软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(10):4493−4509 [doi: 10.13328/j.cnki.jos.006959]

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

面向 PyPI 生态系统的漏洞影响范围细粒度评估方法*

王梓博[1,2], 贾相堃[1], 应凌云[3], 苏璞睿[1]

[1](中国科学院 软件研究所, 北京 100190)

[2](中国科学院大学, 北京 100049)

[3](奇安信技术研究院, 北京 100044)

通信作者: 贾相堃, E-mail: xiangkun@iscas.ac.cn

摘　要: Python 语言的开放性和易用性使其成为最常用的编程语言之一. 其形成的 PyPI 生态系统在为开发者提供便利的同时, 也成为攻击者进行漏洞攻击的重要目标. 在发现 Python 漏洞之后, 如何准确、全面地评估漏洞影响范围是应对 Python 漏洞的关键. 然而当前的 Python 漏洞影响范围评估方法主要依靠包粒度的依赖关系分析, 会产生大量误报; 现有的函数粒度的 Python 程序分析方法由于上下文不敏感等导致存在准确性问题, 应用于实际的漏洞影响范围评估也会产生误报. 提出一种基于静态分析的面向 PyPI 生态系统的漏洞影响范围评估方法 PyVul++. 首先构建 PyPI 生态系统的索引, 然后通过漏洞函数识别发现受漏洞影响的候选包, 进一步通过漏洞触发条件验证漏洞包, 实现函数粒度的漏洞影响范围评估. PyVul++改进了 Python 代码函数粒度的调用分析能力, 在基于 PyCG 的测试集上的分析结果优于其他工具 (精确率 86.71%, 召回率 83.20%). 通过 PyVul++对 10 个 Python CVE 漏洞进行 PyPI 生态系统 (385 855 个包) 影响范围评估, 相比于 pip-audit 等工具发现了更多漏洞包且降低了误报. 此外, 在 10 个 Python CVE 漏洞影响范围评估实验中, PyVul++新发现了目前 PyPI 生态系统中仍有 11 个包存在引用未修复的漏洞函数的安全问题.

关键词: PyPI 生态系统; 漏洞影响范围; 函数粒度评估; 静态分析

中图法分类号: TP311

中文引用格式: 王梓博, 贾相堃, 应凌云, 苏璞睿. 面向PyPI生态系统的漏洞影响范围细粒度评估方法. 软件学报, 2024, 35(10): 4493–4509. http://www.jos.org.cn/1000-9825/6959.htm

英文引用格式: Wang ZB, Jia XK, Ying LY, Su PR. Fine-grained Assessment Method of Vulnerability Impact Scope for PyPI Ecosystem. Ruan Jian Xue Bao/Journal of Software, 2024, 35(10): 4493–4509 (in Chinese). http://www.jos.org.cn/1000-9825/6959.htm

Fine-grained Assessment Method of Vulnerability Impact Scope for PyPI Ecosystem

WANG Zi-Bo[1,2], JIA Xiang-Kun[1], YING Ling-Yun[3], SU Pu-Rui[1]

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

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

[3](QI-ANXIN Technology Research Institute, Beijing 100044, China)

Abstract: The openness and ease-of-use of Python make it one of the most commonly used programming languages. The PyPI ecosystem formed by Python not only provides convenience for developers but also becomes an important target for attackers to launch vulnerability attacks. Thus, after discovering Python vulnerabilities, it is critical to deal with Python vulnerabilities by accurately and comprehensively assessing the impact scope of the vulnerabilities. However, the current assessment methods of Python vulnerability impact scope mainly rely on the dependency analysis of packet granularity, which will produce a large number of false positives. On the other hand, existing Python program analysis methods of function granularity have accuracy problems due to context insensitivity and produce false positives

* 基金项目: 国家自然科学基金 (62232016, 62102406); 中国科学院青年创新促进会项目
收稿时间: 2023-01-14; 修改时间: 2023-03-15; 采用时间: 2023-04-19; jos 在线出版时间: 2023-09-13
CNKI 网络首发时间: 2023-09-14

when applied to assess the impact scope of vulnerabilities. This study proposes a vulnerability impact scope assessment method for the PyPI ecosystem based on static analysis, namely PyVul++. First, it builds the index of the PyPI ecosystem, then finds the candidate packets affected by the vulnerability through vulnerability function identification, and confirms the vulnerability packets through vulnerability trigger condition. PyVul++ realizes vulnerability impact scope assessment of function granularity, improves the call analysis of function granularity for Python code, and outperforms other tools on the PyCG benchmark (accuracy of 86.71% and recall of 83.20%). PyVul++ is used to assess the impact scope of 10 Python CVE vulnerabilities on the PyPI ecosystem (385855 packets) and finds more vulnerability packets and reduces false positives compared with other tools such as pip-audit. In addition, PyVul++ newly finds that 11 packets in the current PyPI ecosystem still have security issues of referencing unpatched vulnerable functions in 10 assessment experiments of Python CVE vulnerability impact scope.

Key words: PyPI ecosystem; vulnerability impact scope; function granularity assessment; static analysis

Python 语言因为其开放性、易用性等特点已经成为最常用的编程语言之一[1], TensorFlow[2]、PyTorch[3]、Flask[4]等大型软件项目都采用了 Python 作为主要开发语言. Python 语言拥有大量功能强大且便捷的开源 Python 包 (package) 支持, 形成了 Python 包索引 (Python package index, PyPI) 生态系统 (如图 1 所示)[5]. 软件包开发者可以注册上传 Python 包到 PyPI 仓库, 软件包使用者可以从 PyPI 仓库下载使用需要的 Python 包, PyPI 的管理者则会对生态系统内的包进行检查管理. 目前, PyPI 仓库作为 Python 包的官方储存库, 已经包含了超过 386 000 个项目[5].

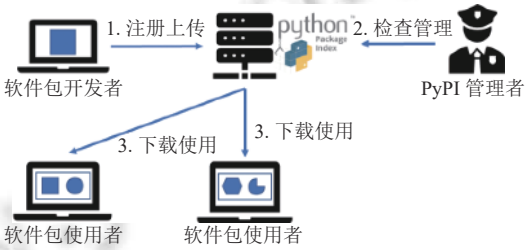

图 1 PyPI 生态系统示意图

开放的生态系统在提供开发者便利的同时, 也为攻击者提供了便利, 成为攻击者发动漏洞攻击的重要目标[6−8]. 根据 Alfadel 等人的研究, PyPI 生态系统正遭受 Python 漏洞的严重威胁[9]. 另一方面, 软件包之间形成的广泛且复杂的依赖关系, 使得 Python 漏洞不再局限于发现漏洞的 Python 包本身, 还影响到在软件供应链上具有依赖关系的 Python 包和应用. 如图 2 所示, 应用 1 受到包 1 中的漏洞影响, 可能遭受 0-day 攻击. 应用 2 因为调用了包 2, 间接调用了包 1, 也受到包 1 漏洞的影响. 糟糕的是, 应用 2 因为包依赖的复杂性可能未发现受到包 1 影响或延迟修复漏洞, 从而遭受 *n*-day 攻击. 然而, 当前的漏洞报告 (如 CVE 报告[10]) 只有发现漏洞的 Python 包信息, 无法提示开发者还有哪些 Python 包受到影响. 在发现 Python 漏洞之后, 如何准确、全面地评估漏洞影响范围是应对 Python 漏洞的关键.

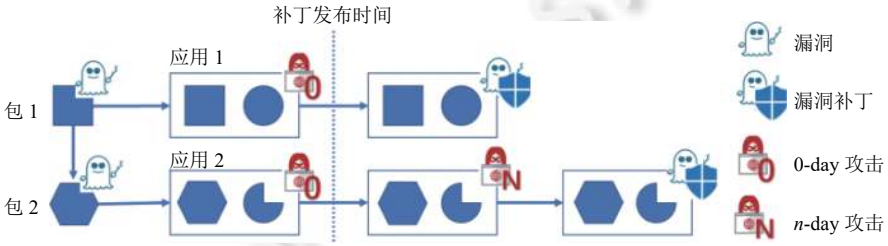

图 2 软件供应链场景下的漏洞攻击示意图

当前的 Python 漏洞影响范围评估方法主要是基于包依赖关系实现的, 例如, pip-audit 根据 Python 项目配置文件 (如 requirement 文件等) 实现对引入包 (import package) 的依赖分析和已知漏洞扫描[11]. 然而, 如果应用引入了漏洞包但是没有调用漏洞函数, 那么该应用实际是不受漏洞影响的, 即包粒度的评估会产生误报. Salis 等人提出

的 Python 程序调用图 (call graph) 构建方法 PyCG 可以进行函数粒度的分析[12]. 但是 PyCG 在处理上下文敏感的函数调用时存在分析不准确的问题, 同时 PyCG 需要下载所有具有依赖关系的 Python 包进行本地分析, 影响分析效率. 因此, Python 漏洞影响范围评估需要准确、高效的函数粒度分析.

