Pretschner. 2022. Feature sets in just-in-time defect prediction: an empirical evaluation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. 22–31.

[6] Zhe Chen and Daniel M German. 2020. REM: Visualizing the ripple effect on dependencies using metrics of health. In *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE, 61–71.

[7] Jusop Choi, Wonseok Choi, William Aiken, Hyoungshick Kim, Jun Ho Huh, Taesoo Kim, Yongdae Kim, and Ross Anderson. 2022. Attack of the Clones: Measuring the Maintainability, Originality and Security of Bitcoin'Forks' in the Wild. *arXiv preprint arXiv:2201.08678* (2022).

[8] Jamie Cleare and Claudia Iacob. 2018. Gemchecker: Reporting on the status of gems in ruby on rails projects. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 700–704.

[9] Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L Silva. 2020. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. *Information and Software Technology* 122 (2020), 106274.

[10] Russ Cox. 2019. Surviving software dependencies. *Commun. ACM* 62, 9 (2019), 36–43.

[11] Fernando López de la Mora and Sarah Nadi. 2018. An empirical study of metric-based comparisons of software libraries. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. 22–31.

[12] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2–12.

[13] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*. 181–191.

[14] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. 2169–2185.

[15] Christof Ebert. 2008. Open source software in industry. *IEEE Software* 25, 3 (2008), 52–53.

[16] Nikolay Harutyunyan and Dirk Riehle. 2019. Getting started with open source governance and compliance in companies. In *Proceedings of the 15th International Symposium on Open Collaboration*. 1–10.

[17] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering* 27, 4 (2022), 1–41.

[18] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. 2013. Is it all lost? A study of inactive open source projects. In *IFIP international conference on open source systems*. Springer, 61–79.

[19] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 102–112.

[20] Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. 2014. Visualizing the evolution of systems and their library dependencies. In *2014 Second IEEE Working Conference on Software*

Visualization. IEEE, 127–136.

[21] Valentina Lenarduzzi, Davide Taibi, Davide Tosi, Luigi Lavazza, and Sandro Morasca. 2020. Open source software evaluation, selection, and adoption: a systematic literature review. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 437–444.

[22] Xiaozhou Li, Sergio Moreschini, Fabiano Pecorelli, and Davide Taibi. 2022. OSSARA: Abandonment Risk Assessment for Embedded Open Source Components. *IEEE Software* 39, 04 (2022), 48–53.

[23] Xiaozhou Li, Sergio Moreschini, Zheying Zhang, and Davide Taibi. 2022. Exploring factors and metrics to select open source software components for integration: An empirical study. *Journal of Systems and Software* 188 (2022), 111255.

[24] Courtney A McKim. 2017. The value of mixed methods research: A mixed methods study. *Journal of mixed methods research* 11, 2 (2017), 202–222.

[25] Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Aymen Saied, and Bram Adams. 2021. Toward using package centrality trend to identify packages in decline. *IEEE Transactions on Engineering Management* (2021).

[26] Robert C Nickerson, Upkar Varshney, and Jan Muntermann. 2013. A method for taxonomy development and its application in information systems. *European Journal of Information Systems* 22, 3 (2013), 336–359.

[27] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[28] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2020. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering* (2020).

[29] Mike Pittenger. 2016. Open source security analysis: The state of open source security in commercial applications. *Black Duck Software, Tech. Rep* (2016).

[30] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2011. Exploring risks in the usage of third-party libraries. In *of the BElgian-NEtherlands software eVOLution seminar*. 31.

[31] Kristiina Rahkema and Dietmar Pfahl. 2022. SwiftDependencyChecker: Detecting Vulnerable Dependencies Declared Through CocoaPods, Carthage and Swift PM. In *2022 IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft)*. IEEE, 107–111.

[32] Donald J Reifer, Victor R Basili, Barry W Boehm, and Betsy Clark. 2003. Eight lessons learned during COTS-based systems maintenance. *Ieee Software* 20, 5 (2003), 94–96.

[33] Ian Rogers. 2002. The Google Pagerank algorithm and how it works. (2002).

[34] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14, 2 (2009), 131–164.

[35] Sefa Eren Sahin, Kubilay Karpat, and Ayse Tosun. 2019. Predicting Popularity of Open Source Projects Using Recurrent Neural Networks. In *IFIP International Conference on Open Source Systems*. Springer, 80–90.

[36] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 644–655.

[37] Tianpei Xia, Wei Fu, Rui Shu, Rishabh Agrawal, and Tim Menzies. 2022. Predicting health indicators for open source projects (using hyperparameter optimization). *Empirical Software Engineering* 27, 6 (2022), 1–31.

[38] Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo. 2020. Why reinventing the wheels? An empirical study on library reuse and re-implementation. *Empirical Software Engineering* 25, 1 (2020), 755–789.