

分类号: TP311.5

单位代码: 10335

密 级: 无

学 号: _____

浙江大学

博士学位论文



中文论文题目: 针对 AI 系统的
供应链安全分析与防护

英文论文题目: Security Analysis & Protection
for AI System Supply Chains

申请人姓名: _____

指导教师: _____

合作导师: _____

学科(专业): 网络与信息安全

研究方向: AI 软件与系统安全

所在学院: 计算机科学与技术学院

论文递交日期 2026 年 3 月

针对 AI 系统的
供应链安全分析与防护



论文作者签名: _____

指导教师签名: _____

论文评阅人 1: _____

评阅人 2: _____

评阅人 3: _____

评阅人 4: _____

评阅人 5: _____

答辩委员会主席: _____

委员 1: _____

委员 2: _____

委员 3: _____

委员 4: _____

委员 5: _____

答辩日期: _____

Security Analysis & Protection

for AI System Supply Chains



Author's signature: _____

Supervisor's signature: _____

External reviewers: _____

Examining Committee Chairperson:

Examining Committee Members:

Date of oral defence: _____

浙江大学研究生学位论文独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 浙江大学 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名： 签字日期： 年 月 日

学位论文版权使用授权书

本学位论文作者完全了解 浙江大学 有权保留并向国家有关部门或机构送交本论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 浙江大学 可以将学位论文的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

(保密的学位论文在解密后适用本授权书)

学位论文作者签名： 导师签名：
签字日期： 年 月 日 签字日期： 年 月 日

勘误表

序言

摘要

Abstract

缩略词表

英文缩写	英文全称	中文全称
ZJU	Zhejiang University	浙江大学

目录

勘误表	I
序言	III
摘要	V
Abstract	VII
缩略词表	IX
目录	XI
图目录	XIII
表目录	XV
1 绪论	1
1.1 研究背景及意义	1
1.2 研究现状与目标	4
1.3 本文研究内容与贡献	9
1.4 本文组织与章节安排	13
2 AI 系统供应链背景知识	15
2.1 软件应用层背景知识	15
2.1.1 软件供应链复杂性	15
2.1.2 软件供应链对 AI 系统的威胁与防护	17
3 针对软件应用层的模块冲突威胁分析与防护框架	23
3.1 引言	23
3.1.1 背景知识	25
3.1.2 研究动机与现状	28
3.2 ModuleGuard 设计概览	32
3.2.1 面临挑战	32
3.2.2 具体设计	33
3.3 ModuleGuard 框架具体实现	38
3.4 实验评估	41
3.4.1 实验设置	41

3.4.2 评估结果与分析	43
3.5 实证结果分析	44
3.5.1 模块冲突定义	45
3.5.2 模块冲突影响与新型攻击面	46
3.5.3 PyPI 生态大规模检测结果	49
3.6 本章小节	56
4 针对模型框架层的新型恶意模型攻击与检测框架	59
4.1 引言	59
参考文献	61
附录	67
A 一个附录	67
B 另一个附录	67

图目录

图 1.1 AI 系统架构图.....	2
图 2.1 软件应用层供应链架构图	16
图 2.2 albumentationsx 软件 2.0.13 版本依赖图	18
图 3.1 Python 软件包层次结构图	26
图 3.2 Python 软件依赖声明的四种方式	27
图 3.3 Python 模块冲突真实案例及其影响	29
图 3.4 模块在安装前后发生改变案例	33
图 3.5 虚拟文件树模拟过程。左侧为原始分发包的物理目录拓扑，右侧为经参数解析与内存级模拟后生成的虚拟模块路径树。	36
图 3.6 ModuleGuard 框架图	38
图 3.7 模块替换攻击原理图	48
图 3.8 历年发布的软件包总数与存在模块冲突的软件包数量统计	52
图 A.1 附录中的图片.....	67

表目录

表 3.1 模块路径和依赖声明相关文件和参数列表	34
表 3.2 模块冲突问题的分类及其引发的威胁统计	40
表 3.3 模块和依赖解析技术评估实验结果	43
表 3.4 软件包中冲突频率最高的前 10 个模块路径	50
表 3.5 向 GitHub 开源社区报告的模块冲突漏洞工单及修复状态	53

1 绪论

1.1 研究背景及意义

随着人工智能 (Artificial Intelligence, AI) 技术的迅猛发展, AI 正在加速融入人类社会的各个领域, 并逐渐成为推动社会进步与产业升级的重要引擎。在日常生活中, AI 技术已广泛应用于自动驾驶、智能助手、自然语言处理等关键场景。例如在自动驾驶领域, 比亚迪推出的“天神之眼”高阶智能驾驶辅助系统, 能够实现全场景的感知与控制辅助功能, 为用户提供更加安全、高效的出行体验^[1]; 在智能助手方面, 苹果公司的“Siri 助手”与华为的“小艺助手”能够执行语音指令, 完成文件操作、应用启动等任务, 显著提升了人机交互的便捷性^[2-3]; 在自然语言生成领域, OpenAI 于 2022 年发布的 ChatGPT 引发广泛关注, 标志着以大参数语言模型 (Large Language Model, LLM) 为代表的生成式 AI 技术进入高速发展阶段^[4]。AI 的广泛应用不仅加速了社会向数字化、信息化与智能化的转型, 也成为衡量国家科技竞争力的重要标志。

AI 系统的分层架构。随着 AI 技术成体系地持续演化, 目前业界研究重点已逐步从单一模型的性能和结构优化, 扩展至模型在真实系统中的集成、部署与运行效率等更为系统性的问题。事实上, 在复杂应用环境中, AI 模型往往被嵌入到一个多层次、异构化的 AI 系统中, 形成从前端应用到后端算力硬件支持的一体化处理链。所谓 AI 系统, 是指由 AI 模型、模型管理软件、运行时环境支持的 AI 框架以及底层硬件资源协同构建而成的综合性技术体系, 其核心任务是对图像、语音、文本等输入数据进行智能化分析, 并输出相应的决策结果或交互反馈。如图 1.1 所示, 现代 AI 系统通常由三层组成: 软件应用层、模型框架层和硬件加速层, 三者之间层层依赖、密切协同, 共同构成支撑 AI 服务运行的完整技术栈。

软件应用层处于 AI 系统的最上层, 直接面向终端用户, 负责构建各类 AI 模型驱动的应用程序。在该层中, 开发者主要使用 Python 语言调用预训练的 AI 模型, 同时结合 Java、C++ 等高级编程语言实现定制化的业务逻辑和系统功能, 例如自动驾驶、人脸识别、智能助手、文字生成等智能服务。这些应用可以通过嵌入式部署或远程服务调用的方式对接模型推理模块, 从而灵活适配本地部署或云端服务等不同运行环境。

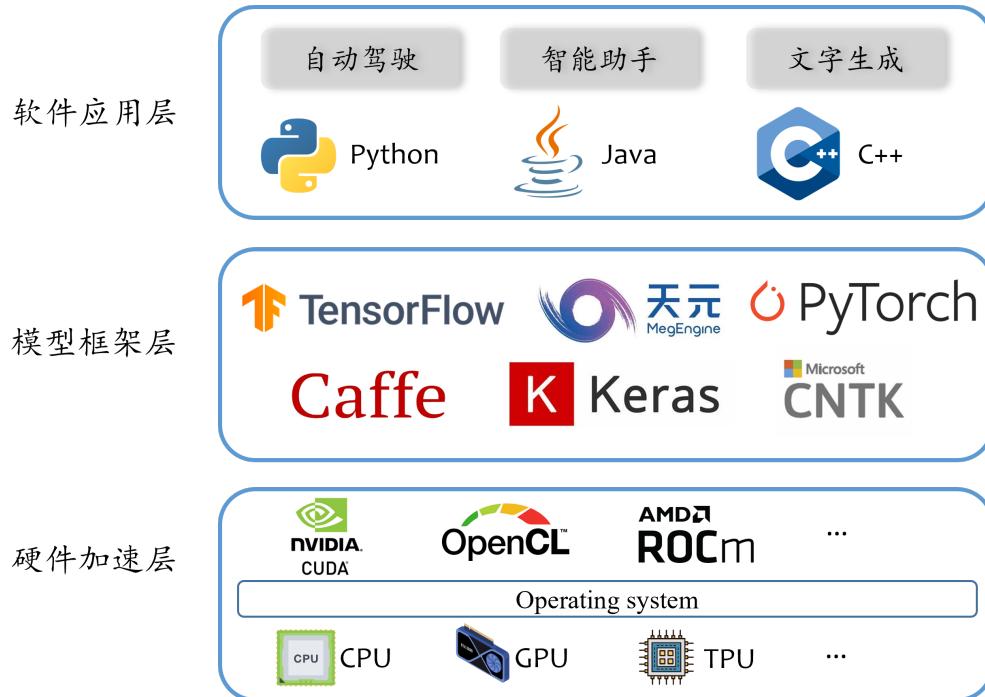


图 1.1 AI 系统架构图

模型框架层位于 AI 系统的中间层，是连接上层应用与下层硬件的核心支撑组件，承担模型训练、推理与部署的功能。在这一层，开发者通常依赖 TensorFlow、PyTorch 等主流深度学习框架^[5-6]，通过其提供的高层 API（多以 Python 形式暴露）定义模型结构，并调用由 C++ 或通用并行计算语言实现的底层算子，高效完成模型计算与参数优化。此外，受限于训练过程对算力资源和高质量标注数据的高昂需求，开发者往往从开源模型平台引入预训练模型，并通过迁移学习或微调的方式实现定制化能力。这一实践在显著提升开发效率和迭代速度的同时，也使模型框架层成为 AI 系统中高度依赖外部资源的关键环节。

硬件加速层位于 AI 系统的最底层，为 AI 模型的算子运算提供实际的运行平台和算力保障。鉴于深度学习模型普遍具有高度并行的计算特性，单纯依赖 CPU 已难以满足性能需求，因此该层通常采用 NVIDIA GPU、Intel NPU、Google TPU 等专用加速硬件。同时，操作系统之上还运行着各类支持通用并行计算的平台，如 CUDA、OpenCL 等。这些平台通过底层驱动与编译器将 AI 框架中的算子编译为 GPU 或 TPU 等硬件指令，并由调度器分配至合适的计算单元，从而实现对模型计算过程的高效加速。

AI 系统的供应链。 在 AI 系统中这种多层异构架构显然极大地帮助开发者提升了开发效率和模型运行效率，然而系统的多层次复杂性也引入了高度复杂的供应链关系，使

系统整体暴露于跨层级、跨组件的安全风险之中。在这种分层结构下，每一层均依赖大量第三方库、开源框架或底层驱动组件。当某一层的组件受到攻击或被植入恶意行为时，由于下层为上层提供运行支撑、上层对下层进行功能抽象，这种威胁极易沿着依赖链条向上传播，最终影响整个 AI 系统的安全性与稳定性，造成信息泄露、资产损失甚至服务中断等严重后果。

在软件应用层，开发者为了提升开发效率、减少重复实现，通常会引入大量开源第三方软件包。例如在 Python 生态中，图像处理相关的 AI 应用往往依赖 opencv-python 库^[7]，该库提供了丰富且高效的图像处理 API，能够在处理图像时采用高效的算法进行增强、还原、除噪。然而这种对第三方依赖的高度信任也构成了显著的供应链风险，一旦依赖包本身或其间接依赖被恶意投毒，或依赖包包含尚未修复的安全漏洞，恶意代码便可能在模型部署或运行阶段被触发，从而破坏整个 AI 系统的安全边界。

在模型框架层，从头开始训练模型的需要大量显卡算力的硬件支持，以及人工标注的数据集的昂贵成本，因此开发者往往选择基于现有预训练模型进行二次开发，修改模型结构或者对其参数进行微调。这些预训练开源模型广泛来源于 HuggingFace、Model Zoo、TensorFlow Hub 等开源模型平台^[8-10]。然而，此类模型来源复杂，且多以二进制格式分发，其内部结构与执行行为对使用者而言往往不可完全验证。一旦模型中被植入恶意后门或隐蔽的可执行逻辑，便可能在推理过程中触发参数篡改、敏感信息泄露，甚至实现任意代码执行，对 AI 系统构成严重威胁。

在硬件加速层中，AI 系统的运行往往使用于不同的加速平台，这些加速平台都依赖底层驱动程序、编译器和固件将 AI 算子映射至具体硬件执行逻辑。然而，这些底层组件通常由硬件厂商封闭实现，缺乏透明性，其内部的内存管理机制、计算单元调度方式等细节对用户不可见。一旦这些驱动或固件中存在安全漏洞，或者没有实现特定的安全防护机制，攻击者便可能通过精心构造的模型输入或算子参数触发底层缓冲区溢出，进一步导致权限提升或敏感信息泄露。

综上所述，AI 系统的安全问题已不再局限于单一模型或单一组件，而是深度嵌入于其跨层级、跨组件的复杂供应链之中。因此，构建可信且安全的 AI 系统，必须从供应链全生命周期的角度出发，对各层依赖关系、潜在威胁与防护机制进行系统性分析与设计。

1.2 研究现状与目标

在 AI 系统日益复杂化的背景下，AI 供应链安全问题已逐步受到研究界与工业界的高度关注。随着 AI 应用从单一模型扩展为由多层组件协同构成的复杂系统，其安全性也愈发依赖于不同层级组件之间的依赖关系及每一层独有的供应链机制。

AI 系统软件应用层供应链研究现状。 Python 作为 AI 软件开发中最为主流的编程语言之一，围绕其软件应用层的供应链安全问题，也层出不穷，根据 Sonatype 自 2019 年以来的多年年度报告，不仅开源软件包的数量在逐年激增，恶意软件包的数量也随之层出不穷，截至 2024 年，Sonatype 组织已经发现超过 704 102 个恶意的开源软件包^[11]。同时报告还指出 CVE 数量也持续呈指数级增长，开发者却无法跟上这样爆炸级的漏洞增长数，无法保证漏洞能够被及时修复。有相关研究表明，部分漏洞在开源软件包中存在的时间甚至长达 3 年以上未修复^[12]。高风险的开源软件包不仅会对 AI 软件的开发造成影响，甚至能对整个 AI 系统造成威胁。

目前已有大量研究从开源依赖管理、软件包漏洞以及运行时环境风险等方面展开深入分析。Cheng 等人提出了 PyCRE 框架，采用静态分析方法修复 Python 供应链中存在的错误依赖问题。其核心思路是通过源码分析与抽象语法树技术 (Abstract Resource Tree, AST) 提取模块之间的依赖关系，并结合软件包配置文件构建依赖图，从而判断依赖图中的依赖项是否存在缺失和冲突，进而修复依赖冲突和版本不一致等问题，以避免因依赖错误导致的 AI 软件部署失败^[13]。Mukherjee 等人提出了 PyDFix 框架，该框架通过在部署阶段收集运行时的控制台信息，判断安装过程中具体是哪些软件包出现错误，以及错误类型是依赖缺失还是版本不一致，并基于这些错误信息实现对依赖冲突的动态检测与修复^[14]。此外，Pipreq 作为一种静态依赖生成工具，它可以通过自动化地分析 Python 项目中 import 语句引入了哪些模块，再通过一个一对一的模块与软件包名的映射，来判断该项目需要哪些软件包，从而能够自动从项目源码中推导出所需的依赖列表，用于生成标准化的 requirements.txt 配置文件^[15]。在进一步扩展依赖修复范围方面，Ye 等人提出了 PyEGo 框架，该框架不仅关注软件包层面的依赖问题，还同时考虑系统环境依赖以及 Python 解释器版本兼容性，从而提升整体部署过程的可复现性与鲁棒性^[16]。此外，Cao 等人提出了 PyDC 框架，针对由于 Python 软件依赖配置错误引发的 Dependency Smell 问题展开研究，系统分析了此类问题的普遍性、成因及其演化过程。