针对当前方法中存在的不足, 本文提出了一种基于静态分析的面向 PyPI 生态系统的漏洞影响范围评估方法 PyVul++, 将包粒度分析细化为函数粒度分析, 其流程如图 3 所示. 首先, PyVul++通过对 PyPI 仓库中所有包的全量依赖关系分析, 构建 PyPI 生态系统 (PyEco) 的索引. 当发现 Python 漏洞之后, 根据漏洞报告的信息描述可以在 PyEco 索引中定位报告中提到的漏洞包. 进一步, 基于 PyEco 索引, 通过对漏洞包中漏洞函数的识别实现对 PyPI 生态系统中可能受到漏洞影响的包的快速分析, 得到候选的漏洞包. 最后, 基于漏洞 PoC 中漏洞触发条件的提取, 确认真正受到漏洞影响的漏洞包, 实现 Python 漏洞在 PyPI 生态系统的影响范围的准确评估.

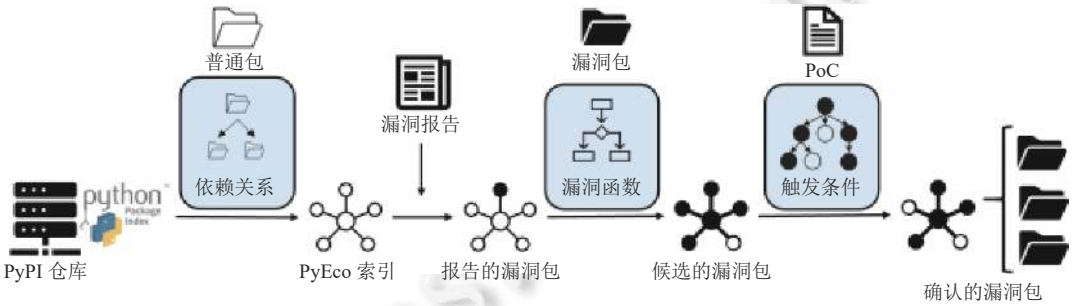

图 3 PyVul++流程图

在实现 PyVul++的过程中, 本文重点解决了以下技术挑战. (1) 针对大规模 Python 包分析的挑战, 本文提出了基于包反向依赖关系构建 PyPI 生态系统索引 (PyEco 索引) 的方法, 能够有效减少需要检查的包范围; (2) 针对 Python 代码函数粒度调用分析时不准确的问题, 本文提出了上下文敏感的函数调用分析方法和命名空间链摘要方法, 提高了 Python 代码函数粒度分析的准确性; (3) 针对缺少漏洞触发条件分析的问题, 本文提出了基于 AST 分析的关键变量提取方法, 进一步降低了漏洞影响范围评估的误报.

为了验证 PyVul++的有效性, 本文 (1) 完成了对 PyPI 生态系统 (385 855 个包) 的分析, 构建了 54 654 个包的反向依赖关系, 作为后续分析的基础; (2) 基于 PyCG 提供的 micro-benchmark 和 macro-benchmark 测试了 PyVul++函数粒度分析的准确性, 分析结果优于 PyCG 在内的其他工具 (PyCG/code2flow/pyan); (3) 基于 10 个 Python CVE 漏洞测试了面向 PyPI 生态系统的 Python 漏洞影响范围评估的准确性, 相比于 pip-audit 等其他工具, 能够发现更多的 Python 漏洞包并降低误报. 此外, 通过 PyVul++的分析, 本文新发现了当前 PyPI 生态系统中 11 个包仍存在引用 Python 漏洞包的安全问题.

本文的贡献如下: (1) 提出了准确的 Python 代码函数调用分析方法和 Python 漏洞存在性验证方法, 设计了面向 PyPI 生态系统的漏洞影响范围细粒度评估工具 PyVul++, 提高了漏洞影响范围评估的准确性. (2) 完成了对 PyPI 生态系统中 10 个 Python CVE 漏洞的影响范围评估, 和 pip-audit 等工具对比验证了 PyVul++能够更准确、全面地发现漏洞包. (3) 本文在 10 个 Python CVE 漏洞影响范围评估实验中, 新发现了目前 PyPI 生态系统中仍有 11 个包存在引用未修复的漏洞函数的安全问题, 并进行了上报. PyVul++工具及实验数据将开源, 方便后续研究.

本文第 1 节介绍 PyPI 生态系统安全研究、Python 程序分析与漏洞检测的相关研究. 第 2 节通过示例分析介绍研究思路. 第 3 节详细介绍 PyVul++的设计及关键技术. 第 4 节通过实验验证 PyVul++的有效性. 最后总结全文并讨论未来工作.

1 相关工作

1.1 PyPI 生态系统安全研究

软件供应链安全已经成为网络空间安全领域的研究热点, 国内外研究者对当前软件供应链的现状和研究进展

进行了总结[13−17]. Python、Ruby、JavaScript 等能够分享程序或代码库的编程语言, 因为其开放性更容易遭受到软件供应链攻击, 也吸引了研究者对编程语言生态系统进行了针对性的研究[18−22].

Python 的 PyPI 生态系统主要面临的供应链安全问题包括投毒攻击、软件漏洞、证书不一致、可维护性风险等. 针对投毒攻击, Vu 等人利用包名的 Levenshtein 距离检测可能存在包名抢注等问题的可疑包[23]; Liang 等人进一步结合代码特征提出了 pip 投毒检测工具 PDD[24]; Duan 等人提出的 MALOSS 结合了元数据分析、静态分析和动态分析技术, 对包括 PyPI 生态系统在内的多个生态系统进行了定量测量[25]. 针对软件漏洞问题, Ruohonen 讨论了在 Web 应用中 Python 包的漏洞问题, 并通过对漏洞包的发行时间序列分析指出, Python 包出现漏洞的概率分布具有马尔科夫属性[26]; Alfadel 等人则研究了 Python 漏洞的传播和寿命, 指出 PyPI 生态系统和 NPM 生态系统在漏洞揭露和修复政策上的不同[9]. 对于 Python 等开源软件中的证书不一致问题, Xu 等人提出了基于学习的方法识别许可条款, 并采用概率上下文无关的语法进行权利义务推断[27]. Wang 等人[28]、Mukherjee 等人[29]、Cao 等人[30]的研究则旨在提供更好的包依赖关系管理方法, 相关研究可以应对 Python 项目构建时面临的可维护性风险. 本文重点关注软件漏洞对 PyPI 生态系统造成的安全威胁.

1.2 Python 程序分析与漏洞检测

Python 语言的复杂性使得研究者需要提出针对性的程序分析方法, 例如针对 Python 程序的动态切片方法[31]、静态类型推断方法[32]、动态类型推断方法[33]等. 其中, Python 项目的包依赖关系分析是开展 Python 项目构建和维护、发现配置错误或漏洞检测的基础. Wang 等人借助了 pip 工具在构建 Python 项目时获取包依赖信息[28]; Mukherjee 等人则提出了通过错误日志分析不断构建依赖关系[29]; Cao 等人提出了通过配置文件 (如 requirement 文件) 提取依赖关系[30]. 相比于包粒度的依赖关系分析, Salis 等人提出了 Python 程序函数调用图生成方法 PyCG[12], 能够支持函数粒度的依赖分析方法. 本文希望实现漏洞影响范围的细粒度评估, 因此需要函数粒度的依赖分析. 在基于 PyCG 的实验中, 本文发现 PyCG 存在分析不准确和开销大的问题, 因此本文提出了对 PyCG 的改进方案.

Python 语言编写的包或程序和 C/C++语言编写的程序一样, 也会产生漏洞并被攻击者利用. 根据 Alfadel 等人的统计, Python 漏洞数量呈现逐年上升的趋势[9]. 在漏洞检测方面, Jiang 等人针对 Python 依赖的虚拟机 CPython 提出了漏洞检测工具 PyGuard[34]. Xu 等人提出了基于执行记录进行变量符号化并求解的 Python 缺陷发现方法[35]. Ma 等人提出了基于静态应用安全测试 (static application security testing, SAST) 进行 Python 代码审计的方法[36]. Fromherz 等人提出了利用抽象解释 (abstract interpretation) 进行静态值集分析并发现运行时错误的方法[37]. Bagheri 等人[38]、Wartschinski 等人[39]、彭双和等人[40]则提出了借助自然语言处理 (natural language processing, NLP) 等机器学习方法表示 Python 代码, 并进行漏洞检测. 然而, 上述相关工作没有关注到漏洞在 PyPI 生态系统中的影响范围. 在供应链场景下, PyPI 仓库管理者或 Python 程序开发者, 都需要确定哪些 Python 包受到漏洞的影响并进行处理. pip-audit 可以根据 Python 项目配置文件 (如 requirement 文件等) 实现对引入包 (import package) 的依赖分析和已知漏洞扫描[11]. 然而其包粒度的评估会产生误报. 因此, 本文聚焦 Python 漏洞发现后、面向 PyPI 生态系统的漏洞影响范围评估, 希望把包粒度的评估推进到函数粒度, 更好地提升 Python 程序质量、维护 PyPI 生态系统安全.

2 示例分析与研究思路

本文通过 Python 加密库 PyCryptodome 的整数溢出漏洞 CVE-2018-15560 说明本文针对的问题及研究思路. 图 4 是 CVE-2018-15560 漏洞的 PoC (proof-of-concept) 代码分析结果. 整数漏洞存在于 encrypt 函数中, 即 encrypt 函数是漏洞函数. PoC 首先调用 Crypto. Cipher.AES 模块的 new 函数 (图 4(a) 第 4 行) 创建一个“_create_cipher”对象 (实际调用路径如图 4(a)、图 4(b)、图 4(c) 所示). 然后调用该对象的 encrypt 函数 (图 4(a) 第 5 行) 触发漏洞 (实际调用路径如图 4(a)、图 4(d) 所示).