除依赖关系修复外，针对 Python 生态中软件漏洞的分析同样是软件应用层供应链研究的重要方向之一。由于 AI 软件通常依赖大量的核心 AI 组件包和其他开源软件包，这些关键依赖项中潜在的漏洞也是影响 AI 系统安全性的重要因素之一。Mahon 等人提出了 PyPitfall 工具，从整体视角系统分析了 PyPI 生态中的依赖结构及漏洞传播关系，揭示了直接依赖与传递依赖在系统漏洞暴露风险中的显著影响^[17]。Alfadel 等人通过对 698 个 Python 包的 1396 条漏洞报告进行实证分析，发现 Python 软件包的漏洞数量呈上升趋势，且部分漏洞在被发现前的生命周期超过三年^[18]。在更宏观的层面，Ladisa 等人对开源软件供应链的攻击实现了一个系统的分类，该分类独立于特定的编程语言或生态系统，并覆盖了从代码贡献到软件包分发的所有供应链阶段。其以攻击树的形式刻画了 107 种不同的攻击向量，并将其与 94 起真实世界事件及 33 类缓解措施进行映射^[19]。类似地，Bogaerts 等人则更专注于 Python 语言，构建了包含 1026 个已公开 Python 漏洞的数据库，并提取了对应的补丁与易受攻击代码，为后续漏洞检测与修复研究提供数据基础^[20]。

综上所述，现有研究在 AI 软件应用层已提出诸多有效工具和框架，可以用于自动修复 AI 项目中常见的依赖配置错误、漏洞风险检测、软件包部署的错误等问题，从而提升 AI 软件包的稳定性和安全性。然而，现有工作大多聚焦于已知漏洞或显式依赖关系分析，并且通常都是以软件包为分析粒度，对更细粒度的模块级行为关注度较少，同时也尚未深入探讨供应链机制本身如何被恶意利用的问题。

AI 系统模型框架层供应链研究现状。在 AI 系统的模型框架层，研究者逐渐意识到预训练模型的本身及其其所依赖的运行框架和算子在 AI 供应链中的关键地位。近年来，开源模型库中的模型数量呈爆炸式增长。以 Hugging Face 为例，仅在 2022 年至 2025 年期间，该平台上累计发布的开源模型数量已超过 200 万个^[21]。如此庞大的模型规模在显著降低模型获取与复用成本的同时，也为恶意模型的传播提供了现实土壤。已有公开报告表明，开源模型库正逐步成为攻击者投放恶意载荷的新型渠道。JFrog 于 2024 年 2 月发布的分析报告指出，其在 Hugging Face 平台上发现了超过 100 个恶意模型，涉及 TensorFlow、PyTorch 等多个主流深度学习框架。这些模型在加载或推理阶段可触发反向 shell、任意文件读写、启动特定程序以及代码执行等恶意行为^[22]。相较于传统的软件包投毒攻击，模型与框架层面的攻击更贴近模型的实际执行路径，能够自然嵌入正常的模型加载与推理流程中，因而通常具备更强的隐蔽性和更高的潜在危害性。

围绕模型框架层的安全风险，现有研究已从多个角度展开系统性探索，相关工作大体可归纳为恶意模型行为分析、模型安全检测机制以及模型框架层漏洞挖掘等方向。从攻击目标与实现方式的角度看，模型层面的恶意逻辑注入主要可以划分为两类。

第一类是传统机器学习语境下的恶意模型，其核心目标在于操纵模型的预测或决策结果，而非直接执行系统级恶意行为。例如，攻击者可通过精心设计的训练过程，使智能驾驶模型在特定条件下将红灯错误识别为绿灯，从而间接诱发交通事故。这类攻击主要关注模型推理行为本身的安全性，对系统执行环境的影响通常是间接的。代表性研究包括后门攻击，即在训练阶段向模型中植入隐蔽触发器，使模型在正常输入下表现正常，而在触发条件出现时输出攻击者预期结果^[23-25]；以及对抗样本攻击，通过对输入样本施加微小扰动诱导模型产生错误分类^[26-28]。近年来，随着大参数模型高效微调技术的发展，研究者进一步发现，可利用 LoRA 等轻量化微调机制在不显著影响模型整体性能的前提下植入恶意触发逻辑，从而实现更加隐蔽的攻击^[29-30]。

第二类则是将 AI 模型本身作为恶意逻辑载体的攻击方式。在这一语境下，模型不再仅用于产生错误预测结果，而是被直接用于承载、隐藏并触发恶意软件或恶意代码，从而对运行模型的系统环境造成实质性威胁。现有研究表明，此类攻击主要通过以下三种方式实现。其一，攻击者将恶意软件或恶意逻辑嵌入模型的二进制参数或特定层次结构中，并在模型运行阶段对恶意载荷进行重组与触发。Hua 等人提出的 Malmodel 技术，将恶意模型嵌入 TensorFlow Lite 模型的层数、覆盖率等参数中，并利用 Java 反射机制主动触发^[31]。Hitaj 等人提出的 MaleficNet，利用扩频信道编码结合纠错技术，将恶意负载注入深度学习网络参数中^[32]。类似地，其他工作如 Evilmodel 1.0、Evilmodel 2.0 以及 StegoNet，则采用最低有效位 (Least Significant Bit, LSB) 隐写术将恶意软件隐藏于模型权重中^[33-35]。其二，攻击者将恶意逻辑直接嵌入模型的 lambda 层中。这类攻击主要适用于支持 lambda 层的模型框架（如 TensorFlow），通过在模型执行过程中触发任意代码执行实现攻击。然而，该方式通常较易被检测，因为仅需检查模型中是否存在 lambda 层并分析其逻辑即可识别异常行为^[36-37]。其三，也是目前最为普遍的一类方式，是利用 pickle 等不安全的模型序列化格式，将恶意逻辑嵌入模型文件中，并在模型反序列化过程中触发代码执行^[38-40]。针对这一威胁，工业界已提出多种检测与分析工具。例如，Pickletools 可对 pickle 格式的模型文件进行反序列化分析，从而识别潜在的恶意函数调用^[41]；Fickling 提供了对 Python pickle 对象的反编译、静态分析和字节码重写能力，既

可用于检测嵌入 PyTorch 模型的恶意行为，也可被用于构造攻击载荷^[42]；Picklescan 同样支持对基于 pickle 的恶意 PyTorch 模型进行检测^[43]。目前，业界较为先进的模型检测工具包括 Protect AI 公司推出的 ModelScan，该工具能够识别包括基于 pickle 的恶意模型和 TensorFlow lambda 层攻击在内的多种模型级恶意行为^[44]。

综上所述，现有研究已从多个角度揭示了模型框架层在 AI 系统供应链中面临的安全风险，充分地证明了模型本身可以被用作攻击载体。然而，这些工作大多将风险归因于恶意模型本身或不安全的序列化机制，从而将模型框架层的安全边界界定在模型层面，这是不完备的，事实上模型框架层自身和为模型框架提供的算子层面的攻击仍未被充分研究。

AI 系统硬件加速层供应链研究现状。在硬件加速层，AI 系统高度依赖 GPU、NPU 等专用计算设备以满足大规模并行计算与高性能推理需求，其底层供应链通常由计算加速硬件、设备驱动、运行时库以及 CUDA、OpenCL 等编程框架共同构成。随着 GPU 架构与配套软件栈复杂度的持续提升，相关供应链组件逐渐暴露出新的安全风险，使得硬件加速层在 AI 系统中不再仅是被动的计算执行单元，而演变为潜在的重要攻击入口。

随着 GPU 架构与配套软件栈复杂度的持续提升，相关供应链组件逐渐暴露出新的安全风险，使得硬件加速层在 AI 系统中不再仅是被动的计算执行单元，而演变为潜在的重要攻击入口。Saileshwar 等人首次将 Rowhammer 类硬件攻击扩展至 GPU 平台，提出了 GPUHammer 攻击方法，利用 GPU 的高并行特性在显存中诱发比特翻转，从而显著破坏深度学习模型参数的完整性，甚至仅通过翻转单个模型权重比特即可导致模型准确率出现灾难性下降^[45]。该工作表明，即便不直接攻击模型代码或框架逻辑，底层硬件的不可靠性本身亦可能成为影响 AI 系统可信性的关键因素。

除硬件本体外，围绕 GPU 构建的配套软件同样构成硬件加速层供应链中的重要组成部分，并已被多次证实存在安全隐患。已有公开漏洞表明，NVIDIA GPU 驱动中存在可被利用的高危漏洞，攻击者可借此实现权限提升或非法内存访问^[46-48]。与此同时，面向 AI 场景广泛部署的 NVIDIA GPU 容器生态亦被发现存在配置缺陷与隔离失效问题，可能引发跨容器攻击或敏感数据泄漏^[49-52]。此外，GPU 编译器及相关开发工具链同样曾被披露存在多项安全漏洞，这进一步扩大了硬件加速层在 AI 供应链中的攻击面^[53-55]。更为严峻的是，上述驱动、容器与编译器等关键供应链组件多处于闭源或半开源状态，用户与研究者难以对其内部实现进行独立审计，使漏洞发现与修复高度依赖厂商响应，

一旦攻击者率先掌握可被稳定利用的漏洞，便可能借助硬件加速层对上层 AI 框架与应用产生连锁影响。

从技术研究角度看，现有国内外学术工作主要从 GPU 架构分析与漏洞利用两个方面对硬件加速层展开系统性研究。在 GPU 架构分析方面，研究者通过微架构测试与逆向工程方法，对不透明或半透明的 GPU 内部实现进行了深入探索。Jia 等人率先对 NVIDIA Volta 架构 GPU 的缓存层次结构与访存机制进行了系统分析^[56]，随后又扩展至 Turing 架构^[57]。此后，多项工作采用类似方法对 NVIDIA 不同代 GPU 架构进行逆向分析，旨在理解其内部设计与安全边界^[58-60]。

在漏洞利用方面，针对 GPU 的攻击研究主要集中于侧信道 (Side-channel Attacks) 与隐蔽信道攻击 (Covert Channel Attack)。Naghibijouybari 等人首次证明，基于 OpenGL 或 CUDA 的间谍程序可以通过 GPU 侧信道提取网页指纹、用户交互行为，甚至恢复其他 CUDA 应用中神经网络模型的内部参数^[61]。Zhang 等人进一步逆向了 Ampere 架构 GPU 的页表实现细节和多级缓存 (Cache) 的实现细节，并指出在多实例 GPU 特性 (Multi-Instance GPU, MIG) 场景下，由于 L3 Cache 共享机制仍然存在跨实例侧信道风险^[62]。Nayak 等人利用统一虚拟内存 (Unified Virtual Memory, UVM) 和快表 (Translation Lookaside Buffer, TLB) 机制，在 GPU 上构建隐蔽信道，实现了对 GPU 加速数据库应用数据的泄漏^[63]。此外，Dutta 等人利用 GPU 与 CPU 之间共享缓存与总线的特性，在 Intel 平台上构建了高带宽隐蔽信道，进一步拓展了跨硬件组件攻击的可能性^[64]。

在内存漏洞分析方面，已有研究揭示了 GPU 内存管理机制中存在的多种安全隐患。Guo 等人对 NVIDIA GPU 上的越界访问 (Out Of Bound, OOB) 漏洞进行了系统性分析，证实了 GPU 上 OOB BUG 利用的可能性，他们还对 GPU 栈内存布局进行了逆向工程^[65]。Mittal 等人对 GPU 漏洞进行了全面综述，从数据泄露、侧信道与隐蔽信道等角度对攻击模式进行了系统分类^[66]。Miele 等人利用 GPU 上的栈溢出漏洞劫持函数指针，并分析了在 GPU 环境中实施返回导向编程攻击 (Return-Oriented Programming, ROP) 的可行性^[67]。此外，Park 等人提出的 Mind Control 攻击通过操纵 GPU 设备内存并利用固定 CUDA 库地址干扰深度学习系统推理过程^[68]；Sorensen 等人提出的 LeftoverLocals 攻击，利用未初始化的 GPU 局部内存实现跨进程或跨容器的数据恢复，他们的研究恢复了 Apple、Qualcomm 和 AMD 等厂商的 GPU 上的局部内存数据，对其他用户的交互式大语言模型会话进行窃听^[69]。Roels 等人进一步研究了 GPU 内存中的 ROP 小组件，并

提出了绕过 NVIDIA 将返回地址存储在寄存器中的防御机制的组合式攻击方法^[70]。

综上所述，现有研究从硬件架构、配套驱动和工具链软件，以及漏洞利用等多个维度系统揭示了硬件加速层的供应链在安全性方面的潜在风险。然而这些工作大多聚焦于单点漏洞、特定攻击技术或底层实现缺陷，并且仅仅将安全边界界定于 GPU 或者 NPU 等硬件加速设备，将其设为孤立的攻击目标，并未深入分析跨设备的安全性，例如是否可以从 GPU 侧威胁到 CPU 侧的安全性，再由此通过框架与运行时接口向上层 AI 系统传导安全影响。

1.3 本文研究内容与贡献

针对当前 AI 系统在多层次架构中逐渐显现的供应链安全风险，本文从系统整体视角出发，对 AI 系统的软件应用层、模型框架层以及硬件加速层三个关键层级开展了系统性的安全分析。针对当前 AI 系统在多层次架构中逐渐显现的供应链安全风险，本文从系统整体视角出发，对 AI 系统的软件应用层、模型框架层以及硬件加速层三个关键层级开展了系统性的安全分析。通过对上述三个层级的深入研究，本文不仅弥补了现有工作在跨层级系统性分析方面的不足，还在每一层级中发现了此前尚未被充分认识的安全问题，并提出了具有实践意义的分析方法与防护思路，从而构建了一套覆盖 AI 系统全栈的安全研究体系，为理解和保障 AI 系统在真实部署环境下的安全性提供了重要的理论与实践参考。

具体而言，本文的研究内容和主要贡献可概括为以下三个层级。

软件应用层：细粒度模块级依赖冲突安全分析。在软件应用层，现有研究主要关注依赖配置错误、已知漏洞传播以及漏洞检测等安全问题，且多以软件包为分析粒度。然而，在以 Python 为代表的 AI 软件生态中，大量项目由复杂的模块级依赖关系构成，不同软件包在安装与运行阶段可能产生细粒度的模块冲突行为，其安全影响尚未得到系统性研究。

为填补这一研究空白，本文围绕 Python 生态中的模块级依赖冲突问题开展了系统研究。首先，本文从 GitHub、Stack Overflow 等开发者社区出发，采用滚雪球式的数据收集方法，系统整理并分析了大量与模块冲突相关的真实事件，对模块冲突的触发场景、表现形式及潜在影响进行了实证研究总结^[71-72]。在此基础上，本文进一步对整个 Python

开源生态 (The Python Package Index, PyPI) 生态中的全部软件包进行了大规模模块收集与分析，系统识别可能存在模块冲突风险的软件包及其潜在影响范围。最后，本文以真实世界的 AI 项目为对象，对 GitHub 上广泛使用的热门 AI 项目进行了深入分析，评估模块冲突问题在实际 AI 软件构建与运行过程中的安全影响。

围绕软件应用层的模块级依赖冲突问题，本文的主要贡献包括：

- **新攻击面：**本文揭示了一种软件应用层的新型攻击面——模块替换攻击。该攻击利用 Python 软件包在安装阶段将不同包的模块部署至同一默认路径的机制，通过构造同名模块引发冲突，从而干扰 AI 软件的正常执行，甚至实现任意代码执行等安全威胁。
- **新技术：**为支撑大规模生态分析，本文提出了两项关键技术：InstSimulator 与 EnvResolution。InstSimulator 通过 AST 解析与动态安装模拟相结合的方式，解决了软件包源代码结构与实际安装后模块布局不一致的问题；EnvResolution 则通过环境语义建模与免下载依赖解析机制，有效提升了依赖图构建的准确性与分析效率。
- **大规模实证研究：**基于上述技术，本文实现了 ModuleGuard 框架，对 PyPI 生态中 43 万余个软件包、420 万余个版本进行了系统性的模块依赖与冲突分析，系统总结了模块冲突问题的成因、特征及其在生态中的分布情况。
- **真实世界影响分析：**利用 ModuleGuard，本文进一步分析了 GitHub 上 3,711 个真实热门 AI 项目（涵盖 93,487 个版本标签），发现其中 108 个高星项目存在实际的模块冲突风险，并已向相关开发者报告问题并提供修复建议。

本部分研究的详细内容见本文第三章，相关代码来源于网站 <https://sites.google.com/view/moduleguard>，研究成果发表于 ICSE (CCF-A 类) 国际软件工程顶级学术会议。

模型框架层：基于合法 API 能力滥用的函数级攻击范式。在模型框架层，现有研究通常将恶意模型视为参数投毒或模型逻辑篡改问题，关注模型输出异常或性能退化等结果性表现。尽管已有少量工作尝试将模型本身作为恶意载体，但其攻击方式往往依赖不安全序列化或显式代码注入，不仅容易影响模型精度，也较易被现有检测工具发现。

随着深度学习框架功能的不断扩展，其所提供的大量高权限、通用型 API 已逐渐具备超出模型计算本身的系统交互能力。然而，这类“合法 API 所隐含的安全风险”在现有研究中尚未得到系统性分析。针对这一问题，本文以 TensorFlow 框架为研究对象，系统分析了其框架 API 的持久化机制及能力边界，提出了 TensorAbuse 攻击模型，揭示攻击者如何在不依赖传统漏洞的情况下，仅通过滥用框架提供的正常 API，将恶意行为嵌入 AI 模型之中，从而在模型运行阶段触发系统级攻击。

围绕模型框架层的 API 能力滥用问题，本文的主要贡献包括：

- **新攻击面：**本文提出了模型框架层的新型攻击范式——TensorAbuse。该攻击利用深度学习框架 API 自身具备的文件访问与网络通信能力，在不影响模型精度的前提下，实现任意代码执行、文件窃取等系统级攻击行为。
- **新技术：**本文提出了 PersistExt 与 CapAnalysis 两项关键技术，用于自动化提取并分析框架 API 的潜在高风险能力。PersistExt 结合静态分析与启发式规则，系统提取可被持久化至模型中的 API 及其跨语言调用链；CapAnalysis 则借助大语言模型能力，对 API 的潜在系统交互能力进行自动化判定。
- **真实世界影响验证：**基于上述技术，本文从 TensorFlow 中识别出 1,083 个可嵌入模型的 API，并进一步筛选出 31 个可用于构造恶意行为的高风险 API，构造了五类攻击原语与四类完整攻击模型，成功绕过 Hugging Face、ModelScan 等主流检测工具，并获得厂商认可与安全奖励。本研究向这些厂商提供相关利用的详细代码，帮助他们构建能力更强的模型扫描工具，同时收获了 1,650 美元的奖励。
- **恶意模型检测工具：**在攻击分析的基础上，本文实现并开源了一套恶意模型检测工具，可对 TensorAbuse 及相关隐蔽嵌入方式进行自动化检测并生成分析报告。

本部分研究的详细内容见本文第四章，相关代码开源于 <https://github.com/ZJU-SEC/TensorAbuse>，研究成果发表于 IEEE S&P (CCF-A 类) 网络安全领域国际顶级学术会议。

硬件加速层：GPU 内存地址随机化安全机制分析与跨设备攻击。在硬件加速层，GPU 等专用计算设备已成为 AI 系统不可或缺的基础设施。尽管已有大量研究对 GPU 架构、侧信道攻击以及驱动与固件漏洞进行了深入分析，但现有工作大多将 GPU 视为相对独立的计算单元，主要关注 GPU 内部的安全问题，而缺乏对 GPU 与 CPU 等异构

计算设备之间安全关联性的系统研究，尤其尚未从供应链视角分析 GPU 侧安全机制失效对整个 AI 系统的潜在影响。

针对这一不足，本文从 GPU 侧防护机制的安全性出发，重点分析了 NVIDIA GPU 中地址随机化 (Address Space Layout Randomization, ASLR) 机制的设计假设与实际实现之间的差异。本文发现，GPU ASLR 在实现层面存在非随机区域、粗粒度随机化以及地址相关性等问题，使得攻击者仅需利用 GPU 侧的简单内存安全漏洞（如越界读），即可泄露 GPU 内部的敏感地址信息。进一步地，本文系统分析了 GPU 与 CPU 在统一虚拟内存和物理内存映射机制下的交互关系，揭示了 GPU 地址泄露如何被用于逐步推断 CPU 侧地址空间布局，最终首次实现了从 GPU 出发影响 CPU 地址随机化安全性的跨设备攻击路径。

围绕硬件加速层的 GPU 地址随机化机制，本文的主要贡献包括：

- **新攻击面：**本文揭示了一种硬件加速层的新型攻击面。攻击者可利用 NVIDIA GPU 中地址随机化机制的弱点，通过 GPU 侧的 OOB 漏洞泄露关键地址信息，并进一步借助 GPU 与 CPU 地址空间之间的相关性，推断 CPU 侧 glibc 等关键段的地址布局，从而削弱 CPU 侧 ASLR 的安全性。
- **新技术：**本文提出了 FlagProbe 与 AnchorTrace 两项关键逆向分析技术，用于在缺乏文档支持的黑盒环境下恢复 NVIDIA GPU 虚拟地址空间的语义结构。FlagProbe 通过启发式标志插入与访问行为分析，恢复不同 GPU 内存区域的语义；AnchorTrace 则利用系统常量内存作为锚点，递归追踪稳定指针链，从而高效收集 GPU 侧随机化地址信息。这两项技术共同构成了一套可复用的 GPU 内存布局分析框架。
- **新发现：**本文系统揭示了多个此前未被公开披露的 GPU 内存布局特性：多个 CUDA 进程共享关键 GPU 内存区域；GPU 与 CPU 在不同权限模型下映射相同的物理内存页；GPU 堆内存完全缺乏随机化，其余内存区域采用统一的粗粒度随机偏移；此外，部分 CPU 地址空间（如 glibc）与 GPU 地址空间之间仅存在极低熵差异。这些特性共同削弱了 GPU ASLR 的防护效果，并为跨处理器攻击提供了现实基础。
- **跨设备攻击验证：**基于上述发现与技术，本文发现了一个 GPU 缺陷。基于该缺陷，本文构建并验证了一条完整的跨设备攻击路径，展示了攻击者如何从 GPU 侧漏洞

出发，逐步破坏 GPU ASLR，并进一步推断 CPU 地址随机化信息，从而对整个 AI 系统的安全性产生连锁影响。相关缺陷已经汇报给 NVIDIA 官方，并已经在 570 版本驱动中修复。

本部分研究的详细内容见本文第五章，相关代码开源于 <https://github.com/ZJU-SEC/NvidiaASLR>，研究成果发表于 IEEE S&P (CCF-A 类) 网络安全领域国际顶级学术会议。

1.4 本文组织与章节安排

2 AI 系统供应链背景知识

随着 AI 技术的大力发展，其被应用在现代社会的多个关键领域，并成为当前时代下的核心生产力。在这一过程中，AI 系统也由原本的单一推理模型，轻量级数据和简单应用交互的特点演进为由软件应用、模型框架、硬件加速平台等多层组件共同构成的复杂系统。在这一结构下，AI 系统的开发、训练、部署与运行高度依赖第三方代码库、预训练模型、运行时框架以及异构硬件与驱动程序，逐步形成了一条跨越多个技术层级与信任边界的 AI 系统供应链。相比于传统软件供应链，AI 系统的供应链结构更加复杂，也引入了大量的新型攻击面。本章首先对 AI 系统供应链的整体架构进行梳理，从软件应用层、模型框架层以及硬件加速层三个关键层级出发，介绍各个层级的供应链基本组成，功能职责以及所面临的威胁，从而为后续章节针对不同层级展开的安全分析与攻击奠定背景基础。

2.1 软件应用层背景知识

在 AI 软件开发过程中，为了显著降低研发成本并加速功能迭代，开发者通常遵循“避免重复造轮子”的工程实践，广泛复用已有的开源代码、第三方组件以及成熟的软件开发工具链。与传统软件相比，AI 软件在数据处理、数值计算与模型推理等方面对功能组件的依赖更为密集，这使得其软件应用层天然构建在一条高度依赖外部资源的供应链之上。

2.1.1 软件供应链复杂性

总体而言，AI 系统的软件应用层供应链主要由两条相互交织的子链路构成：一方面是开源组件自身的开发、演化与分发过程；另一方面是 AI 软件在开发、构建与维护过程中对外部代码或组件的引入和复用过程。如图 2.1 所示，这两条链路共同构成了 AI 软件应用层的基础生态，并在实际工程实践中紧密联系。

开源组件开发过程链路。在组件开发阶段，组件开发者通常围绕通用功能或特定需求实现可复用的软件模块，并将其发布至公共的开源组件库中，供其他开发者下载与使

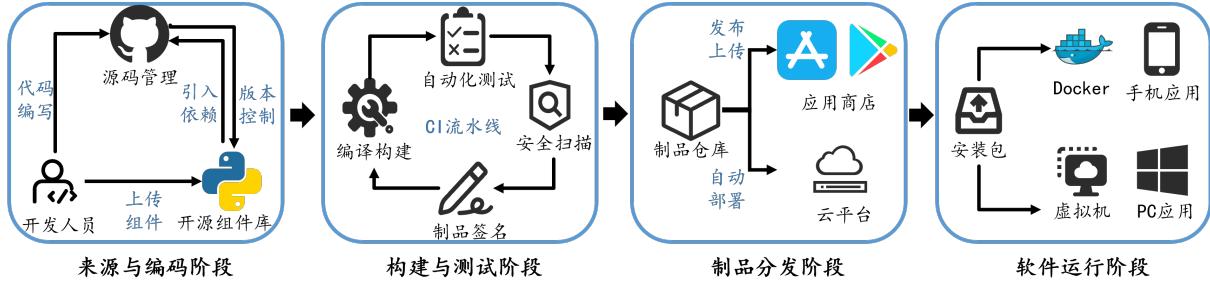


图 2.1 软件应用层供应链架构图

用。与此同时，组件开发者自身也往往依赖已有的第三方组件，通过直接引入、二次封装或定制化修改的方式完成新组件的开发。因此，组件开发过程本身同样嵌套在更上游的组件供应链之中。不同编程语言通常对应不同的组件分发生态。例如，Python 语言主要依赖 PyPI，Java 语言对应 Maven Central，Rust 语言则以 Crates.io 作为官方组件仓库^[73-75]。这些集中式组件库极大地降低了依赖获取与分发成本，使得复杂 AI 功能可以通过少量依赖声明快速集成。然而，与之相伴的是依赖规模和复杂度的持续增长。对于 C/C++ 等底层语言而言，由于其发展历史较早、应用场景高度多样，整体生态呈现去中心化特征，缺乏统一的官方组件仓库。这类语言通常依赖系统原生库（如 GNU C Library）或厂商提供的专有库（如 NVIDIA CUDA-X 库）^[76-77]。近年来，尽管出现了诸如 Conan 等第三方包管理工具，但其应用范围与生态成熟度仍相对有限^[78]。这种多样化的分发模式进一步增加了 AI 软件应用层供应链的异构性。

AI 软件的开发过程链路。在 AI 软件的实际开发过程中，软件开发者通常借助源代码管理工具对项目进行协作开发与持续维护。除通过包管理器引入第三方开源组件外，开发者还可能直接采用源码克隆、代码片段复用等方式，将外部项目中的实现集成至自身代码库中，从而进一步缩短开发周期。这一过程中，版本控制系统与代码托管平台成为软件应用层供应链的关键基础设施。当前，Git 已成为事实上的标准版本控制工具，而 GitHub、GitLab、Gitee 以及企业内部代码托管平台则承担了代码协作、审计与发布的重要角色。在软件开发完成后，构建与打包工具被用于生成可部署的软件形态，例如可执行二进制文件、安装包或容器镜像，并最终交付给终端用户。

依赖解析的复杂性和动态性。在上述两条供应链链路的共同作用下，AI 软件应用层逐步形成了结构复杂、层级深度较大的依赖关系网络，通常以依赖图的形式进行刻画。如图 2.2 所示，albumentationsx 软件在 2.0.13 版本中依赖多个第三方组件，并进一步引

入多层间接依赖^[79]。在一个依赖图中，单个软件包通常对应一个特定版本，但在某些语言生态（如 JavaScript）中，同一依赖图中可能同时存在同一组件的多个版本。每一个软件包既可能作为直接依赖被上层软件显式引入，也可能作为间接依赖隐藏在更深层的依赖链路中。例如，图中 numpy 2.2.6 版本既是 albumentationsx 的直接依赖，同时也是其经由 scipy 1.16.3 引入的间接依赖。此外，AI 软件的依赖图并非静态不变。依赖版本往往通过范围约束进行声明，例如在 albumentationsx 的配置文件 setup.py 中指定的一条依赖为 numpy $\geq 1.24.4$ ^[80]。在此情况下，依赖解析工具通常会选择满足约束条件的最新版本。一旦上游组件发布新版本，软件在重新构建或安装时，其依赖图结构便可能随之发生变化，从而引入新的行为差异。

依赖图的动态性也使得依赖解析算法变得极其复杂。以 Python 生态为例，用户通常通过官方包管理器 pip 安装第三方依赖组件^[81]。在安装过程中，pip 会解析配置文件中的依赖声明，并从 PyPI 获取候选版本列表，依据版本约束与兼容规则逐步选择合适的版本进行下载与安装。该解析过程会递归地处理新引入的依赖，并在出现冲突时进行回溯与重新选择。由于该过程采用边解析边安装的策略，依赖规模较大时往往面临解析效率低下、冲突频繁等问题，甚至可能因为依赖地狱或者依赖冲突问题导致整个依赖解析过程失败^[82-83]。这种高度动态且依赖密集的安装机制，使得 AI 软件应用层在功能灵活性提升的同时，也不可避免地引入了更复杂的且难以全面发现的供应链安全风险。