图 5 是美国国家计算机通用漏洞数据库 (national vulnerability database, NVD) 中 CVE-2018-15560 的漏洞报告. 报告中对漏洞影响范围的描述局限于 Python 包本身 (如图 5 中红框 2 所示, 版本小于 3.66 的 PyCryptodome 包受到影响), 并未列举调用 PyCryptodome 包的其他 Python 包或 Python 项目. 同时, 漏洞威胁程度描述也是针对

包本身给出, (如图 5 中红框 1 所示, 该漏洞的 CVSS Version 2.0 评分为 5.0 Medium). 然而实际上, 众多项目直接或间接调用了 PyCryptodome 包进行密码运算, 该 Python 包的月下载量超过 2 900 000[41], 实际影响范围在供应链场景下扩大了. 因此, 本文主要研究 Python 漏洞发现之后, 如何在 PyPI 生态系统中评估漏洞影响范围, 进一步准确检测 Python 项目是否受到漏洞影响.

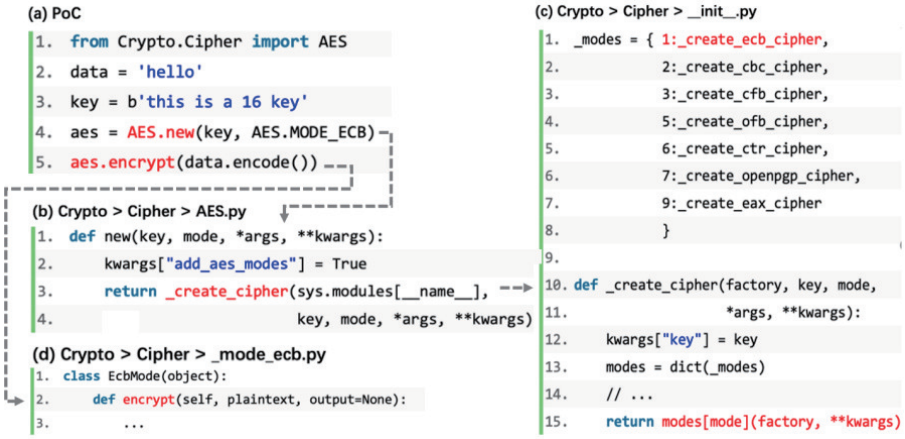

图 4 CVE-2018-15560 漏洞 PoC 代码分析

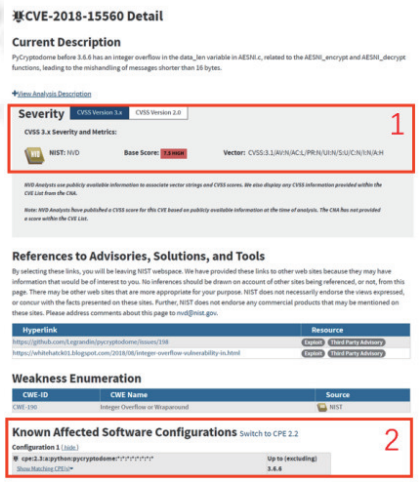

图 5 CVE-2018-15560 漏洞报告

本文面临的主要挑战是准确性问题. 之前的方法主要通过分析 requirement 文件或源码 import 关键字提取包依赖关系, 如果检测到依赖 PyCryptodome 包即报告存在漏洞. 然而通过上述漏洞 PoC 代码分析结果可以发现, 漏洞的触发需要调用 new 函数和 encrypt 函数. 而且, 由图 4(c) 中的代码可知, new 函数会根据第 2 个参数“AES.MODE_XXX”返回不同的对象 (图 4(c) 第 15 行), 代表不同的加密模式. 不同的对象在后续调用 encrypt 函数时也是不同的, CVE-2018-15560 漏洞仅在“AES.MODE_ECB”模式下才能被触发. 因此, 漏洞及其影响范围评估需要从包粒度细化到函数粒度, 并考虑漏洞触发条件. 尽管有一些研究工作能够支持 Python 程序函数级别分析 (如 PyCG), 但是其仍存在准确性方面的不足.

3 面向 PyPI 生态系统的漏洞影响范围细粒度评估方法 PyVul++

本文提出了一种面向 PyPI 生态系统的漏洞影响范围细粒度评估方法 PyVul++, 其流程如前文图 3 所示. 首先,

为了支持面对 PyPI 生态系统的大规模评估, 本文提出了通过包依赖分析构建整个生态系统索引 (PyEco 索引) 的方案. 因为 Python 包的依赖关系是相对固定的, 通过对 PyPI 生态系统中所有包进行预分析, 可以在发现漏洞包之后快速反应. 值得注意的是, 为了更好地在发现漏洞包之后对整个 PyPI 生态系统进行扫描, 相比于之前通常建立的基于调用关系的包依赖图, 本文建立的 PyEco 索引是基于被调用关系的包反向依赖图 (详见第 3.1 节).

之后, 为了提高漏洞影响范围评估的准确率, 本文提出了基于漏洞函数识别的函数粒度影响范围评估方法. 虽然利用 PyEco 索引可以在漏洞报出后进行粗粒度的漏洞影响范围评估, 但是存在包依赖关系并不能代表受到漏洞影响. 因此, PyVul++通过对漏洞 PoC 文件的代码静态分析, 建立了 Python 包内模块的函数调用图, 进而在函数调用图中识别 PoC 中涉及的漏洞函数 (详见第 3.2 节).

最后, 为了进一步消除误报, 本文提出了基于触发条件提取的漏洞存在性验证方法. 虽然通过对漏洞函数识别可以定位漏洞函数, 但是函数内漏洞点的触发可能还需要其他条件. 例如, 当函数参数能够影响控制流分支的走向时, 会导致程序执行不同的功能, 可能触发漏洞点也可能执行正常程序代码. 因此, 本文通过对 PoC 中漏洞函数内部关键变量的提取, 对漏洞的存在性进行验证 (详见第 3.3 节).

3.1 基于依赖关系分析的 PyPI 生态系统索引构建

在建立包依赖关系时, 之前的方法大多采用基于调用关系的正向依赖[28−30]. 当发现漏洞包后, 通过建立的包正向依赖关系, 扫描本地是否存在漏洞包及相应的版本, 或者本地模块是否引用了漏洞包. 形式化地, 如果在模块 B 中通过 import A 引入 A 模块/包, 而模块 C 中通过 import B 引入 B 模块/包, 则 B 正向依赖于 A, C 正向依赖于 B, 记为:

$$C \in B \in A.$$

此时, 如果基于被调用关系, 则 A 反向依赖于 B, B 反向依赖于 C, 记为:

$$A \ni B \ni C,$$

其中, $\in$ / $\ni$ 符号的方向代表包的依赖关系, 同时也表达了包的上下游关系. 当包 A 出现漏洞后, 需要检测 A 的下游包 (B/C) 是否存在漏洞. 在通过正向依赖关系寻找下游包时, 需要检查 PyPI 仓库中所有包是否和 A 具有依赖关系. 而通过预分析建立包 A 的反向依赖关系, 则可以直接找到下游包并进行漏洞检测. 因此, 本文选择建立反向依赖图更好地支持漏洞影响范围大规模评估.

具体来讲, 本文的 PyVul++对 PyPI 仓库 (PyPI repository) 中所有的包进行依赖信息提取和依赖关系分析. 每个上传到 PyPI 仓库的包都会指明安装自身所需要的依赖包, 该信息可以通过配置文件 (config files, 如“requirement.txt”或“setup.py”) 获取. 因此本文为每一个包建立数据结构 PACKAGE_INDEX 结构体, 包括 Python 包的信息、该包相关的正向依赖关系 PyDep 和反向依赖关系 PyReDep. 基于依赖关系分析的 PyPI 生态系统索引构建算法伪代码如算法 1 所示.

算法 1. PyPI 生态系统索引构建算法.

```
(1) Function ExtractDep(PyPI repository)
(2)     for package ∈ PyPI repository do
(3)         PACKAGE_INDEX pack
(4)         pack.PyDep = Analysis(config files)
(5) Function BuildReDep(pack, exclude)
(6)     PyReDep = pack.PyReDep, ExcludeSet = exclude
(7)     if pack ∉ ExcludeSet then
(8)         ExcludeSet← {pack} ∪ ExcludeSet
(9)     for p∈ pack.PyDep() do
(10)        if p ∉ ExcludeSet then
(11)            pack.PyReDep← pack.PyReDep∪ BuildReDep(p, ExcludeSet ∪ {p})
(12) return pack.PyReDep
```

首先提取每个包的依赖信息并储存在 pack.PyDep 中 (Function ExtractDep). 然后再次遍历所有包, 并根据其 pack.PyDep 对 PyPI 仓库中的其他包进行搜索, 如果发现当前包被其他包应用, 即加入 pack.PyReDep 中 (Function BuildReDep). 最终, 通过递归的方法构建整个 PyPI 生态系统中包的反向依赖关系, 即 PyPI 生态系统索引 (PyEco 索引). PyEco 索引将作为作为后续进行 Python 漏洞影响范围评估的基础, 同时其可以随 PyPI 仓库中包的更新而更新.

值得注意的是, 为了避免嵌套引用而引起的循环 (例如 A 的下游包引用了 A), 本文在对每个节点的迭代分析中排除该节点本身 (算法中涉及 ExcludeSet 的操作). 如果出现 A 的多个下游包反向引用相同包的情况, 本文会分别计算, 因为在漏洞函数向下游包传递的过程中可能会引入新的函数, 因此同一个包在不同包的反向检测中可能存在不同的漏洞函数, 即对于不同的两个包, 反向依赖的包可以是相同的.

3.2 基于漏洞函数识别的函数粒度影响范围评估

为了更准确地评估漏洞影响范围, 本文将原有的包粒度评估细化为函数粒度评估, 即通过对包中所有模块构建函数调用图 (CG), 检测包中是否存在对漏洞函数的调用. 然而 Python 语言由于其存在的特性, 静态分析难度大 (例如, 如何抽象 Lambda 函数, 如何计算迭代器, 如何解析装饰器等). 最近的研究工作 PyCG 能够针对 Python 静态生成函数调用图, 其采用指针分析构建赋值关系图 (多轮迭代指针变量分析直到求解出不动点, 详见文献 [12]), 之后生成函数调用图. PyCG 解决了很多动态特性模拟问题, 提高了函数调用图的准确性. 然而, 该工作仍然存在以下不足.

首先 PyCG 采用的是上下文不敏感的过程间分析, 对函数进行分析时没有区分调用点, 导致变量在函数内部被合并计算, 影响最终生成的函数调用图精度. 例如在图 6 的代码片段中, x1 和 x2 分别为两次调用 NewX 返回的结果 (如图 6 中第 12、13 行). 在上下文不敏感的分析中, 无法区分两次不同的 NewX 调用 (参数分别为 n1 和 n2). 这导致在构建函数调用图时, 尽管参数 p 的两次调用取值不同 (如图 6 第 7 行), 但在函数内部被合并并赋值给 x.f, 即 x.f 同时指向 n1 和 n2 (如图 6 第 9 行), 而 x1 和 x2 都指向 NewX 内部的 x (如图 6 第 12、13 行). 最终, 因为无法区分 n1、n2 导致在分析 n1.vul()(如图 6 第 17 行) 时, 得到 n1.vul() 既调用了 A.vul() 也调用了 B.vul() 的错误结果.

```
1.  class A(Base):
2.      def func(self):
3.          // vulnerable function
4.  class B(Base):
5.      def func(self):
6.          // normal function
7.  def NewX(p):
8.      x = X()
9.      x.f = p
10.     return x
11. n1, n2 = A(), B()
12. x1 = NewX(n1)  //  x1.f = n1
13. x2 = NewX(n2)  //  x2.f = n2
14. //  n1 = A(), n2 = B()
15. n1, n2 = x1.f, x2.f
16. //  A.vul()
17. n1.vul()
```

图 6 函数调用示例

PyCG 的另一个问题是缺少对第三方库的处理. PyCG 仅会对与待分析包在同一目录下的第三方依赖包, 通过 importlib.import_module 的方式导入分析, 因此对安装在本地 Python 第三方库目录下的第三方依赖包缺乏导入分析的能力, 会导致数据流在遇到外部引用包时中断, 这在引用关系复杂的 PyPI 生态中会导致很低的召回率. 其次, 如果手工将依赖的第三方包置于待分析包所在目录, PyCG 虽然会进入第三方库进行指针分析, 但这也会产生额外的分析开销. 尽管这种处理方式在构建函数调用图时更精确, 但当引用深度大时开销是很大的. 在实际的漏洞影响范围评估中, 本文发现包之间的依赖复杂且深度大, 导致漏洞影响范围评估资源消耗过大.

为了解决第 1 个问题, 本文提出的 PyVul++在 PyCG 的基础上引入了上下文敏感的分析方法. PyCG 是借助抽象语法树 (AST) 实现的, 在 AST 上 PyCG 抽象了 Python 源代码的语法结构, 保留了执行程序所需要的各种信息, 去掉了不必要的标点和分隔符, 同时树的存储结构也便于对其进行多次遍历. 通过在 AST 上的多次遍历实现对赋值语句的约束求解不动点, 从而得出每个指针变量的取值[12]. 本文的改进具体来讲是将函数调用点处的代码行号作为上下文信息, 在抽象语法树的遍历环节增加对 Call 节点上下文信息的处理. 对于 if 和 while, 本文依次遍历其“test”“body”“orelse”节点, 根据节点的具体 ast 结构进行下一步的分析. 对于闭包, 本文记录外层函数调用点的上下

文信息, 作为闭包的上下文. 例如在上面的例子中, 在第 10 行调用 NewX 时, 传递的参数为带上行号的 n1, 这里用 10:n1 表示 (如图 7 所示). 除此之外, 本文还对 PyCG 进行了一些其他的优化, 包括对内置函数的识别, 例如列表, 字典等数据结构的内置函数等; 在常量判定中除了 int/str/float, 还增加了 ast.Constant 判断; 以及对常见第三方库常见函数的解析, 例如“re.findall”“json.loads”“os.path.abspath”等. 该部分优化有利于更准确地处理 Python 代码分析 (详见第 4.4 节研究问题 2).

```
1.  def NewX(p):        //  10:p = n1,  11:p = n2
2.      x = X()         //  10:x = 10:X(),  11:x = 11:X()
3.      x.f = p         //  10:x.f = 10:p,  11:x.f = 11:p
4.      return x        //  10:NewX.return = 10:x, 11:NewX.return = 11:x
5.  n1, n2 = A(), B()
6.  x1 = NewX(n1)       //  x1 = 10:NewX.return
7.  x2 = NewX(n2)       //  x2 = 11:NewX.return
8.  n1 = x1.f           //  n1 = x1.f = 10:NewX.return.f = 10:x.f = 10:p = n1
9.  n2 = x2.f           //  n2 = x2.f = 11:NewX.return.f = 11:x.f = 11:p = n2
10. n1.vul()            //  A.vul()
```

图 7　PyVul++的函数调用处理示意图

为了解决第 2 个问题, 本文提出的 PyVul++针对 Python 包中存在外部调用第三方库的情况, 提出了命名空间链摘要 (namespace abstraction) 方法. 由于漏洞影响范围评估的方法是找到所有的函数调用并与漏洞函数进行匹配, 因此只需要保证漏洞函数和函数调用以相同的方式表示即可. 具体来讲, 本文不进入第三方库进行分析, 而是将外部调用函数过程按照命名空间标记为模块和函数名构成的调用链. 这种处理方式虽然丢失了外部调用函数的内部细节, 但是保留了该函数调用信息, 可以支撑漏洞函数匹配. 以第 2 节的示例 (CVE-2018-15560, 如图 4) 为例, 在缺少 Crypto 源码的情况下, PyCG 无法识别两个函数调用的函数定义 (图 4(a) 的第 4、5 行), 导致对该 PoC 的分析结果为空. 本文通过对无源码的第三方库进行命名空间链摘要, 可以得到漏洞函数为 Crypto.Cipher.AES.new.encry, 在后续漏洞识别和漏洞影响范围评估时即以此进行匹配. 由于对 POC 和源码的分析均为同一工具, 因此可以自动化实现调用链的生成和匹配. 基于漏洞函数识别的函数粒度影响范围评估算法伪代码如算法 2 所示.

算法 2. 函数粒度影响范围评估算法.

```
(1) Function VulScopeDetect(VulPackage)
(2)     VulFuncSet = {VulFunc}, VulScope = {VulPackage}
(3)     for pack ∈ VulPackage.PyReDep() do
(4)         Pack.CG = AnalysisCG(pack)
(5)         VulSet = Pack.CG.Search(VulFuncSet)
(6)         if (VulSet != ∅) then
(7)             VulScope = VulPackageSet ∪ {pack}
(8)             VulFuncSet = VulFuncSet ∪ VulSet
(9) return VulScope
```

在进行漏洞影响范围评估时, PyVul++首先对漏洞包 (VulPackage) 生成 call graph, 并标记漏洞函数 (VulFunc). 然后对于待评估的 Python 包生成 call graph, 并根据漏洞函数标记进行匹配. 如果匹配则说明该 Python 包存在对漏洞函数的调用, 即受到漏洞影响.