2.1.2 软件供应链对 AI 系统的威胁与防护

由于软件供应链普遍存在的无条件信任机制，加之依赖解析过程的复杂性与动态性，AI 系统在应用层面临着多维度的安全威胁。从 AI 软件生命周期视角来看，主要分为四个大类，代码来源威胁，基础设施威胁，软件运行时威胁和依赖传播威胁。

代码来源威胁与防护。代码来源威胁主要源于三个维度：一是组件或软件开发人员编写代码时因疏忽引入的逻辑缺陷或者安全漏洞；二是攻击者通过劫持开发者账户或者非法获取仓库权限实施的恶意投毒；三是开发者滥用大语言模型生成的代码而引入的衍生风险。在开发者编写代码阶段，漏洞通常源于开发者编码时的错误，或者是通过代码克隆将别人包含漏洞的代码无意地引入到开发代码中的缺陷，此类漏洞往往具有极强的隐蔽性和较长的潜伏周期。例如潜伏于 Linux 内核 TCP 系统中的高危漏洞 CVE-2024-36904，允许攻击者利用该漏洞获得在内核权限下执行任意代码的能力，尽管

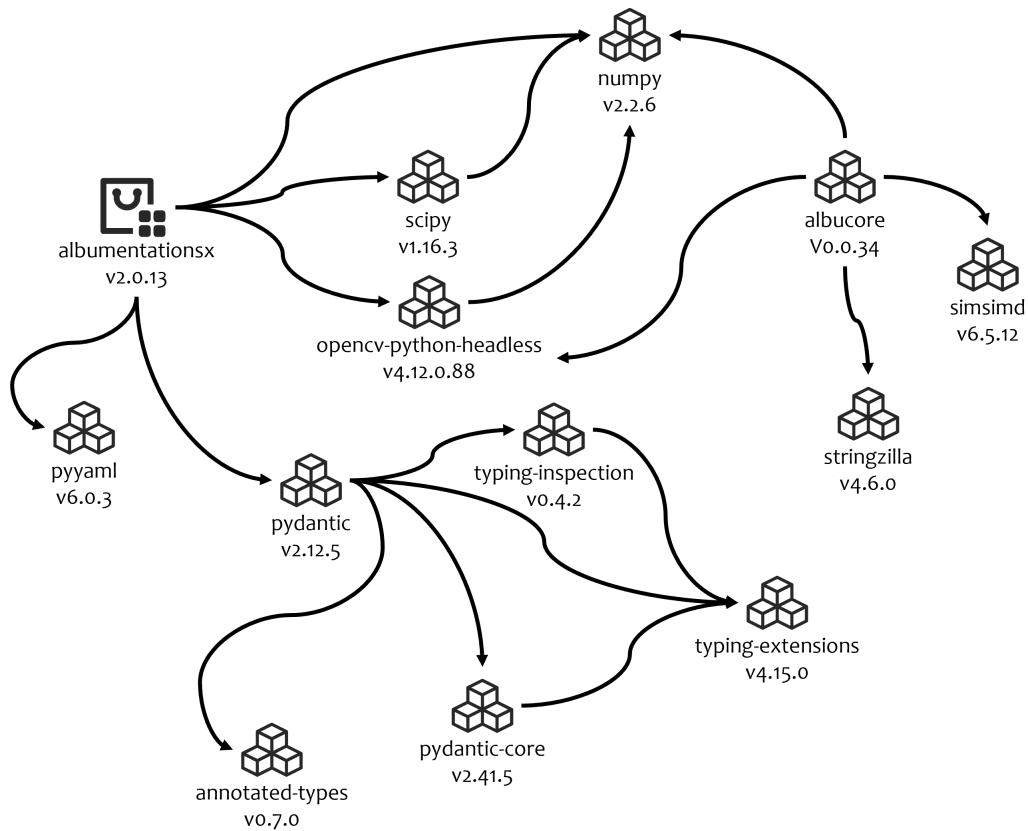


图 2.2 albumentationsx 软件 2.0.13 版本依赖图

该缺陷在 2017 年就已经存在，但直到 2024 年才被发现并披露，在此期间，任何依赖受影响内核版本的系统均暴露于潜在攻击风险之中^[84]。在攻击者控制账户恶意投毒方面，恶意攻击者常利用钓鱼攻击、账户密码破解、账户重注册接管以及社会工程学等手段获取到开发者账户或者代码提交权限，并直接向上游项目植入恶意逻辑，此种攻击方式往往范围较广，且容易发现。较为著名的是 2025 年知名开发者 qix 遭受恶意邮件钓鱼，导致其账户下维护的 18 个超高下载量组件全部遭受恶意混淆代码植入，影响范围极大^[85]。也有原本开发者是良性由于某种特别原因投毒，比如 node-ipc 开发者因为俄乌战争反俄故意投毒事件^[86]。此外，随着 LLM 在软件开发中的普及，AI 幻觉及训练语料中既有的缺陷导致生成的代码可能携带安全隐患。CodeRabbit 2025 年的报告指出，在其分析的开源项目中，由 AI 参与编写的项目比例极高，而由此引入的硬编码凭证、不安全对象引用等漏洞导致安全风险增加了 2.74 倍^[87]。

对于此类威胁的防护，企业往往采用代码审查，代码仓库的安全配置和完整性校验等方式进行防护。代码审查指的是在软件开发的全流程中通过威胁建模，代码审计等方

式对编写的代码进行安全检查，比如在编码阶段和代码提交阶段采用静态分析工具或者动态模糊测试工具进行漏洞检测，或者通过人工的方式对引入的组件和第三方代码片段进行来源审查，安全评估等。代码仓库安全配置用以防护攻击者的恶意投毒，旨在启用多重验证 (Multi Factor Authentication,MFA) 等方式，并且对软件实行依赖来源验证和校验和来防止自身软件被恶意篡改。而对于 AI 生成不安全的代码引入的风险，采用的方式往往也是代码审查，将 AI 生成的代码视为初级开发者代码，并对其进行加强审计，同时对 AI 生成的命令行等命令进行核实，此外还应禁止关键的安全逻辑代码交由 AI 来生成，明确使用 AI 代码的范围和边界。

基础设施威胁与防护。 AI 系统的软件应用层来自于基础设施威胁的主要有两方面：一个是利用持续集成和持续部署的管道 (continuous integration and continuous deployment, CI/CD) 构造的威胁，一个是利用编译器或者编译平台构造的威胁。攻击者可以利用 CI/CD 管道中集成的工具链投毒的方式使得攻击者使用和运行这些工具时执行恶意代码，或者在 CI/CD 部署脚本中嵌入恶意逻辑，或者是脚本编写者自身不注意写入的错误逻辑，使得构建平台能够运行这些脚本中的恶意逻辑，此外，更为严重的是，这些脚本中还可能包含一些密钥在内的敏感信息，容易被人窃取。例如 CVE-2020-14188 表明，Atlassian 在 GitHub 上提供了 atlassian/gajira-create 脚本，以帮助开发者从不同用户处跟踪问题报告，然而不幸的是，该脚本在处理插值的属性数据时，额外引入了一个模板引擎，从而导致了一个可由用户输入控制的任意代码执行漏洞^[88-89]。在利用编译器和编译平台方面，攻击者可以利用恶意篡改的编译器，例如 2015 年著名的 xcodeGhost 事件，攻击者利用国内下载 Xcode 速度慢的特点，在网盘上传了下载好的 xcode 供人下载，但是该版本的安装包被人嵌入了恶意 xcodeGhost 框架，使得使用该编译器编译出的 app 会自动执行非开发者预期的功能，包括唤起电话，发短信等^[90]。攻击者还可以通过构建平台的恶意代码注入进行攻击，较为著名的是 2020 年 Solarwinds 攻击，攻击者未经授权访问了 SolarWinds 的构建平台，该平台用于编译和打包 SolarWinds Orion 软件。此脚本将恶意代码注入编译后的 SolarWinds Orion 软件中。当用户安装受感染的软件时，恶意代码会在他们的系统上执行，从而使攻击者能够未经授权访问他们的系统。攻击者还能够从他们的系统中窃取敏感数据，例如凭证、知识产权和客户信息^[91]。

防御此类攻击的方式主要是使用可信来源的构建系统或者构建工具，并且将 CI/CD 的基础设施，包括脚本和工具也加入安全边界，作为审查内容的一环。此外对于入侵构

建系统方面，往往采用最小权限原则限制构建系统的访问权限，并且为不同的构建服务设置不同的构建虚拟环境，将他们进行隔离。最后，采用不同的安全扫描工具来防止脚本的隐私泄露也是一个较为不错的方法。

软件运行时威胁与防护。软件运行时威胁和代码来源威胁较为相似，但是软件运行时威胁更侧重于一个动态的过程，其包括来自于运行环境的威胁和错误配置的威胁。来自于运行环境的威胁来源于两方面，一方面，由于依赖解析或者依赖声明的错误，可能会导致软件在构建编译时引入包含漏洞的组件版本或者人为恶意投毒的版本，从而在运行阶段触发安全问题，这种情况最典型的攻击为依赖混淆攻击，即攻击者向公用开源软件仓库（如 npm, PyPI, Maven 等）上传一个名称和一些企业内部私有软件依赖包名字相同的，且软件包版本号更高的恶意公共包，当公司内部构件项目时，依赖解析工具（如 npm, pip, mvn 等）就会错误的使用版本号更高的远程恶意软件包而非企业内部的私有包，从而导致企业内部遭遇攻击^[92]。另一方面，AI 软件通常运行于容器，云环境或者虚拟机中，若底层操作系统或者容器配置存在漏洞，攻击者便可以利用这来达到提权的目的，例如，如果云环境中的宿主机的内核版本中存在 Dirty Pipe 漏洞（CVE-2022-0847），恶意的 AI 程序就可以利用管道机制的缺陷和多个 AI 服务通常共享一个宿主机的机制，覆盖宿主机内核页的缓存，并向主机的只读文件中写入任意内容^[93]。而来自于错误配置的威胁往往是由于容器的配置错误或者 AI 软件的一些配置错误导致的攻击，以大模型运行框架 Ollama 为例，其在默认配置中并未包含身份验证配置，并且其服务监听的默认地址为 0.0.0.0，这导致其 Ollama 服务会直接暴露在公网之上，使得攻击者可以直接访问推理接口，甚至触发路径遍历漏洞，进而窃取私有模型权重^[94-95]。

针对 AI 软件运行时的防护有效手段较少，针对类似依赖混淆攻击的威胁，现有防护主要依赖于版本锁定，或者采用 DevOps 等软件供应链管理服务来帮助扫描来自于公共库的威胁。而针对容器，云环境所带的漏洞或者错误配置导致的威胁，往往只能依靠及时补丁以及不安全配置的清理来解决，或者采用容器的权限检测和验证工具等方式。

依赖传播威胁与防护。依赖传播威胁关注的是复杂的软件供应链由于依赖关系网的庞大，一个组件遭受风险会使风险扩散，最终导致整个依赖关系网都遭受风险，这种威胁覆盖面广，隐蔽性强，且较难检测。例如 2021 年的 Log4shell 漏洞，由于 Java 生态中 Log4j2 和 log4j 组件存在漏洞，导致依赖该组件作为基础日志框架的数以万计的下游软件均受到影响^[96]。此外，依赖传播往往是隐式的，这是因为 AI 软件往往只在配置文

件中声明直接依赖，而不声明依赖的依赖（间接依赖），这导致即使未引用下游开发者的代码，仍然可能在没有感知的情况下遭受风险。

防护这种依赖传播威胁往往从依赖的可见性和完整性触发，例如可以通过物料清单（Software Bill of Materials, SBOM）和依赖图分析技术来全面刻画软件所直接和间接依赖的软件，而对于已知物料是否存在漏洞需要借助持续监测工具和漏洞扫描工具进行所有组件的扫描，评估高风险节点，并定期打补丁。如果确定某个组件存在漏洞，则还需要通过可达性分析来判断漏洞是否可触发，是否会对 AI 软件产生影响。

3 针对软件应用层的模块冲突威胁分析与防护框架

3.1 引言

随着开源软件生态逐渐扩张, AI 系统以及其相关的各类辅助软件对第三方库(Third-party Libraries, TPLs) 的依赖程度也日益加深, 软件供应链的威胁已成为 AI 软件应用层的核心安全问题之一。由于现代软件开发模式往往高度依赖于第三方组件, 这些第三方组件通过高管理工具实现 AI 软件的环境部署和依赖自动化解析, 这不仅意味着开发者往往在无条件的信任第三方组件, 在引入第三方组件时形成的庞大复杂的依赖树也加剧了软件层的安全问题分析难度, 此时软件应用不仅面临传统意义上的依赖传播和漏洞传播的威胁, 更因为编程语言差异和包管理机制的细节差异, 衍生出了一系列隐蔽且破坏力极强的新型安全风险。

在所有编程语言中, Python 语言凭借其代码简洁性, 易读性以及其强大的生态系统 PyPI, 已经成为 AI 软件开发领域和数据科学领域的首选编程语言。而随着软件供应链技术的快速发展, 开源的第三方库的数量呈现出了爆炸增长的态势。根据 Sonatype2026 报告显示, 仅仅 Python 生态, 在 2025 年一整年中, 组件数量增加了 21.48 万, 版本数量增加了 154 万, 相较于整个生态的 82.13 万个组件和 885 万个版本, 新增组件数占据了 26%, 新增版本数占据了 17%^[97]。然而, 随着软件规模的扩大和依赖关系的日益复杂, 命名空间冲突的问题也日益显现, 且成为了阻碍生态系统健康发展的重要因素。

不同语言针对命名空间冲突问题的处理机制各不相同, 但是它们的目标都是“隔离和唯一性”。Java 生态采用 (groupId, artifactId, version) 形式的三元组来唯一区别一个软件包名, 并且在安装 TPLs 时, 会将不同的软件包安装在以 groupId/artifactId/version 打头的路径中, 例如软件包 log4j 的 2.17.0 版本下载到本地时会被安装在一个相对路径 org/apache/logging/log4j/2.17.0/下, 所以, 只要软件包不同, 其生态中使用模块便不会冲突。因此 Java 生态几乎没有命名空间冲突的问题。相比之下, Rust 生态采用源码共享, 编译期隔离策略, 它将所有下载软件包默认安装到 ~/.cargo/registry/cache/ 目录下, 然后将这些软件包解压后的源码放置在 ~/.cargo/registry/src/ 目录下, 使所有 Rust 项目共享这些源码。在项目编译期间, cargo 会将不同软件包的源码编译到不同的二进制中, 并放

置在项目的路径中，例如项目中依赖了软件包 rand 的 0.8 版本，使用 cargo build 命令会使项目 target/debug/deps 路径中产生名为 rand-ba1d2ca50538ba79.d 的二进制文件，并被链接到项目中使用。由此，Rust 生态也几乎不会存在命名空间冲突的问题。

然而，Python 生态虽然软件包名是唯一索引，然而其模块名却不是唯一，这样的管理机制存在很严重的命名空间冲突问题。首先，Python 生态的第三方软件包默认是以平铺的方式放置在同一个 site-packages 目录下，例如安装 requests 和 urllib3 两个软件包时，pip 会从 PyPI 上下载这个两个软件包的 whl 分发版本，并分别解压到 site-packages 目录下，可是当两个 TPLs 存在两个同名模块时，例如软件包 A 存在 module1.py 的顶层模块，软件包 B 也存在 module1.py 的顶层模块，这两个软件包被同时安装时，由于系统路径中只允许存在一个 module1.py 模块，因为必然会有个软件包的模块将另一个软件包的模块覆盖。其次，Python 项目在引入模块时存在一个搜索优先级机制，其优先搜索当前目录，然后搜索 Python 内置的系统库和动态链接库中的模块，最后搜索第三方模块，假如某个第三方软件包中存在和系统中相同的 sys.py 模块，则项目在使用 import sys 语句引入模块时，永远引入的都是系统中的 sys 模块，而不会引用到第三方软件包的 sys 模块。

本研究将 Python 生态系统中由于命名空间资源重叠而引发的安全问题正式定义为模块冲突（Module Conflict, MC），并针对 AI 系统软件应用层面临的这一严峻威胁，设计并实现了一套名为 ModuleGuard 的自动化检测与防护框架。本章节主要涵盖了以下四个维度的创新性工作：

首先，本研究填补了学术界在细粒度命名空间冲突研究上的空白，完成了业界首次针对 MC 问题的大规模实证研究。通过构建基于 GitHub Issues 与 StackOverflow 等多源异构数据的分析框架，本研究对 97 个真实案例进行了深度剖析，系统性地总结出模块冲突的三种核心模式（模块对库、模块对三方库、依赖内冲突）以及两类现实威胁（模块覆盖与导入混淆），为后续的大规模生态治理奠定了坚实的理论基石。

其次，在威胁建模层面，本研究深入挖掘了 Python 包管理工具（如 pip）在依赖解析与安装过程中的安全缺陷，首次定义了一种针对 AI 系统的新型软件供应链攻击向量——“模块替换攻击”。该攻击利用 Python 扁平化的安装目录结构与“首次匹配”的模块导入机制，使攻击者能够通过构造同名模块路径，在依赖安装阶段静默覆盖目标系统的根本组件，或在运行时劫持正常的模块加载流程。本研究详细论证了该攻击破坏 AI

软件完整性、导致环境不可逆损坏以及在大规模依赖树中潜伏的机理。

在技术实现层面，为了解决大规模软件包模块提取中存在的动态性强、开销大等难题，本研究提出了一套新型模块冲突分析技术与防护框架。其中包括一种名为 InstSimulator 的无安装语义模拟技术，通过静态模拟 `setup.py` 等脚本的执行逻辑，实现了对模块路径的高保真提取；以及一种环境感知型依赖解析算法 EnvResolver，通过引入环境变量约束与优先级策略，有效解决了复杂依赖场景下解析精度低与速度慢的瓶颈。基于上述核心技术实现的 ModuleGuard 框架在基准测试中达到了 95% 以上的检测精度。

最后，本研究利用 ModuleGuard 模块冲突分析框架，进行了大规模的生态分析，其包括分析了截至 2023 年 4 月的整个 PyPI 生态中的 43 万个软件包（包括 420 万个软件包版本），研究不仅揭示了模块冲突随生态演进呈指数级增长的趋势，还发现了冲突包在命名语义上的相似性规律。此外，本研究针对 GitHub 上 3,711 个高星的热门 AI 项目（涉及 93,487 个标签）进行了模块冲突检测，结果显示共有 108 个顶级项目受到供应链模块冲突的影响。本研究通过向开发者报告、工单提交及邮件沟通等方式，协助开发者修复了大量潜在风险，其实用性与有效性得到了开源社区的广泛验证与认可 ModuleGuard 能够有效检测 AI 软件应用层的模块冲突，从而防御来自于模块冲突的威胁。

3.1.1 背景知识

3.1.1.1 Python 软件包层级

Python 生态存在复杂的软件包层级，本节内容旨在阐明 Python 生态中相关的专有名词和供应链的具体过程，如图 3.1 所示。Python 生态使用 pip 作为其官方的包管理器，来帮助用户自动化下载和构建依赖。当用户下载软件包时，只需声明要下载的软件包名称或者版本，在命令行中键入 `pip install <软件包>` 即可下载对应软件包的最新版本，当然用户也可以通过键入 `pip install <软件包><限定符><版本号>` 来下载特定的软件包版本，其中限定符可以是 ==, >, < 等。如图例中所示，当用户需要下载特定的 beautifulsoup4 软件包时，pip 软件会根据 beautifulsoup4 这一项目名称到 PyPI 上寻找项目的所有版本号，在根据对应的限定符和版本号来确定选择具体项目的哪一个版本，图例中由于没有给定限定符，因此默认选择的是 beautifulsoup4 的最新版本，即 4.9.3。然后 pip 会从

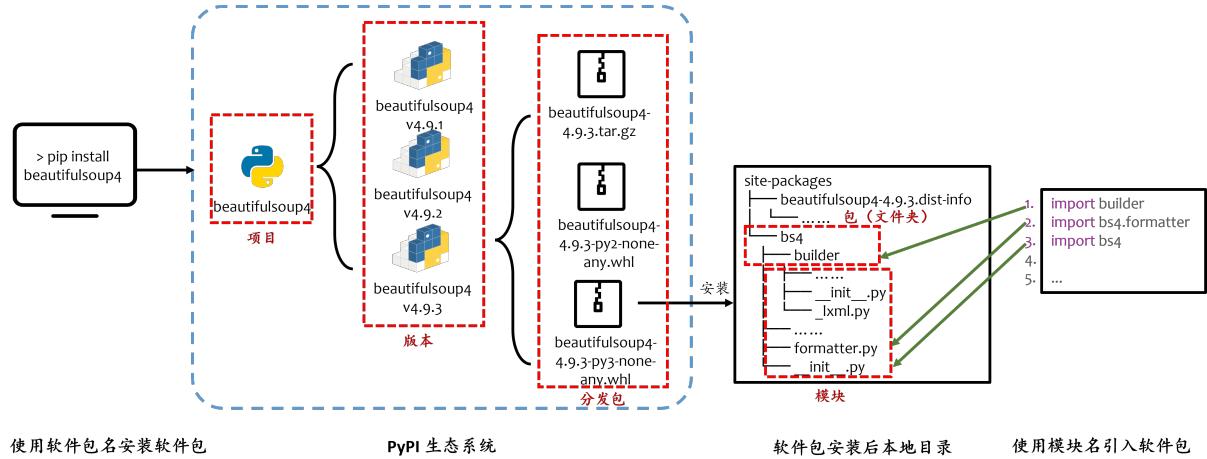


图 3.1 Python 软件包层次结构图

4.9.3 版本的所有分发包里选择一个分发包，然后下载到用户本地，图例中选择的是 whl 版本的分发包，用户下载后将其在本地直接解压到 site-packages 文件夹中，形成最终可引入的包和模块。包是一个文件夹名，而模块是一个.py 文件，其中有个特殊的模块为 `__init__.py`，其模块名不是 `__init__`，而是其所在的文件夹的文件夹名，如图例中所示的 `__init__.py` 的模块名为 `bs4`，所以用户最终可以通过 `import bs4` 语句进行引入。

3.1.1.2 Python 依赖管理

Python 语言于 1991 年首次发布依赖，以简洁易读的语法特性和强大的生态库而沿用至今，然而这也使得 Python 存在极为严重的历史遗留问题，即 Python 的软件依赖管理较为复杂，可以通过多种方式进行依赖声明，甚至是利用代码执行逻辑进行声明，这也使得传统依赖提取不完全，进而导致依赖图解析准确率低。

具体来说，一个 Python 项目，往往通过创建虚拟环境的方式在本地隔离不同项目的依赖，避免版本冲突，即用户通过 `python -m venv <虚拟环境名>` 创建本地环境后并激活该环境，即可在其中任意使用 `pip` 工具而不影响到别的项目。随后用户便可以通过如图 3.2 所示的四种方式进行依赖声明。第一种是文本声明方式，即用户也可以在本地项目中包含一个文本文件来声明依赖，该文本文件的每一行都表示一个依赖，这种方式最原始也最灵活，可以添加注释，也可以添加 `github` 项目或者一个本地路径，用户在安装依赖时只需使用命令 `pip install -r <文件名>` 即可，而文本文件的名字通常是 `requirements.txt`，当然也可以是其他任意的名字，此外用户还可以通过 `pip freeze > requirements.txt` 命令自动地生成依赖文本文件。

```

requirements.txt
1 wheel==0.46.3
2 coverage==7.2.7
3 setuptools==71.1.0;python_version=="3.12"
4 # Requirement for setuptools>=71
5 packaging==24.1;python_version=="3.12"
6
7 # Pytest specific deps
8 pytest==8.1.1
9 pytest-cov==5.0.0
10 atomicwrites>=1.0 # Windows requirement
11 colorama>0.3.0 # Windows requirement

setup.py
1 from setuptools import setup, find_packages
2 requires = [
3     'botocore>=1.42.45,<1.43.0',
4     'jmespath>=0.7.1,<2.0.0',
5     's3transfer>=0.16.0,<0.17.0',
6 ]
7 setup(
8     name='boto3',
9     version="0.1",
10    packages=find_packages(exclude=['tests*']),
11    install_requires=requires,
12    extras_require={"dev": ["flake8"]},
13 )

setup.cfg
1 [metadata]
2 name = boto3
3 version = 0.1
4 [options]
5 packages = find:
6 install_requires =
7     botocore>=1.42.45,<1.43.0,
8     jmespath>=0.7.1,<2.0.0
9     s3transfer>=0.16.0,<0.17.0
10 [options.packages.find]
11 exclude = tests*
12 [options.extras_require]
13 dev = flake8

pyproject.toml
1 name = "boto3"
2 version = "0.1"
3 dependencies = [
4     "botocore>=1.42.45,<1.43.0",
5     "jmespath>=0.7.1,<2.0.0",
6     "s3transfer>=0.16.0,<0.17.0",
7 ]
8 packages = { find = "**", exclude = ["tests*"] }
9
10 [project.optional-dependencies]
11 dev = ["flake8"]

```

文本文件声明方式 setup.py 声明方式

setup.cfg 声明方式 pyproject.toml 声明方式

图 3.2 Python 软件依赖声明的四种方式

第二种方式是 `setup.py` 的传统打包声明方式，该方式依赖于 `setuptools` 工具。这种方式首先需要用户在 Python 文件中引入 `setuptools` 模块的 `setup` 函数，并且调用 `setup` 函数来实现，用户可以通过该函数的 `install_requires` 参数进行声明传统依赖，通过 `extras_require` 声明额外依赖（Extra Dependencies）。由于这种方式是 Python 脚本形式，所以它支持灵活的动态逻辑，这意味着它可以通过字符串构造，正则匹配，读取文件等多种方式构造出一个 Python 的列表来声明依赖。而用户安装依赖只需通过命令 `pip install <项目路径>` 即可，`pip` 会自动解析该项目路径下的 `setup.py` 文件进行动态运行安装依赖。

第三种方式是 `setup.cfg` 的声明式配置方式，这也是一种静态的配置文件，同样需要 `setuptools` 工具配合。与 `setup.py` 不同，`setup.cfg` 采用纯文本的键值对格式（INI 风格），用户无需编写 Python 代码，而是通过在 `[options]` 节中的 `install_requires` 字段声明基础依赖，并在 `[options.extras_require]` 中声明额外依赖。这种声明式（declarative）的设计避免了任意 Python 代码的执行，从而提升了可读性、可维护性以及可分析性，使得工具链（如 `pip`、构建系统或安全扫描工具）能够更容易地静态解析依赖关系。

第四种方式是基于 `pyproject.toml` 的现代依赖声明方式，该方式由 PEP 517、PEP 518 以及 PEP 621 等一系列 Python 官方提案所定义，已逐渐成为当前 Python 生态系统中推荐的标准打包与构建机制^[98-100]。`pyproject.toml` 通过 TOML 格式统一描述项目的构建后端与依赖信息，从而将构建逻辑与 `setuptools` 等具体工具解耦。用户可以在 `[project]` 节中通过 `dependencies` 字段声明基础依赖，并在 `[project.optional-dependencies]` 中声明额外依赖（extra dependencies），而无需再编写 `setup.py` 或 `setup.cfg` 文件。与 `setup.py` 的命令式脚本不同，`pyproject.toml` 完全采用声明式配置，所有依赖信息均以静态数据形式存在，因此构建工具能够在不执行任意代码的情况下直接解析依赖关系。这种设计显著提升了构建过程的可预测性与安全性，同时也便于自动化工具、依赖分析器以及安全扫描系统对依赖图进行静态分析。得益于上述优势，现代包管理工具（如 pip、Poetry）均优先支持或默认使用 `pyproject.toml`，使其逐渐取代传统的 `setup.py/setup.cfg` 方案，成为 Python 项目的事事实标准配置方式。

3.1.2 研究动机与现状

3.1.2.1 研究动机

模块冲突问题已成为破坏真实世界 AI 软件供应链安全与稳定性的严重威胁。如图 3.3 所示，我们通过两个典型的真实案例来揭示其危害机制。

案例一：跨环境模块冲突导致的导入混淆威胁。软件包 `jwt` (v1.3.1) 与 `pyjwt` (v2.6.0) 均包含完全相同的模块路径 `jwt/exceptions.py`。在复杂的开发环境中，这两个包可能分别被安装于系统级的 `site-packages` 目录与用户虚拟环境的 `site-packages` 目录中。当 AI 软件代码执行 `from jwt import exceptions` 语句时，Python 解释器遵循 `sys.path` 的“首次匹配原则”进行模块搜索。系统级路径的优先级往往高于虚拟环境路径，因此解释器会错误地加载系统目录中的 `jwt` 包的模块，而非用户预期在虚拟环境中配置的 `pyjwt` 的模块。这种导入混淆往往导致运行时属性错误或逻辑异常，使得 AI 软件无法正常启动或产生错误结果。

案例二：扁平化依赖安装导致的模块覆盖威胁。pip 在安装 `ysr-monitor` (v0.1.5) 软件时会自动解析并安装其依赖树中所有包含的软件包。然而，其传递性依赖图中存在两个相互模块冲突的软件包：`board` (v1.0) 和 `adafruit-blinka` (v8.20.1)。这两个包在根目录下均包含名为 `board.py` 的模块文件。由于 Python 包管理器采用扁平化的安装策略，即