值得注意的是, 在检测过程中, 漏洞函数被调用时可能存在函数别名或间接调用的情况. 例如图 8 的 CVE-2022-24303 示例, 漏洞函数位于包 pillow 中的 Image 模块, 通过调用 open (图 8(a) 第 3 行) 和 show (图 8(a) 第 4 行) 进行触发. 在进行漏洞检测时, 发现 sokort 包存在该漏洞函数的调用 (如图 8(b) 中的 show_plot 函数). 因此如果其他包存在对 show_plot 的调用也能触发漏洞, 需要对包 sokort 的下游包进行评估, 并把 show_plot 加入作为漏洞函数集合 (将模块名连接调用链作为新的漏洞函数).

```python
from PIL import Image
im = Image.open(
    r"/var/lib/app-info/icons/ubuntu-focal-updates-universe/48x48/inkscape_inkscape.png")
im.show()
```

(a) PoC of CVE-2022-24303

```python
class PILPresenter(Presenter):
    def show_plot(self, images: list):
        for an_image in images:
            image = PILImage.open(f"./{an_image['path']}")
            image.show()
```

(b) sokort>_presenter

图 8 CVE-2022-24303 示例

3.3 基于触发条件提取的漏洞存在性验证

在确定了漏洞函数存在之后, 本文进一步提出了验证函数调用是否能触发漏洞逻辑. 事实上, 并非所有对漏洞函数的调用都能真正触发漏洞, 还需要在调用时提供符合一定条件的参数, 使得漏洞函数按照漏洞路径运行, 即存在漏洞触发条件. 这种方法的理论依据是, 漏洞 PoC 包含一组引导漏洞函数走向漏洞点的参数变量, 如果正在分析的软件包的漏洞函数在相同变量上取值一致, 则该函数也会走到该漏洞点. 本文通过对漏洞 PoC 中漏洞函数内部的关键变量 (如分支条件, 函数参数等) 分析, 实现对漏洞触发的进一步验证. 关键变量提取算法如算法 3 所示.

算法 3. 关键变量提取算法.

(1) **Function** KeyParaExtract(PythonFiles)
(2) para = ∅
(3) **for** PythonFile ∈ PythonFiles **do**
(4) **for** func, args ∈ PythonFile **do**
(5) **for** ast.node ∈ func **do**
(6) **if** func.child.node ∈ (ast.If, ast.While, ast.IfExp) **then**
(7) para ← para ∪ ExtractPointTo(func.child.node)
(8) **for** arg ∈ args **do**
(9) **if** ParaVerify(arg, para) **then**
(10) iargs ← iargs ∪ arg
(11) **return** iargs

Function KeyParaExtract 的重点是在 AST 的遍历过程中对 3 种节点 (即 ast.If, ast.While, ast.IfExp) 进行处理. 首先通过 ExtractPointTo 对 if, while, ifexp 这 3 种类型的函数子节点 (func.child.node) 指向的值和类型进行提取, 作为备选变量集 (para). 之后, 通过 ParaVerify 比较变量集和函数参数 (args), 将具有指向关系的参数作为关键变量 (iargs) 输出, ParaVerify 是一个验证变量是否具有指向关系的函数.

在进行漏洞触发条件验证的过程中, 首先对漏洞 PoC 文件调用 Function KeyParaExtract 进行分析, 得到 PoC 文件的漏洞触发条件, 即 VulCondition. 然后对 PyVul++的前一步骤中分析得到的 VulScope 中的 Python 包进行分析, 提取 VulScope 中包的关键变量, 和从 PoC 文件提取到的 VulCondition 进行对比, 判断该 Python 包是否存在 PoC 中的漏洞. 漏洞存在性验证的算法伪代码如算法 4 所示.

算法 4. 漏洞存在性验证算法.

(1) **Function** VulVerify(PoC, VulScope)
(2) VulCondition = KeyParaExtract(PoC)
(3) **for** package in VulScope **do**
(4) **if** KeyParaExtract(package) not contains VulCondition
(5) VulScope = VulScope – {package}
(6) **return** VulScope

本文的方案是基于 PoC 所给出的漏洞触发路径去验证 PyPI 包的漏洞触发情况, 而漏洞点的触发可能不止有一条路径, 因此可能存在漏报的情况.

4 实验与分析

4.1 实验设计

本文设计了 4 个方面的实验评估 PyVul++的有效性, 重点回答以下 4 个研究问题.

研究问题 1: PyPI 生态系统中的进行漏洞影响范围评估的必要性和难度如何?

研究问题 2: PyVul++在进行函数调用分析时准确性如何, 和 PyCG 等工具对比结果如何?

研究问题 3: PyVul++的漏洞影响范围评估效果如何和 pip-audit 等工具对比效果如何?

研究问题 4: PyVul++的漏洞影响范围评估结果如何提高实际项目的漏洞检测能力?

4.2 实验数据

本文的方法是面向整个 PyPI 仓库的. 由于 PyPI 仓库是不断变化的, 本文在设计实验方案对 PyPI 仓库进行了快照处理, 即采用某一时刻的 PyPI 仓库及仓库内的 Python 包进行实验. 同时在进行 CVE 漏洞影响范围评估实验时, 本文模拟了漏洞报出时的 PyPI 仓库快照, 即根据 Python 漏洞报告回溯 Python 包版本, 并对当前 PyPI 生态系统进行了实验. 这样能更好展示 PyVul++的分析效果, 并观察到 PyPI 生态系统漏洞包更新的信息.

对于研究问题 2, 本文选择了 PyCG 提供的 micro-benchmark 和 macro-benchmark.micro-benchmark 与 macro-benchmark 的区别在于程序大小, micro-benchmark 包含小型的 Python 程序, 而 macro-benchmark 包含的是流行的真实 Python 包. 本文在 micro-的基础上增加了上下文敏感的测试用例和引用第三方无源码包的测试用例. 最终 micro-的测试集涉及 18 种 Python 语言特性, 共 121 个测试用例, macro-benchmark 包含 5 个真实的 Python 包.

对于研究问题 3、4, 本文根据 Python 官方 (Python packaging authority, pypa) 维护的 Python CVE 漏洞数据 (advisory-database[42], 截至 2022 年 7 月共收集 2 089 个与 Python 相关的 CVE 漏洞, 漏洞报告时间从 2005 年到 2022 年), 按照下载量、受影响包数量这两个权重对漏洞包进行排序, 并根据是否有漏洞 PoC 文件和是否有漏洞函数说明, 挑选了 10 个 CVE 漏洞进行实验.

4.3 实验环境

本文的实验运行在 8 核 Intel i7 CPU、64 GB 内存的机器上, 操作系统为 Ubuntu 18.04.

4.4 实验结果与分析

研究问题 1: PyPI 生态系统中进行漏洞影响范围评估的必要性和难度如何?

本文共收集到 PyPI 仓库中 385 855 个 Python 包 (仓库快照时间为 2022 年 7 月). 通过第 3.1 节介绍的依赖关系分析, 共发现存在正向依赖关系的包 156 445 个, 即 40.55% 的包和其他包存在依赖关系. 进一步, 存在反向依赖关系的包共 54 654 个, 平均反向依赖的数量为 48.94, 其中数量最多为 typing-extensions (共有 66 093 个), 平均反向引用深度为 1.63, 其中深度最大为 setuptools (深度为 33). 表 1 列举了部分漏洞报告中曝出存在漏洞的包的依赖关系统计.

由实验结果可见, 如果一个 Python 包存在漏洞, 那么 PyPI 生态系统中由于广泛的依赖关系, 漏洞将影响更多的 Python 包. 在缺少漏洞影响范围评估的情况下, PyPI 管理者很难准确、全面发现和处理漏洞包, Python 项目开发者也很难检查自己的项目是否存在 Python 漏洞. 而由于包之间存在的复杂依赖关系, 实现细粒度的漏洞影响范围评估具有很大难度. 同时, 从表中可以看出, 如果采用正向搜索, 对于每个漏洞包其搜索范围都是当前 PyPI 仓库的所有包 (385 855 个), 而采用反向搜索, 可以大幅减少搜索包范围 (即表 1 中影响包数量).

研究问题 2: PyVul++在进行函数调用分析时准确性如何, 和 PyCG 等工具对比结果如何?

本文提出的 PyVul++在 PyCG 的基础上增加了上下文敏感的分析方法, 能够得到更准确的 Python 函数调用信息. 本文主要通过对比不同工具的精确率和召回率来说明这一点. 精确率是工具生成的函数调用关系准确性的反映, 高精确率可以降低出现错误漏洞函数调用的可能性, 减少第 2 阶段的误报. 召回率则是工具所生成的函数调

用关系充分性的反映, 高的召回率则更有可能检测出更多的漏洞函数调用, 在第 2 和第 3 阶段检测出更多的漏洞包, 减少漏报率. 因此工具生成 CG 的精确率和召回率对其在漏洞影响范围细粒度评估中具有重要影响.

表 1　部分漏洞包的依赖关系统计 (2022 年 7 月)

序号	CVE编号	漏洞包	版本	影响包数量	反向依赖深度
1	CVE-2018-15560	pycryptodome	3.6.5	1 969	12
2	CVE-2020-14343	pyyaml	5.3	12 010	11
3	CVE-2022-24303	pillow	9.0.0	7 670	10
4	CVE-2020-36242	cryptography	3.3	9 353	12
5	CVE-2019-7548	sqlalchemy	1.2.17	3 409	10
6	CVE-2020-28493	jinja2	2.11.2	8 444	12
7	CVE-2021-21330	aiohttp	3.7.3	6 260	12
8	CVE-2020-28975	scikit-learn	0.23.2	6 914	8
9	CVE-2021-28957	lxml	4.6.2	8 285	22
10	CVE-2021-29510	pydantic	1.8.1	5 775	9
平均				**7 008**	**11**

为了验证 PyVul++在进行函数调用分析时的准确性, 本文选取了 3 个 Python 程序分析工具 (PyCG/code2flow/pyan) 进行对比. 表 2 展示了不同工具对测试集的所有测试用例分析结果的平均精确率和召回率 (精确率指工具生成的边有多少是正确的, 召回率指有多少边被工具正确生成). 图 9 和图 10 是按照 Python 特性 (即 classes/returns/imports/dynamic/assignments/args 等) 展示的细节数据.