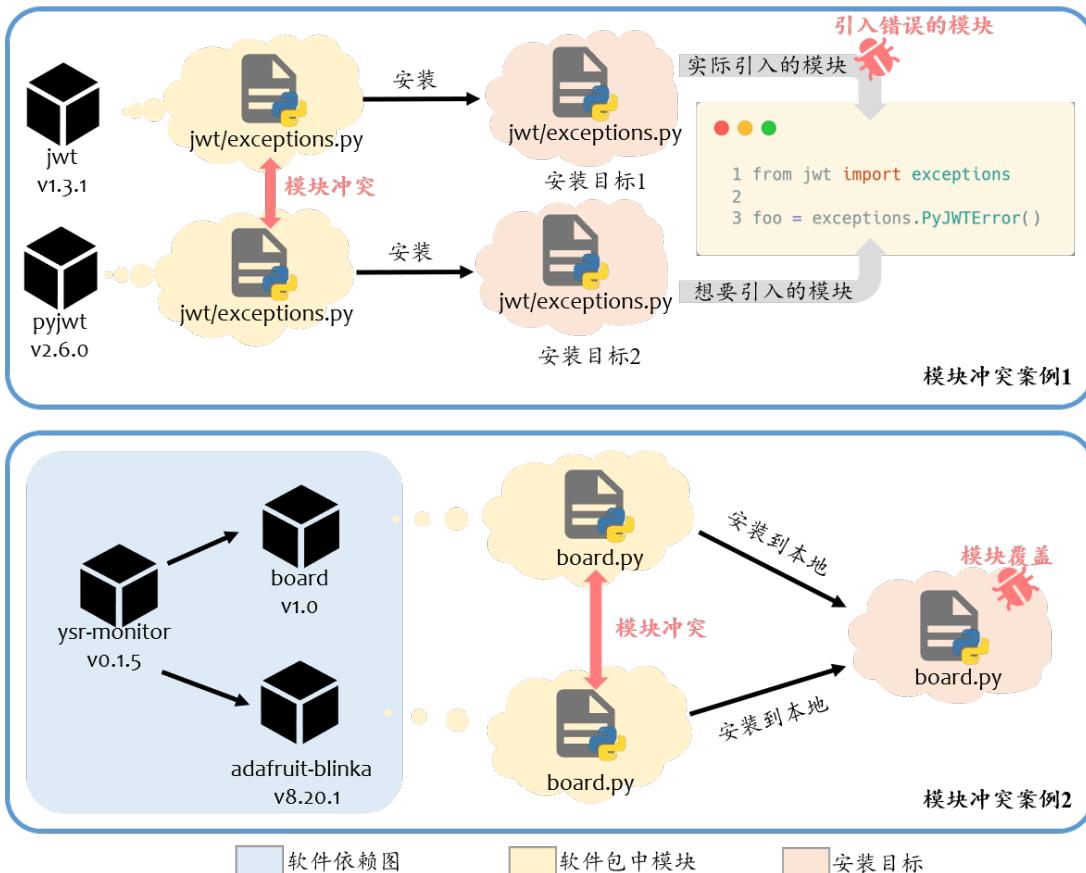


图 3.3 Python 模块冲突真实案例及其影响

默认将所有依赖包解压至同一个 `site-packages` 目录下且不进行物理隔离，这导致后安装的软件包会直接、静默地覆盖先安装软件包中的同名模块。这种模块覆盖行为不仅破坏了原有软件包的完整性，还导致本地运行环境遭受不可逆的损坏，用户将无法再正常调用被覆盖的模块功能。

3.1.2.2 研究现状

针对 Python 生态系统的依赖管理与安全问题，现有工作已经在环境推断、依赖修复以及恶意包检测等方面进行了深入研究。然而，针对本章核心关注的模块冲突问题，现有研究仍然没有很好地进行涉猎研究，在生态系统层面的分析上，现有工作在依赖解析技术和分析粒度上仍存在显著的局限性。

Python 运行环境推断与修复。 针对 Python 项目跨环境部署时因依赖缺失或版本不兼容引发的运行故障，学术界提出了一系列自动化推断与恢复技术。Cheng 等人提出的 PyCRE 旨在利用领域知识图谱推断兼容的运行时环境，然而该方法依赖于执行 pip 安装

命令动态提取模块信息及依赖关系，这种高开销方式严重限制了其扩展性与成功率（仅覆盖 10,000 个软件包），且其采用的最长前缀匹配策略过于简化，无法有效应对复杂的模块冲突威胁^[13]。Ye 等人提出的 PyEGo 则基于知识库自动推断 Python 程序的兼容依赖版本，进一步涵盖了解释器版本与系统库约束，但在处理模块冲突时，PyEGo 倾向于利用 Libraries.io 的 SourceRank 指标选择最流行的软件包以缩减搜索空间，这种基于流行度的假设往往导致错误的依赖归因^[16]。针对特定场景，Wang 等人提出的 SnifferDog 致力于恢复 Jupyter Notebook 的执行环境，但并未考虑模块冲突问题^[101]；而 Garcia 等人开发的 DockerizeMe 能够为 Python 项目自动生成 Docker 镜像文件，但在解决模块归属时同样依赖于软件包流行度指标^[102]。尽管上述工作在环境恢复领域取得了进展，但在应对模块冲突问题上存在根本性局限。这些方法通常基于“模块-包”的一对一静态映射假设，或依赖最长前缀匹配、流行度优先等启发式算法进行简化处理。这种处理方式忽略了多软件包共享同一模块路径的客观事实，导致其难以在复杂软件供应链中检测并修复真实的模块冲突。

依赖冲突管理与修复。现有依赖管理研究主要集中在依赖冲突分析层面，即不同的软件组件需要同一依赖包的不兼容版本。Wang 等人提出了 Watchman 框架，首次针对依赖冲突问题进行了大规模分析，通过收集的 235 个真实的依赖冲突问题总结了依赖冲突的模式，并对整个生态的软件包进行持续监控预测分析依赖冲突问题，帮助开发者进行修复。随后，pip 工具更新了它的依赖解析算法，采用回溯方法来解决依赖冲突问题^[103]。Wang 等人根据新的依赖解析策略进行实证重访，发现新型算法虽然能解决依赖冲突，但是存在效率低下的问题，为此他们提出了 SmartPip 框架，采用离线依赖知识库和 SMT 约束求解策略求解依赖图，并用软件包缓存和软链接机制解决隔离环境空间浪费问题^[82]。Li 等人开发了 EasyPip 静态分析工具，将依赖检测和冲突修复转化为图搜索问题，提出了一种基于图搜索和贪婪算法的修复策略，实现了对原始配置文件的最小化修改^[104]。此外，Pipreqs 和 PyDFix 等工具旨在生成依赖文件或通过日志分析修复安装失败的问题^[14-15]。然而，上述工作与本研究关注的模块冲突存在本质区别。依赖冲突关注的是版本约束的可满足性，而模块冲突关注的是文件系统层面的命名空间冲突。现有工具通常假设只要安装过程没有报错，环境就是健康的。它们忽略了 Python 包管理器在无隔离机制下，后安装的包可能静默覆盖先安装包的同名模块这一关键威胁。此外，Pipreqs 等工具采用的简单映射机制并无法处理不同包包含相同模块路径的复杂情况。

Python 生态系统安全分析。在安全领域，现有研究主要聚焦于恶意软件包的检测，涵盖了针对拼写错误抢注、依赖混淆以及恶意代码植入的大规模分析。Alfadel 等人基于 1,396 份漏洞报告，深入刻画了 PyPI 中安全漏洞的生命周期与传播机制，并揭示了漏洞在依赖链中具有较长的潜伏期以及下游项目修复严重滞后的现状^[18]。在针对解释型包管理器安全性的系统评估方面，Duan 等人提出了一套比较分析框架，定性分析了 npm、PyPI 和 RubyGems 的安全特性差距。他们结合元数据分析、静态分析和动态分析技术构建了恶意包检测流水线，成功在官方开源仓库中发现了 339 个恶意软件包，揭示了解释型语言生态中供应链攻击的严峻现状^[105]。Guo 等人构建了一个包含 4,669 个恶意软件包的多源数据集，对它们进行大规模地实证研究，剖析了恶意代码的生命周期与攻击向量，发现 setup.py 是最主要的恶意代码注入载体，且超过半数的恶意包表现出复合型的恶意行为特征^[106]。Ruohonen 等人对 PyPI 生态中的 19.7 万个软件包进行了大规模实证研究。利用静态分析工具 Bandit，他们发现约 46% 的软件包包含至少一种安全隐患，其中异常处理缺陷和由 subprocess 模块引发的代码注入风险最为普遍^[107]。Vu 等人研究了 Python 生态系统中的潜在拼写错误等攻击向量，但并未深入探讨由于良性或恶意的模块命名冲突引发的系统性风险^[108]。尽管上述工作在恶意包检测和漏洞传播分析方面取得了显著成果，但它们主要聚焦于代码层面的恶意性，例如软件包中存在的后门和挖矿脚本等，并未考虑即便是不包含任何恶意代码的良性包，也可能因模块路径冲突覆盖关键系统组件，导致类似“拒绝服务”或“环境劫持”的严重后果。

综上所述，现有工作主要关注依赖版本的兼容性或安装过程的成功与否修复，对于软件应用层的安全分析也缺乏对安装后模块冲突问题的系统性分析。其技术主要局限性体现在：

- **模块信息提取精度低：**大多工作都是直接从软件压缩包中获取模块名，并且采用模块-包的简单一对一映射，忽略了 Python 包安装过程中复杂的动态行为。
- **依赖解析准确率和效率低：**大多工作仅从单一文件源，如 setup.py 或者 requirements.txt 提取静态依赖信息，未能识别多个动态脚本依赖情况，导致依赖解析准确率低；部分工作采用 pip 安装的手段进行回溯安装，导致大量软件包下载浪费，存在解析效率低下问题。
- **模块冲突分析与防护不全面：**大部分工具都没考虑到模块冲突问题，即使考虑到

存在不同软件包有相同模块的情况，也采用的是最长前缀匹配、流行度优先等启发式算法，并未真正地分析和解决模块冲突问题。

针对这些不足，本章提出的大规模检测框架 **ModuleGuard** 旨在填补这一空白，通过精确的无安装模块提取技术和环境感知的依赖解析算法，系统地揭示并防御 AI 系统软件应用层的模块冲突威胁。

3.2 ModuleGuard 设计概览

针对 Python 软件供应链中普遍存在的模块冲突威胁，本章提出并实现了一套名为 **ModuleGuard** 的自动化分析与检测框架。该框架面向 AI 系统软件应用层，旨在实现从模块信息提取、依赖关系解析到冲突检测的全流程自动化防护。本节将首先剖析在生态系统规模下实现上述目标所面临的技术挑战，进而详细阐述 **ModuleGuard** 应对这些挑战的系统架构与设计细节。

3.2.1 面临挑战

在 Python 生态系统级别实现模块冲突的自动化分析与防护，主要面临以下两个关键技术挑战：

异构分发格式带来的模块信息提取复杂性。首先，Python 软件包分发格式的高度异构性显著增加了信息提取的难度。受限于 Python 生态的历史演进，同一软件包版本往往存在多种分发格式（如.egg, .whl, .zip 等源码包或二进制包），且针对不同的操作系统（如 Windows, Linux）及 Python 解释器版本，软件包的构建方式与文件结构亦存在差异。这种文件格式与配置规范的多样性，加之缺乏统一的模块元数据解析标准文档，使得静态解析并提取模块信息极具挑战性。其次，模块信息具有动态性，即在安装过程中可能受到配置文件（如 setup.py）逻辑的控制而发生结构性变更。如图 3.4 所示，pugs 软件包在安装过程中，其模块目录被重命名为 pugs_lib，而 namespace_pugs/_init__.py 文件则被移除。这种安装前后的状态不一致性，意味着单纯基于静态文件分析的方法难以获取准确的运行时模块视图。

大规模依赖图解析中的效率与准确性权衡难题。本研究旨在对包含超过 420 万个软件包版本的整个 PyPI 生态系统进行全量分析，这对依赖解析算法的效率与准确性提

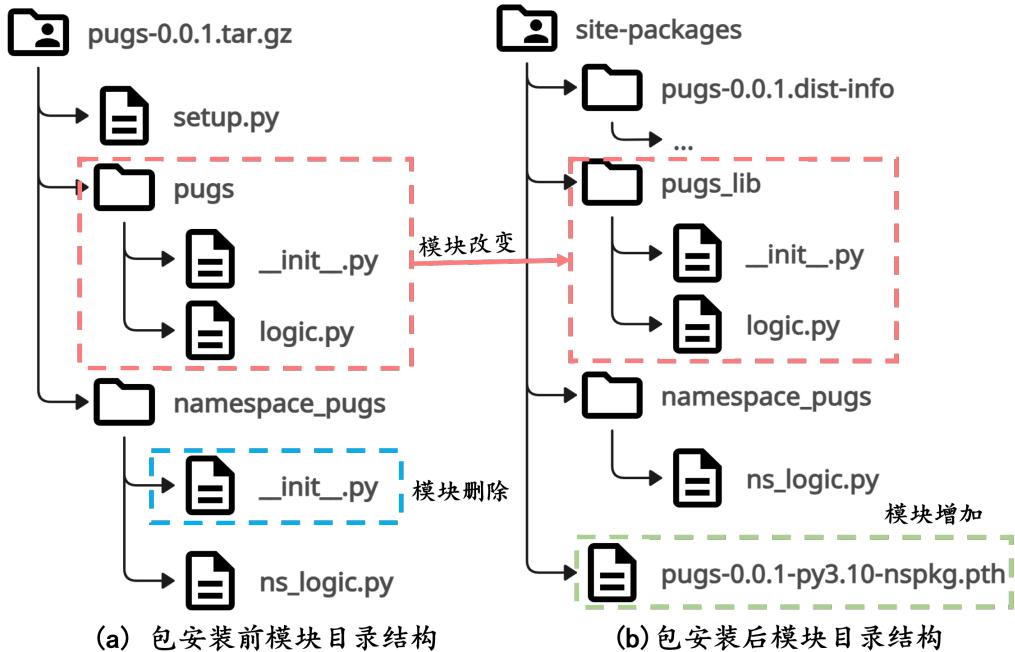


图 3.4 模块在安装前后发生改变案例

出了双重挑战。在准确性方面，现有工作虽尝试利用本地知识库加速和 SMT 约束求解，但其依赖元数据获取方式往往过于片面，通常仅解析单一类型的配置文件（如仅分析源码包中的 `setup.py` 或二进制包中的 `requirements.txt`），导致元数据缺失。此外，Python 依赖关系具有强环境敏感性，现有静态方法常忽略环境标记（Environment Markers）和额外依赖（Extra Dependencies），从而导致解析结果不完整。在效率方面，尽管直接调用 `pip` 进行实际安装可获取准确的依赖图，但由于 Python 包管理器采用回溯算法进行依赖版本仲裁，该过程会触发海量软件包的重复下载与安装试错。在生态系统级别的分析任务中，这种基于动态安装的解析方式将产生极其高昂的时间成本与计算开销，不具备可扩展性。

3.2.2 具体设计

为应对上述技术挑战，本研究提出并设计了两项关键技术：静态安装模拟技术 `InstSimulator`，以及一种具备环境感知能力的依赖解析器 `EnvResolver`。二者协同工作，为 `ModuleGuard` 提供准确且高效的模块与依赖分析基础。

表 3.1 模块路径和依赖声明相关文件和参数列表

分类	文件 (File)	模块相关数据	依赖相关数据
元数据文件	EGG-INFO*	top_level.txt, SOURCES.txt	requires.txt
	egg-info*	top_level.txt, SOURCES.txt, namespace_packages.txt	requires.txt
	dist-info*	top_level.txt, RECORD, namespace_packages.txt	METADATA
配置文件	setup.py	py_modules, packages, package_dir, namespace_packages	install_requires, extras_require
	setup.cfg	py_modules, packages, package_dir, namespace_packages	install_requires, extras_require
	pyproject.toml	py-modules, packages, package-dir	dependencies, optional-dependencies

* 大写的 EGG-INFO 是 egg 类型包的元数据；而 egg-info 是 tar.gz 类型包的元数据； dist-info 是 whl 类型包的元数据。