表 2　不同工具的 Python 函数调用分析在 micro-benchmark 上结果对比 (%)

指标	PyVul++	PyCG	code2flow	pyan
精确率	**86.71**	83.61	48.48	44.63
召回率	**83.20**	79.15	23.42	50.07

由表 2、图 9、图 10 可见, PyVul++在 Python 函数调用分析方面的准确性更高. 值得注意的是, code2flow (图 9、图 10 中深蓝色数据柱, 位于每张图的第 3 列) 和 pyan (图 9、图 10 中浅蓝色数据柱, 位于每张图的第 4 列) 都出现了 CG 生成失败的情况 (如子图 imports/dynamic/assignments/exceptions). 其中, lists 由于对内置函数的处理而优于 PyCG, 常量判定中新增的 ast.Constant 也使得 PyVul++在分析 dicts.return 这个 case 时能正确解析字典 b[‘a’], 从而得到正确的调用链.

本文对每个测试用例的分析时间进行统计. 表 3 展示了所有测试用例的平均分析用时. 可以看到因为 PyVul++引入了上下文敏感的程序分析计算, 相比 PyCG 耗时有所增加, 但仍保持了较高的分析效率 (0.25 s).

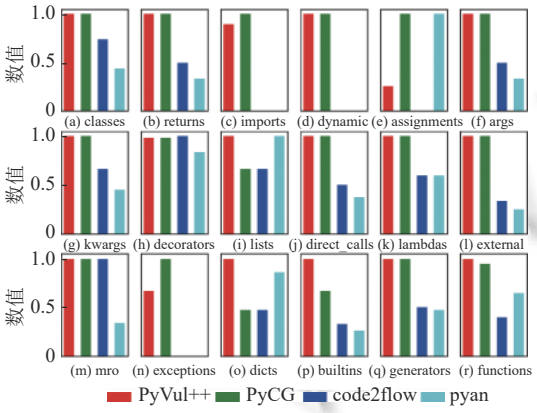

图 9　精确率

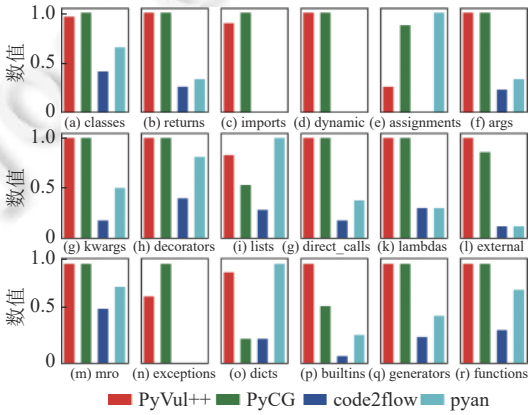

图 10　召回率

进一步, 本文在 PyCG 的 macro-benchmark 上对 PyVul++和 PyCG 进行了手工对比, 结果如表 4 所示. 需要注意的是, macro-benchmark 由于代码量大, 目前没有完美的 CG 生成工具, 而靠人工生成精确的 CG 也非常困难, 因此只能得到相对精确的 ground-truth. 从表 4 可以看出, PyVul++在召回率上远高于 PyCG. 精确率方面, 由于 PyCG 不对第三方库进行分析, 丢失了所有与第三方库有关的数据流, 产生的边远少于 PyVul++, 因此在精确率计算上有很大的优势, 略高于 PyVul++. 值得一提的是, 本文在进行该部分实验时, 发现了若干 macro-benchmark 的错误 (例如“sys.platform.startswith”实际应为”<**PyStr**>.startswith”, 缺少“re.findall”“<**PyStr**>.strip”等). 因此本文展示的 PyCG 的精确率和召回率均为对 macro-benchmark 进行修正后的正确结果, 并已将该修正结果反馈给 PyCG 作者用以更新 benchmark.

表 3　不同工具的 Python 函数在 micro-benchmark 上调用分析时间对比 (s)

工具	平均用时
PyVul++	**0.25**
PyCG	0.14
code2flow	0.03
pyan	0.06

表 4　不同工具在 macro-benchmark 上的对比结果 (%)

包	精确率		召回率	
	PyVul++	PyCG	PyVul++	PyCG
autojump	98.90	99.20	80.54	67.39
fabric	92.04	98.47	80.09	59.72
asciinema	99.65	97.01	79.39	65.74
face_classification	98.90	99.75	94.51	86.37
Sublist3r	99.42	98.00	84.07	59.80
平均	**97.78**	**98.49**	**83.72**	**67.80**

同样的, 本文对两个工具的分析时间也进行比对 (表 5). 特别地, 分析时间与分析代码量和分析轮数密切相关. macro-benchmark 项目的代码量 (平均 2 006 行代码) 远高于 micro-benchmark (平均 8 行代码), 因此增加的上下文信息会在每一轮的迭代中消耗更多的计算资源, 导致时间开销较大. 在实际项目检测中也可以根据实际需求配置指针分析的分析轮数, 取得分析效率和分析精确度的平衡.

表 5　macro-benchmark 函数调用分析时间对比 (s)

工具	autojump	fabric	asciinema	face_classification	Sublist3r
PyCG	0.40	0.49	0.58	0.41	0.31
PyVul++	20.32	73.19	2.26	2.66	3.36

研究问题 3: PyVul++的漏洞影响范围评估效果如何, 和 pip-audit 等工具对比效果如何?

本文对选取的 10 个 CVE 漏洞使用 PyVul++进行分析 (如表 6 所示, 每一纵列为对一个 CVE 漏洞的分析数据). 第 1–3 行记录了 CVE 编号及其出现的 Python 包和 Python 包版本, 第 4 行记录了 CVE 漏洞的类型. 第 5 行记录了 CVE 漏洞报告的公布日期. 对于每个包的版本, 本文根据 CVE 漏洞报告的公布日期, 选择该包在 PyPI 上离 CVE 公布日期最近的前一个版本进行分析, 即模拟了 PyPI 在 CVE 漏洞曝出时的快照.

根据第 3 节的描述, PyVul++的第 1 阶段分析是预处理 (第 6 行), 在漏洞曝出之后, PyVul++进行第 2 阶段的分析 (第 7 行), 即对漏洞包和 PyPI 中所有包进行漏洞函数识别 (该阶段为了实验可运行设置了 10 min 的超时). 对于第 2 阶段分析得到的候选漏洞包, PyVul++借助漏洞 PoC 进行第 3 阶段的漏洞触发条件验证, 得到最终的漏洞影响范围 (第 8 行). 第 3 阶段的结果即为工具的最后分析结果. 本文对最终结果进行了手工验证, 具体是通过审计受影响的软件包的源代码, 并通过分析尝试生成能够触发漏洞的 PoC 来确定漏洞的存在.

纵向查看表 6 中的数据可以看出, PyVul++在对不同 CVE 漏洞的分析过程中各阶段分析均发挥了作用, 最终实现了细粒度的漏洞影响范围评估. 进一步, 本文对最终确认的漏洞影响范围中存在的包依赖类型进行了分析, 并展示在了表 6 的最后两行. 其中, 直接依赖和间接依赖分别指结果中直接/间接依赖于漏洞包的数量. 例如, 对于 CVE-2021-28957, 直接依赖包为 3 个 (具体是“boilerpipy”, “jsonify-html”和“mastobot”), 间接依赖包为 1 个 (具体是“trafilatura”).

本文对实验结果进行了手工验证. PyVul++存在一个漏报, CVE-2021-29510 的最终分析结果应该为 1. 进一步

分析, PyVul++的错误出现在触发条件验证过程中 (第 3 阶段), 漏报的漏洞包 (odmanticti) 中漏洞函数的参数是由外部输入控制, PyVul++无法判断该参数的取值, 因此无法与已知条件匹配, 导致漏报. 漏报在 3 个阶段中均有可能存在, 在第 1 阶段可能由于包的引用关系缺失导致漏报, 第 2 阶段可能由于 CG 不精确导致漏洞函数调用的缺失, 第 3 阶段可能因为 PoC 提供的触发路径并非唯一导致漏报. 这 3 种漏报的可能并非本文关注的重点, 且由于目前工具的局限性也没有办法对其进行准确评估, 因此本文没有对漏报率进行评估.