3.2.2.1 InstSimulator 设计

为解决异构分发格式条件下模块路径信息难以准确提取的问题，本研究提出了一种静态安装模拟技术 InstSimulator。该技术在不实际执行软件包安装过程的前提下，对不同分发格式的软件包进行静态解析与抽象建模，生成统一的配置中间表示，并通过语义化的虚拟文件树模拟，精确恢复软件包在真实安装环境中的模块可见性，从而为后续供应链安全分析提供可靠的模块路径信息。

多源元数据提取。在多源元数据建模方面，为实现对 PyPI 生态中海量异构软件包的全面兼容，本研究首先对 PyPI 生态中与模块及依赖声明相关的元数据参数和文件进行了系统性梳理。本研究首先对 Python 体系的打包、构建与分发机制进行了自底向上的系统性分析。结合对 pip 包管理器底层源码的深度剖析与官方文档规范^[81,109]，本研究采用差分测试方法，通过系统性地变异各类配置文件参数，精准定位了不同元数据字段对模块空间与依赖关系的实际影响链路。基于表 3.1 所示的分析结果，InstSimulator 将软件包的元数据载体归纳为两类：一类为构建过程中自动生成的元数据文件，另一类为开发者显式编写的配置文件。前者通常存在于已编译的分发包中，例如 .whl 与 .egg 格式，其元数据集中存放于 dist-info 或 EGG-INFO 目录下；后者主要存在于源码分发包中，包括 setup.py、setup.cfg 以及 pyproject.toml 等配置文件。

针对已编译分发包，InstSimulator 重点解析能够直接反映安装后文件布局与模块暴露情况的元数据文件。其中，`top_level.txt` 描述了软件包在安装完成后向 Python 运行时暴露的顶层模块名称，`SOURCES.txt` 或 `RECORD` 文件则刻画了分发包中源文件的完整路径集合。依赖信息在不同分发格式中的表达方式存在差异，例如在早期格式中以 `requirements.txt` 的形式存在，而在 `.whl` 格式中则被统一整合进 `METADATA` 文件。通过对上述文件的联合分析，InstSimulator 能够在无需执行构建或安装逻辑的情况下，较为完整地恢复模块文件的空间分布关系。

对于未编译的源码分发包，InstSimulator 以配置文件为主要分析对象。其系统性分析了 `setuptools` 体系下的核心配置机制，明确了与模块路径解析直接相关的关键参数语义，包括用于声明独立模块文件的 `py_modules`，用于指定包级模块集合的 `packages`，用于描述逻辑包名与物理路径映射关系的 `package_dir`，以及用于定义命名空间包结构的 `namespace_packages`。此外，随着 `pyproject.toml` 的引入，部分配置语义被进一步简化和重构，例如将命名空间包声明并入统一的包发现机制。InstSimulator 对上述配置差异进行了统一抽象，以保证不同构建系统和配置风格下的语义一致性。

在解析策略上，InstSimulator 针对不同类型的元数据文件采用差异化的静态分析方法。对于静态文本文件（如 `requirements.txt`）与结构化配置文件（如 `setup.cfg`），系统通过正则匹配与逐行解析的方式将其转换为统一的中间表示；对于具备可执行语义的 `setup.py` 文件，InstSimulator 不直接执行脚本，而是借鉴 PyCD 工具的技术思路，从抽象语法树层面开展数据流与执行流分析，精确提取与模块构建相关的函数调用及参数赋值行为，从而规避动态执行带来的不确定性与安全风险^[110]。所有解析结果最终被归一化为结构化的模块描述集合，并作为虚拟文件树建模的输入。

虚拟文件树模拟。在完成多源异构数据的统一解析后，InstSimulator 引入了内存级虚拟文件树结构。为避免物理磁盘的全量压缩包解压开销与潜在的恶意代码执行风险，该机制采用流式读取方式，按需提取压缩包中的关键元数据与配置凭证，以重构完整的模块路径参数上下文。基于获取的目录结构，InstSimulator 在内存中实例化初始文件树，并根据流式提取配置文件和元数据文件解析取得的模块构建相关参数，精确模拟包管理器在安装阶段对文件系统执行的节点映射、重定向与剪枝操作。在完成所有参数驱动的操作模拟后，InstSimulator 应用深度优先搜索（Depth First Search, DFS）算法遍历该虚拟文件树，从根节点至叶子节点的每一条连通路径即代表一个规范化的模块路径。

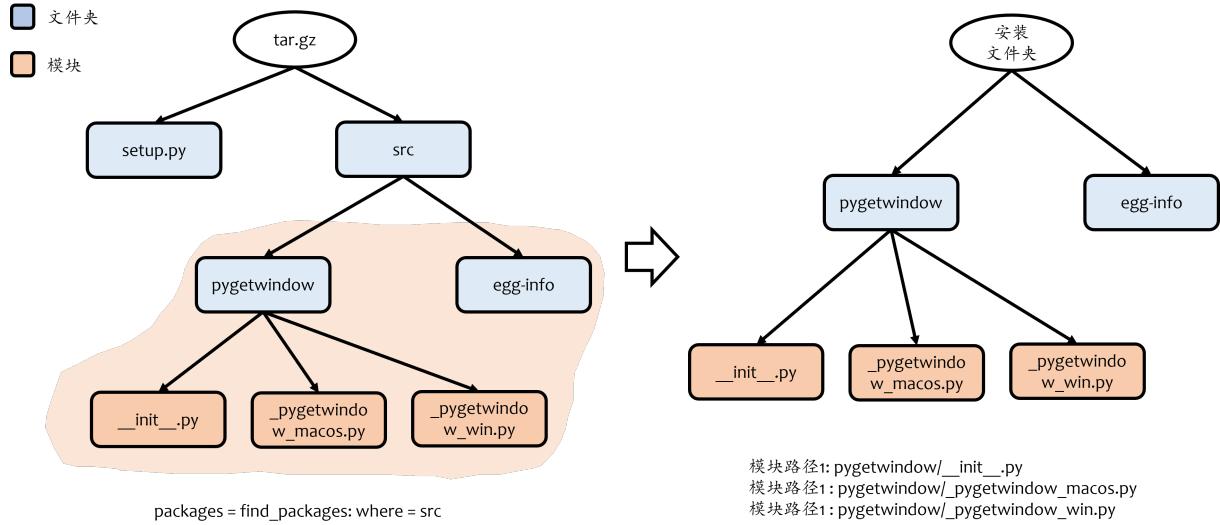


图 3.5 虚拟文件树模拟过程。左侧为原始分发包的物理目录拓扑，右侧为经参数解析与内存级模拟后生成的虚拟模块路径树。

如图 3.5 所示，以 PyGetWindow 软件包 (v0.0.9) 的源码分发包为例，其基于 `setup.py` 声明了 `packages = find_packages(where='src')` 参数，旨在将 `src` 目录下的所有子包部署至目标环境。针对该构建逻辑，InstSimulator 以 `setup.py` 配置文件所在目录为工作目录，运用广度优先搜索 (Breadth First Search, BFS) 算法在虚拟文件树中定位 `src` 节点，并将其子树整体重定向至虚拟安装环境的根节点之下 (如图 3.5 右侧所示)。随后，系统通过广度优先遍历重建后的目录拓扑结构，准确提取出该软件包预期的三条核心模块路径集合。

在处理更为复杂的参数逻辑时 (例如参数为 `package_dir = {'pugs_lib': 'pugs'}` 和 `packages = ['pugs_lib', 'namespace_pugs']`)，InstSimulator 会严格遵循底层构建语义进行节点模拟。其中，`package_dir` 记录了分发包源码目录与逻辑安装目录间的映射关系，而 `packages` 定义了最终需要分发的目标包列表。基于此规则，InstSimulator 首先利用 BFS 算法以配置文件所在层级为工作目录，检索 `package_dir` 中声明的原始节点 `pugs`，并将其在虚拟树中的标识修改为目标逻辑名称 `pugs_lib`。随后，系统执行主动剪枝，剔除所有未在 `packages` 参数中声明的冗余子树。通过上述语义驱动的内存级模拟，InstSimulator 在无需触发任何真实动态安装过程中，构建出与实际安装结果等价的模块文件树，并输出完整且准确的模块路径集合。

3.2.2.2 EnvResolver 设计

为解决 PyPI 生态中软件包数量庞大背景下依赖图解析效率低下以及环境敏感依赖处理不充分的问题，本研究设计并实现了一种环境感知的快速依赖解析器 EnvResolver。该解析器在依赖分析过程中引入本地知识库与启发式解析顺序优化策略，同时综合考虑异构分发格式下依赖信息来源的多样性，并显式支持本地环境信息与额外依赖，从而在保证解析准确性的同时显著提升解析效率。

多维度依赖信息提取。针对现有方法通常仅从单一数据来源提取依赖信息、忽略源码包中依赖声明的问题，EnvResolver 依据 PyPI 软件包的异构性特征，采用多维度的依赖信息提取策略。具体而言，对于 .whl 类型分发包，EnvResolver 直接从 PyPI 官方 API 获取其依赖声明；对于其他格式的分发包，则依据表 3.1 中所列的依赖相关文件与参数，采用与模块信息提取一致的静态分析方法，包括基于抽象语法树的数据流与执行流分析，以提取软件包的直接依赖、额外依赖及其对应的环境约束。考虑到同一软件包版本可能针对不同环境条件提供多种分发包，EnvResolver 会对这些分发包逐一解析，并将依赖信息统一规范化为三元组形式，即 <依赖, 版本约束, 环境条件>。例如，requests 软件包在 Python 版本不低于 3.11 时要求依赖 `numpy>=1.21.0`，而在较低版本 Python 环境下不存在该依赖关系，EnvResolver 将其统一表示为 `numpy>=1.21.0; (python_version >= 3.11)`。

环境信息处理。在环境信息建模方面，本文通过分析 pip 源码确定了 11 类会影响依赖解析结果的本地环境因素，包括 Python 版本、操作系统名称、平台类型以及机器架构等^[111]。EnvResolver 在解析过程中将上述环境信息维护于全局字典结构中。当解析到依赖项包含环境约束时，系统首先判断当前本地环境是否满足其约束条件，从而决定是否将该依赖纳入后续解析过程。此外，EnvResolver 将额外依赖视为一种特殊形式的环境条件进行统一处理。例如，在解析到 `pandas[compression]` 形式的依赖时，系统会在全局字典中增加键值对 `(compression, pandas)`，从而使额外依赖能够以与环境依赖一致的方式参与解析。

依赖图解析优化。为进一步提升依赖解析效率，EnvResolver 对依赖解析顺序进行了系统性优化，以尽可能减少回溯算法的深度与回溯次数。具体而言，EnvResolver 以 resolvelib 框架作为核心回溯解析引擎，并结合本地知识库机制加速解析过程中对软件

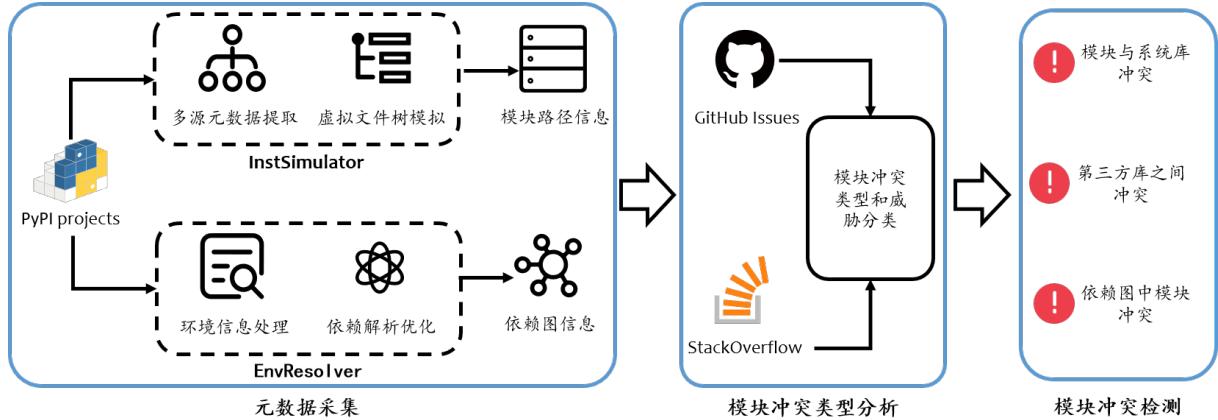


图 3.6 ModuleGuard 框架图

包元数据的访问^[112]。得益于多维度依赖信息提取策略，EnvResolver 能够在解析前获得更为完整的依赖约束信息，包括环境依赖与额外依赖，从而降低解析过程中的不确定性。在解析顺序上，EnvResolver 采用基于优先级的启发式策略。其基本原则是，在保证解析结果不变的前提下，通过合理调整依赖处理顺序以减少回溯开销。具体而言，系统优先解析固定版本依赖，其次解析带有作用域约束的范围版本依赖，最后解析无作用域约束的依赖；同时，距离依赖图根节点较近的依赖项始终优先于远离根节点的依赖项进行解析，这样做能减少回溯深度。通过上述优化策略，EnvResolver 在大规模依赖图解析场景下显著降低了回溯次数，从而有效提升了解析效率。

3.3 ModuleGuard 框架具体实现

本节从系统实现的角度，详细介绍本研究提出的 ModuleGuard 自动化分析与防护框架。如图 3.6 所示，ModuleGuard 的整体工作流由三个相互衔接的核心阶段构成，分别负责元数据采集与解析、模块冲突的实证溯源与威胁建模，以及全生态规模的模块冲突检测与安全治理。

第一阶段为元数据采集与解析引擎构建，其目标是对 Python 生态系统中所有可用软件包及其历史版本的元数据进行系统化提取与规范化处理，从而为后续模块冲突分析提供准确、统一且具备可扩展性的基础数据支撑。在实现层面，ModuleGuard 首先对 PyPI 官方索引中的软件包及其历史版本进行离线抓取，并依据分发格式进行分类存储。针对每一个软件包分发版本，系统分别调用 InstSimulator 与 EnvResolver，对其模块路径布局与依赖关系进行解析。解析结果被统一归一化为结构化的中间表示，包括规范化

的模块路径集合以及引入环境约束后的直接依赖描述，并最终写入本地知识库。

第二阶段为模块冲突模式的实证溯源与威胁建模阶段，旨在对现实世界中模块冲突现象进行细粒度刻画，深入分析其根本成因及潜在影响。本研究系统性地采集了 GitHub 与 StackOverflow 等开源社区中与模块冲突相关的真实 Issue 与讨论记录^[71-72]，并通过多维度的定性分析与归纳，对冲突实例进行抽象与总结。基于上述分析，本研究最终提炼出三类具有代表性的模块冲突模式，并进一步严格界定了其在实际应用场景中可能引发的两类核心安全威胁。

第三阶段为全生态规模的模块冲突检测与安全治理阶段。在该阶段，ModuleGuard 基于前两个阶段构建的解析引擎与威胁模型，对整个 PyPI 生态仓库以及 GitHub 上的热门 AI 项目开展自动化、系统化的模块冲突扫描。通过对检测结果的统计与分析，本研究不仅量化评估了模块冲突风险在软件供应链层面的分布特征与演化趋势，还通过主动向开源社区披露潜在风险并协助开发者修复依赖配置，在 AI 软件应用层面构建了一套覆盖发现、分析与缓解的闭环防御机制。