表 6　漏洞影响范围评估结果及各阶段分析的中间结果

	CVE编号	CVE-2018-15560	CVE-2020-14343	CVE-2022-24303	CVE-2020-36242	CVE-2019-7548	CVE-2020-28493	CVE-2021-21330	CVE-2020-28975	CVE-2021-28957	CVE-2021-29510
CVE漏洞实验基本信息	包	pycryptodome	pyyaml	pillow	cryptography	sqlalchemy	jinja2	aiohttp	scikit-learn	lxml	pydantic
	版本	3.6.5	5.3	9.0.0	3.3	1.2.17	2.11.2	3.7.3	0.23.2	4.6.2	1.8.1
	CWE	CWE-190	CWE-20	NVD-CWE-noinfo	CWE-190/CWE-787	CWE-89	CWE-400	CWE-601	NVD-CWE-noinfo	CWE-79	CWE-835
	公布日期	2018-08-19	2021-02-09	2022-03-27	2021-02-07	2019-02-06	2021-02-01	2021-02-25	2020-11-21	2021-03-21	2021-05-13
实验过程数据及结果	第1阶段	1 969	12 010	7 670	9 353	3 409	8 444	6 260	6 914	8 285	5 775
	第2阶段	81	769	6	0	0	0	1	0	14	1
	第3阶段 (工具结果)	**7**	**20**	**2**	**0**	**0**	**0**	**1**	**0**	**4**	**0**
	手工验证	7	20	2	0	0	0	1	0	4	1
依赖分析	直接依赖	7	20	2	0	0	0	0	0	3	1
	间接依赖	0	0	0	0	0	0	0	0	1	0

为了更好地展示 PyVul++在漏洞影响范围评估方面的能力, 本文设计了和现有 PyPI 漏洞影响范围评估工具 pip-audit 的对比实验 (本文采用 pip-audit –path 的选项), 同时本文设计了基于 PyCG 的漏洞影响范围评估方法 (即第 2 阶段采用 PyCG) 的对比实验. pip-audit 是一个基于包依赖关系实现的漏洞扫描工具, 通过扫描当前 Python 环境或项目配置文件 (如 requirement 文件等) 识别已知漏洞. 这是目前常用的已知漏洞识别方法, 通过与其对比可以更好地展示函数级漏洞识别工具的有效性. 为了展示本文针对漏洞影响范围评估场景而对 PyCG 一系列改进的有效性, 本文增加了一个与 PyCG 的对比实验, 将本文工具的第 2 阶段 CG 分析直接替换为 PyCG 的版本. 因为 PyCG 需要下载并进入外部调用的第三方库进行分析, 本文同时列举了在本地存在对应版本漏洞包的情况下的分析结果, 即 PyCG*. 最终实验结果如表 7 所示. 实验结果显示, 基于 PyCG 的方法会造成大量误报, 即其结果中的 Python 包实际不受漏洞影响, 同时 PyCG 的分析需要下载和进入第三方库进行分析, 限制了 PyCG 的使用效果. pip-audit 存在大量漏报, 主要原因是其通过提取包使用的第三方库版本, 与已知的漏洞包版本进行比对. 而对于没有在 requirement 中明确指定版本, 或者没有 requirement 的情况, 都会因为版本缺失而产生漏报. 可见, 包粒度的漏洞影响范围评估方法无法满足实际中 Python 漏洞影响范围评估的需求.

表 7　不同工具的 Python 漏洞影响范围评估结果对比

工具	CVE-2018-15560	CVE-2020-14343	CVE-2022-24303	CVE-2020-36242	CVE-2019-7548	CVE-2020-28493	CVE-2021-21330	CVE-2020-28975	CVE-2021-28957	CVE-2021-29510
PyVul++	7	20	2	0	0	0	1	0	4	0
PyCG	0	1 237	0	0	0	0	1	0	23	1
PyCG*	0	1 408	0	0	0	0	1	0	30	1
pip-audit	0	0	0	0	0	0	0	0	0	0

注: *表示本地安装了对应版本的漏洞包

研究问题 4: PyVul++的漏洞影响范围评估结果如何提高实际项目的漏洞检测能力?

本文首先对当前的 PyPI 生态系统 (2022 年 7 月) 进行了分析. 实验发现当前仍存在 11 个包存在引用未修复的漏洞函数的安全问题 (如表 8 所示). 进一步通过对 11 个包的手工验证, 本文发现 11 个包中大部分是因为没有在 requirement 或 setup 文件中指明版本, 其中 sense-core 包则没有在配置中指明引用了漏洞包. 本文发现的安全问题已经上报了 PyPI 管理者和相应包的开发者.

表 8 当前 PyPI 漏洞影响范围评估结果

CVE编号	CVE-2018-15560	CVE-2020-14343	CVE-2022-24303	CVE-2020-36242	CVE-2019-7548	CVE-2020-28493	CVE-2021-21330	CVE-2020-28975	CVE-2021-28957	CVE-2021-29510
数量	7	1	3	0	0	0	0	0	0	0

为了进一步展示漏洞影响范围评估结果对实际 Python 项目的漏洞检测的作用, 本文选取最近发布的 CVE-2022-24303 进行说明. 为了展示漏洞的实际影响, 本文选择对当前的 PyPI 生态系统 (2022 年 7 月) 进行分析. 漏洞函数位于包 pillow (9.0.0) 中, pillow 是 Python 3 最常用的图像处理库, 根据 PyVul++的分析共影响 7 670 个 Python 包, 包括常见的绘图库 matplotlib 和常用的机器学习库 sklearn. 进一步, PyVul++在漏洞函数识别阶段将受影响包的范围缩小到 8 个, 借助漏洞 PoC 的漏洞触发条件验证最终确定 3 个受影响包 (如图 11 所示).

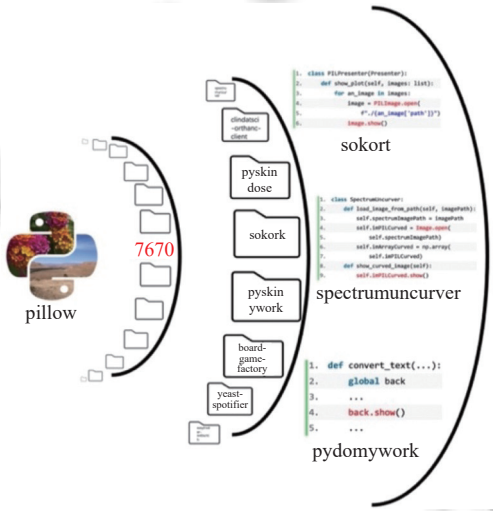

图 11 CVE-2022-24303 影响范围

5 讨 论

本文提出的 PyVul++实现了细粒度的漏洞影响范围评估, 尽管实验验证了方法的有效性, 但是在实际的 PyPI 生态系统漏洞影响范围评估过程中, PyVul++仍然存在一些不足. 首先, 现实中 Python 漏洞和 PoC 文件仍缺少公开可用的测试集, 后续工作希望能构建较大规模的数据集并公开, 本文也将开源文章数据方便同行研究. 其次, 本文方法仍有改进空间. 一方面, 方法仍借助于包配置文件分析建立 PyPI 生态系统索引, 然而根据 Cao 等人[30]的研究, Python 代码和配置文件可能存在不一致情况, 即配置文件中指明的依赖包可能未使用, 或程序中用到的依赖包未在配置文件中指明; 另一方面, 本文实现的上下文敏感的 Python 函数调用分析方法仍然存在不足, 如上文讨论的 CVE-2021-29510 分析结果. 未来 Python 代码分析的工作进展可以用于对 PyVul++的改进. 最后本文采用的漏洞触发条件验证方法无法处理 Python 代码变化过大的情况, 即 Python 漏洞相关代码重构, 这部分将作为未来研究内容继续开展后续研究.

除了主流的 Python 分析工具 (静态分析), 符号执行工具也能够进行上下文敏感分析并生成调用图 (动态分析), 因此, 本文也尝试与符号执行工具进行对比. 具体来讲, 本文尝试通过修改 Python 的符号执行工具 (PyExZ3) 进行 CG 的生成. PyExZ3 是针对 Python 的最新符号执行工具, 支持对分支进行记录[43], 本文根据 PyExZ 用法, 通过记录当前执行函数的分支记录形成函数调用链, 组合多个调用链最终形成 CG. 然而由于以下 4 方面问题, 通过符号执行产生 CG 失败, 主要包括: (1) 需要手动指定入口函数. (2) 无法给未执行的函数生成 CG. (3) PyExZ3 异常执行退出. 因为 micro-benchmark 主要测试工具的函数调用关系, 省略了函数参数和分支, 虽然我们手工指定了每个测试用例的入口函数, PyExZ3 无法对函数参数进行符号化和跟踪执行函数内部的不同分支, 进而无法执行生成 CG.

6 结束语

本文提出了面向 PyPI 生态系统的 Python 漏洞影响范围细粒度评估方法 PyVul++, 通过 PyPI 生态系统索引构建、基于漏洞函数识别的函数粒度影响范围评估和基于触发条件提取的漏洞存在性验证, 将之前的包粒度分析提升为函数粒度的分析. 通过 Python 代码函数粒度分析对比实验和 Python 漏洞在 PyPI 生态系统中的漏洞影响范围评估对比实验, 本文验证了 PyVul++的有效性, 并新发现了当前 PyPI 生态系统中 11 个包存在引用未修复的漏洞函数的安全问题.

References:

[1] Programming language rankings in May 2022. 2022 (in Chinese). https://hellogithub.com/report/tiobe?month=5

[2] TensorFlow. https://tensorflow.org

[3] PyTorch. https://pytorch.org

[4] Flask. https://flask.palletsprojects.com

[5] The Python package index (PyPI). https://pypi.org/

[6] Potential remote code execution in PyPI. 2021. https://blog.ryotak.me/post/pypi-potential-remote-code-execution-en/

[7] Cryptominers slither into Python projects in supply-chain campaign. 2021. https://threatpost.com/cryptominers-python-supply-chain/167135/

[8] Several malicious typosquatted Python libraries found on PyPI repository. 2021. https://thehackernews.com/2021/07/several-malicious-typosquatted-python.html

[9] Alfadel M, Costa DE, Shihab E. Empirical analysis of security vulnerabilities in Python packages. In: Proc. of the 2021 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Honolulu: IEEE, 2021. 446–457. [doi: 10.1109/SANER50967.2021.00048]

[10] CVE. https://cve.mitre.org/

[11] pip-audit. https://pypi.org/project/pip-audit/

[12] Salis V, Sotiropoulos T, Louridas P, Spinellis D, Mitropoulos D. PyCG: Practical call graph generation in Python. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 1646–1657. [doi: 10.1109/ICSE43902.2021.00146]

[13] Hassija V, Chamola V, Gupta V, Jain S, Guizani N. A survey on supply chain security: Application areas, security threats, and solution architectures. IEEE Internet of Things Journal, 2021, 8(8): 6222–6246. [doi: 10.1109/JIOT.2020.3025775]

[14] Enck W, Williams L. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. IEEE Security & Privacy, 2022, 20(2): 96–100. [doi: 10.1109/MSEC.2022.3142338]

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

[16] Wu ZH, Zhang C, Sun H, Yan XX. Survey on application of binary reverse analysis in detecting software supply chain pollution. Journal of Computer Applications, 2020, 40(1): 103–115 (in Chinese with English abstract). [doi: 10.11772/j.issn.1001-9081.2019071245]

[17] Ji SL, Wang QY, Chen AY, Zhao BB, Ye T, Zhang XH, Wu JZ, Li Y, Yin JW, Wu YJ. Survey on open-source software supply chain security. Ruan Jian Xue Bao/Journal of Software, 2022, 34(3): 1330–1364 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6717.htm [doi: 10.13328/j.cnki.jos.006717]

[18] Decan A, Mens T, Claes M. An empirical comparison of dependency issues in OSS packaging ecosystems. In: Proc. of the 24th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Klagenfurt: IEEE, 2017. 2–12. [doi: 10.1109/SANER.2017.

7884604]

[19] Decan A, Mens T, Constantinou E. On the impact of security vulnerabilities in the NPM package dependency network. In: Proc. of the 15th IEEE/ACM Int'l Conf. on Mining Software Repositories. Gothenburg: IEEE, 2018. 181–191.

[20] Staicu CA, Pradel M, Livshits B. SYNODE: Understanding and automatically preventing injection attacks on NODE.JS. In: Proc. of the 25th Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2018.

[21] Ohm M, Plate H, Sykosch A, Meier M. Backstabber's knife collection: A review of open source software supply chain attacks. In: Proc. of the 17th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment. Lisbon: Springer, 2020. 23–43. [doi: 10.1007/978-3-030-52683-2_2]

[22] Zahan N, Zimmermann T, Godefroid P, Murphy B, Maddila C, Williams L. What are weak links in the NPM supply chain? In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Pittsburgh: IEEE, 2022. 331–340. [doi: 10.1145/3510457.3513044]

[23] Vu DL, Pashchenko I, Massacci F, Plate H, Sabetta A. Typosquatting and combosquatting attacks on the Python ecosystem. In: Proc. of the 2020 IEEE European Symp. on Security and Privacy Workshops (EuroS&PW). Genoa: IEEE, 2020. 509–514. [doi: 10.1109/EuroSPW51379.2020.00074]

[24] Liang GP, Zhou XY, Wang QY, Du YT, Huang C. Malicious packages lurking in user-friendly Python package index. In: Proc. of the 20th IEEE Int'l Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom). Shenyang: IEEE, 2021. 606–613. [doi: 10.1109/TrustCom53373.2021.00091]

[25] Duan R, Alrawi O, Kasturi RP, Elder R, Saltaformaggio B, Lee W. Towards measuring supply chain attacks on package managers for interpreted languages. In: Proc. of the 28th Annual Network and Distributed System Security Symp. The Internet Society, 2020.

[26] Ruohonen J. An empirical analysis of vulnerabilities in Python packages for Web applications. In: Proc. of the 9th Int'l Workshop on Empirical Software Engineering in Practice (IWESEP). Nara: IEEE, 2018. 25–30. [doi: 10.1109/IWESEP.2018.00013]

[27] Xu SH, Gao Y, Fan LL, Liu ZL, Ji H. LiDetector: License incompatibility detection for open source software. ACM Trans. on Software Engineering and Methodology, 2023, 32(1): 22. [doi: 10.1145/3518994]

[28] Wang Y, Wen M, Liu YP, Li ZM, Wang C, Yu H, Cheung SC, Xu C, Zhu ZL. Watchman: Monitoring dependency conflicts for Python library ecosystem. In: Proc. of the 42nd IEEE/ACM Int'l Conf. on Software Engineering. Seoul: IEEE, 2020. 125–135. [doi: 10.1145/3377811.3380426]

[29] Mukherjee S, Almanza A, Rubio-González C. Fixing dependency errors for Python build reproducibility. In: Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Virtual: ACM, 2021. 439–451. [doi: 10.1145/3460319.3464797]

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

[31] Chen ZF, Chen L, Zhou YM, Xu ZG, Chu WC, Xu BW. Dynamic slicing of Python programs. In: Proc. of the 38th IEEE Annual Computer Software and Applications Conf. Vasteras: IEEE, 2014. 219–228. [doi: 10.1109/COMPSAC.2014.30]

[32] Peng Y, Gao CY, Li ZJ, Gao BW, Lo D, Zhang QR, Lyu M. Static inference meets deep learning: A hybrid type inference approach for Python. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering. Pittsburgh: IEEE, 2022. 2019–2030. [doi: 10.1145/3510003.3510038]

[33] Chen ZF, Li YH, Chen BH, Ma WWY, Chen L, Xu BW. An empirical study on dynamic typing related practices in Python systems. In: Proc. of the 28th Int'l Conf. on Program Comprehension. Seoul: ACM, 2020. 83–93. [doi: 10.1145/3387904.3389253]

[34] Jiang CM, Hua BJ, Ouyang WR, Fan QL, Pan ZZ. PyGuard: Finding and understanding vulnerabilities in Python virtual machines. In: Proc. of the 32nd IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Wuhan: IEEE, 2021: 468–475. [doi: 10.1109/ISSRE52982.2021.00055]

[35] Xu ZG, Liu P, Zhang XY, Xu BW. Python predictive analysis for bug detection. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 121–132. [doi: 10.1145/2950290.2950357]

[36] Ma L, Yang HH, Xu JX, Yang ZX, Lao QD, Yuan D. Code analysis with static application security testing for Python program. Journal of Signal Processing Systems, 2022, 94(11): 1169–1182. [doi: 10.1007/s11265-022-01740-z]

[37] Fromherz A, Ouadjaout A, Miné A. Static value analysis of Python programs by abstract interpretation. In: Proc. of the 10th NASA Formal Methods Symp. Newport News: Springer, 2018. 185–202. [doi: 10.1007/978-3-319-77935-5_14]

[38] Bagheri A, Hegedűs P. A comparison of different source code representation methods for vulnerability prediction in Python. In: Proc. of the 14th Int'l Conf. on the Quality of Information and Communications Technology. Algarve: Springer, 2021: 267–281. [doi: 10.1007/978-3-030-85347-1_20]

[39] Wartschinski L, Noller Y, Vogel T, Kehrer T, Grunske L. VUDENC: Vulnerability detection with deep learning on a natural codebase for

Python. Information and Software Technology, 2022, 144: 106809. [doi: 10.1016/j.infsof.2021.106809]
[40] Peng SH, Liu PY, Zhao JL. Vulnerability detection method of Python clone code based on feature matrix. Journal of Wuhan University (Natural Science Edition), 2019, 65(5): 472–478 (in Chinese with English abstract). [doi: 10.14188/j.1671-8836.2019.05.008]
[41] PyPI Stats. 2023. https://pypistats.org/packages/pycryptodome
[42] advisory-database. 2023. https://github.com/pypa/advisory-database
[43] PyExZ3. 2015. https://github.com/thomasjball/PyExZ3

附中文参考文献:

[1] 2022年5月编程语言排行榜. 2022. https://hellogithub.com/report/tiobe?month=5
[15] 何熙巽, 张玉清, 刘奇旭. 软件供应链安全综述. 信息安全学报, 2020, 5(1): 57–73. [doi: 10.19363/J.cnki.cn10-1380/tn.2020.01.06]
[16] 武振华, 张超, 孙贺, 颜学雄. 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. [doi: 10.11772/j.issn.1001-9081.2019071245]
[17] 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, 2022, 34(3): 1330–1364. http://www.jos.org.cn/1000-9825/6717.htm [doi: 10.13328/j.cnki.jos.006717]
[40] 彭双和, 刘佩瑶, 赵佳利. 基于特征矩阵的Python克隆代码漏洞检测方法. 武汉大学学报 (理学版), 2019, 65(5): 472–478. [doi: 10.14188/j.1671-8836.2019.05.008]

王梓博(1996－), 男, 博士生, CCF 学生会员, 主要研究领域为系统与软件安全.

应凌云(1982－), 男, 博士, 研究员, CCF 专业会员, 主要研究领域为恶意代码分析, 软件供应链安全.

贾相堃(1990－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为软件与系统安全, 漏洞挖掘与分析.

苏璞睿(1976－), 男, 博士, 研究员, CCF 专业会员, 主要研究领域为系统安全, 恶意代码分析, 漏洞挖掘.