元数据采集阶段。该阶段构成了 ModuleGuard 框架的基础数据层，其核心目标是在大规模、动态变化的 PyPI 生态环境中，构建一个稳定、可复现且准确的模块与依赖数据库。由于 PyPI 上的软件包持续演进，不同时间点的软件包集合及其依赖关系难以直接复现，且依赖图结构随时间发生变化，本研究采用 bandersnatch 工具对 PyPI 官方仓库进行全量镜像同步^[113]。其在本地构建了约 13TB 规模的 PyPI 镜像。该镜像有效避免了在线访问带来的不稳定因素，为后续大规模离线分析提供了可控且可重复的数据基础。在镜像构建完成后，ModuleGuard 对每一个软件包的历史版本进行系统化解析。具体而言，系统结合 AST 数据流分析与执行流分析技术，从软件包元数据文件中提取与模块路径布局及依赖声明相关的关键信息。随后，利用 InstSimulator 对软件包的安装过程进行语义模拟，通过在虚拟文件系统中重建其安装后的目录结构，推导软件包在真实运行环境中对外暴露的模块路径。最后，系统使用 EnvResolver 对依赖声明进行高效的大规模解析，生成最终的依赖图结构，其中每一个依赖图由结点集合及其之间的依赖边构成。在上述过程中生成的模块路径信息与归一化后的依赖关系被统一组织为结构化中间表示，并存入 PostgreSQL 数据库中。最终，ModuleGuard 构建了一个覆盖 434,823 个软件包、约 420 万个版本的超大规模知识库。该数据库作为框架的核心数据支撑，为后续模块冲突溯源、模式建模以及全生态检测阶段提供了高效、可查询的数据基础。

表 3.2 模块冲突问题的分类及其引发的威胁统计

模块冲突分类	模块覆盖威胁	引入混淆威胁
模块与标准库冲突 (21)	-	✓
模块与第三方包冲突 (64)	✓	✓
依赖图中模块冲突 (12)	✓	-

模块冲突实证溯源阶段。该阶段旨在从真实的软件开发实践中系统挖掘模块冲突的类型及其可能造成的影响机制，为冲突判定规则的构建提供经验依据。本研究采用滚雪球式的搜索策略，在 GitHub 与 StackOverflow 等开源社区中系统搜集与 Python 模块冲突相关的 Issue 与讨论记录。具体而言，本研究组合使用两组关键词进行笛卡尔积组合搜索，其中第一组为 [module, 模块, name, namespace, 名称, 命名空间]，第二组为 [clash, conflict, overwrite, 冲突, 覆盖]。在 GitHub Issue 搜索过程中，进一步限定条件 is:issue and language:python，以确保结果的相关性。由于 GitHub 搜索结果最多仅显示 100 页，本研究共检索并初步收集了约 4,000 条 Issue。随后，通过人工查验与筛选，最终确定其中 55 条 Issue 与模块冲突高度相关。在此基础上，进一步采用滚雪球技术，分析这些 Issue 所引用或被引用的相关讨论，最终在 GitHub 平台上共收集到 78 条与模块冲突强相关的 Issue。对于 StackOverflow 平台，本研究在关键词前引入 Python 这一限定词，并对最受关注的 200 条问题进行人工分析，最终识别出 19 个由模块冲突引发的典型问题。综上，本研究共收集并分析了 97 个与模块冲突密切相关的真实案例。基于上述实证分析结果，本研究将模块冲突形式化划分为三类：模块与标准库之间的冲突、模块与第三方软件包之间的冲突，以及依赖图内部的模块冲突。同时，进一步分析发现，这三类冲突主要导致两种严重威胁，即模块覆盖威胁与引入混淆威胁，其统计结果如表 3.2 所示。

模块冲突检测阶段。在该阶段，ModuleGuard 将前述构建的模块与依赖知识库应用于真实软件生态环境中，开展大规模的模块冲突风险分析。在 PyPI 生态层面，本研究首先系统收集运行环境中的系统标准库模块信息，构建包含 199 个标准库模块全集。随后，基于第一阶段构建的超大规模数据库，对 434,823 个软件包及其约 420 万个版本中声明并实际安装后的模块路径进行枚举与规范化处理，提取所有对外暴露的模块名称集合。在此基础上，ModuleGuard 分别从三个维度对模块冲突进行检测与统计。在此基础上，本研究进一步将分析对象扩展至热门 GitHub 应用，以评估模块冲突在高影响力项目中的实际风险暴露情况。具体而言，本研究以 GitHub 上的热门 AI 项目作为主要分析

对象，根据项目的 star 数筛选出影响力最高的 3,000 个项目。此外，从 awesome-Python 项目中额外收集 1,187 个具有代表性的 Python 项目^[114]。对两个来源的项目进行去重后，最终获得 3,711 个项目，涵盖 93,487 个标签。ModuleGuard 随后对这些项目的依赖配置进行系统化解析，对于每一个目标项目，系统首先基于本地 PyPI 镜像与已构建的依赖知识库，通过 EnvResolver 构建完整的依赖图结构。随后，依据第二阶段形成的冲突判定规则，自动识别潜在的模块冲突风险。在检测过程中，ModuleGuard 同时综合考虑模块覆盖关系与依赖安装导致的模块覆盖，以评估冲突在真实运行环境中的可触发性及其潜在影响范围。

3.4 实验评估

本节对 ModuleGuard 框架的核心组件进行了系统的实验分析与评估。具体而言，本研究构建了两个包含数千个真实 Python 软件包的数据集，并将其通过 pip 原生安装所得的真实结果作为基准，重点从模块提取准确性与依赖图构建准确性两个维度，对 InstSimulator 和 EnvResolution 技术进行了量化评估。

3.4.1 实验设置

基准数据集构建。鉴于 PyPI 生态系统的高动态性，现有工作提供的数据集往往难以实现可重复的精确复现。为此，本研究基于前文构建的 PyPI 全量本地镜像，通过在受控环境中执行实际安装的方式，获取具备高保真度的运行时模块路径及依赖图拓扑，以此作为测试的基准数据集。本研究共构建了以下两个数据集：

- **数据集 1（高频流行项目）：**本研究交叉比对了 Libraries.io 平台中流行度最高的 3,000 个项目，以及 PyPI 官方统计表中 2022 年 8 月至 2023 年 2 月期间下载量排名前 3,000 的项目，经去重处理后构成该流行项目集。
- **数据集 2（全量随机抽样项目）：**从 PyPI 生态的完整项目列表中，采用随机抽样策略选取了 5,000 个项目，作为评估框架泛化能力的基准对照集。

对于上述两个数据集，本研究统一选取各项目的最新版本进行基准测试。在构建基准的实际安装过程中，由于环境异构性（如部分早期包不兼容本实验的 Python 3 环境）以

及安装脚本执行异常等不可控因素，部分软件包无法顺利完成构建。剔除这些无效样本后，数据集 1 与数据集 2 最终分别包含 4,232 个和 3,989 个成功安装的有效软件包。这些有效数据构成了本评估的核心测试集。

评估指标定义。为细粒度量化框架的技术准确度，本研究定义了以下四个维度的评估指标：

- **正确 (Correct):** 框架解析所得的模块列表或依赖图拓扑与 pip 实际安装产生的基准结果严格相同，既无缺失也无冗余。
- **缺失 (Missing):** 框架解析结果未能覆盖基准结果中存在的特定模块路径、依赖节点或依赖边，存在假阴性。
- **多余 (Excess):** 框架解析结果中包含了基准结果中不存在的模块路径、依赖节点或依赖边，存在假阳性。
- **错误 (Incorrect):** 包含路径解析错误、节点命名错误及所有其他不匹配的解析失败情况。

实验环境设置。所有实验均部署于搭载 Ubuntu 22.04 操作系统的服务器上，并统一采用 pip 23.1.2 版本以确保解析逻辑的确定性。在获取模块路径基准时，本实验执行命令 `pip install <package_name> -t <target_dir> --no-dependencies`。该命令不仅将目标包强制隔离安装于指定目录，且禁用了依赖软件包的安装，从而有效消除了依赖项对目标包模块命名空间的干扰，并大幅提升了构建效率。提取路径时，本实验仅过滤并保留扩展名为 .py 的有效 Python 模块文件，自动忽略图像、文本等非代码资源文件，因其不参与 Python import 机制的命名空间解析。在获取依赖图基准时，本实验通过 `pip install -i localhost/simple` 命令将包管理器指向本地克隆的 PyPI 镜像源。为降低大规模全量安装带来的高昂系统开销，本研究通过代码分析，在 pip 核心解析器源码层面注入了钩子函数，并在钩子函数中将解析结果直接写入文件后退出。该机制使得 pip 在完成版本仲裁并生成最终依赖解析树后立即将图拓扑序列化至本地磁盘，并在此处拦截后续的物理下载与解压安装流程，从而在保证基准准确性的同时实现了轻量级的数据采集。

表 3.3 模块和依赖解析技术评估实验结果

技术名称	测试集	正确	缺失	多余	错误	准确率
InstSimulator	数据集 1	4,045	152	28	7	95.58%
	数据集 2	3,834	116	37	2	96.11%
EnvResolution (节点维度)	数据集 1	4,177	41	8	13	98.70%
	数据集 2	3,795	93	30	20	96.37%
EnvResolution (图维度)	数据集 1	4,133	46	11	47	97.66%
	数据集 2	3,748	107	40	33	95.18%

- * 节点维度: 仅评估依赖图拓扑中节点集合的完整性与一致性。
- * 图维度: 严格评估依赖图拓扑中节点集合与有向边集合(依赖关系)的一致性。
- * 准确率: 正确解析的样本数占所在测试集总有效样本数的百分比。

3.4.2 评估结果与分析

InstSimulator 模块提取准确性。如表 3.3 所示, InstSimulator 在流行项目(数据集 1)和随机项目(数据集 2)中的模块提取准确率分别达到了 95.58% 和 96.11%, 这证明了其基于虚拟文件树和语义模拟的无安装提取技术具备极高的可靠性。针对导致精度损失的失败案例, 本研究进行了深度根因分析。结果显示, InstSimulator 在两个数据集中分别出现了 152 例(3.59%) 和 116 例(2.91%) 的“缺失”判定。导致这一现象的核心瓶颈在于: InstSimulator 采用基于抽象语法树的静态数据流分析来提取 setup.py 中的配置参数, 该方法难以处理包含高度动态特性或复杂控制流的安装脚本。例如, 部分恶意或不规范的构建脚本通过动态字符串拼接, 或在运行时读取外部自定义非标准配置文件的方式来构造模块列表, 这超出了传统静态数据流分析的能力边界。此外, 多分发包的不确定性亦是导致误差的原因。一个特定版本的 Python 项目在 PyPI 仓库中可能同时存在多个结构迥异的分发包, 如针对不同平台编译的 .whl。由于 InstSimulator 依据内部启发式规则仅选取其中一个工件进行虚拟文件树推演, 其选择可能与基准测试中 pip 基于宿主环境上下文动态选取的物理包不一致。例如, jaxlib 项目的 0.4.4 版本包含 12 个不同的分发工件, 其内部的底层模块路径存在平台级差异。总体而言, InstSimulator 实现了超越 95% 的模块高保真提取率, 完全满足了对百万级生态系统进行全量安全审计的需求。

EnvResolution 依赖解析准确性。为全面评估 EnvResolution 技术的鲁棒性, 本实验

从节点级和图级两个维度对其进行了对比测试。节点级评估侧重于验证直接依赖与传递性依赖集合的完备性（即依赖包的存在性），而图级评估则实行更严格的标准，要求依赖图有向边的约束逻辑完全同构。表 3.3 的统计数据表明，EnvResolution 的解析准确率在两个维度上均稳定维持在 95.18% 至 98.70% 的高水平区间；受限于传递路径的复杂性，图级准确率符合预期地略微低于节点级。通过对解析失败的案例进行人工审查，本研究确认了两个主要的误差来源。首先，规避高开销的技术设计折衷带来了局部解析盲区。为了保证生态系统级别的解析速度，EnvResolution 被设计为仅从压缩包内部提取特定标准配置文件（如 `setup.cfg`, `requirements.txt`）进行分析，而不执行全量解压。然而，少数非标准化项目并不直接定义参数列表，例如 `ta` 包的 `setup.py` 会强行要求读取包内部深层目录下的自定义依赖文件，这导致 EnvResolution 因没有全量解压而无法捕获该非标准配置，从而引发解析失败。其次，跨环境的元数据不同也是导致不一致的重要原因。部分依赖元数据是在项目维护者的异构本地环境中静态生成的。例如，`fortnitepy` 项目的元数据声明中包含三个直接依赖，但在本实验的 Linux 测试环境中，pip 原生解析器通过动态评估环境标记最终仅认定仅有其中两个依赖。总体而言，上述实验数据有力地证明了，尽管 EnvResolution 在应对高度非标准化的动态脚本与部分环境异构导致的软件包不同时存在一定局限性，但其在极大程度降低解析时间开销的同时，成功保证了面向大规模供应链分析的极高准确率。

3.5 实证结果分析

鉴于模块冲突问题在现有研究中尚未得到系统性刻画与深入分析，本小节基于对 GitHub 与 StackOverflow 上相关 Issue 的大规模收集与人工实证分析，总结并形式化定义了三类典型的模块冲突类型，并进一步归纳其可能引发的两类核心安全威胁。在此基础上，本研究对从 GitHub 收集的 420 万个软件包版本以及 3,711 个高星开源项目进行了定量评估，以系统分析模块冲突在真实生态中的存在性及其潜在影响范围，从而做到对软件应用层的防护作用。

3.5.1 模块冲突定义

通过对真实开发者反馈的归纳分析，本研究发现，在软件包发布至 PyPI 以及用户通过 pip 进行安装与依赖解析的过程中，主要存在三类模块冲突现象：模块与标准库之间的冲突、模块与第三方库之间的冲突，以及依赖图内部的软件包之间的模块冲突。为精确刻画上述冲突类型之间的差异，本文首先给出形式化符号定义如下：

P ：表示 PyPI 上所有软件包的集合

$p : p \in P$, 表示 PyPI 中某一具体软件包

$M_p : \{m \mid m \text{ 为软件包 } p \text{ 安装后生成的模块}\}$

$M_{Lib} : \{m_l \mid m_l \text{ 为 Python 标准库模块}\}$

$DG(p)$ ：表示软件包 p 的依赖图（Dependency Graph）

$m_1 \xleftarrow{\text{conflict}} m_2 : m_1 \text{ 与 } m_2 \text{ 具有相同模块名称或相同模块路径}$

模块与标准库之间冲突。模块与标准库之间冲突（Module-to-Lib）是指某一软件包在安装后生成的模块与 Python 标准库中的模块在名称上发生冲突。具体而言，若存在软件包 $p \in P$ 及其模块 $m \in M_p$ ，同时存在标准库模块 $m_l \in M_{Lib}$ ，且二者满足 $m \xleftarrow{\text{conflict}} m_l$ ，则认为该软件包存在模块与标准库之间的冲突。其形式化定义如式 3-1 所示。例如，python-hgjson 在 Issue #14 中报告，其 v1.5.0 版本包含名为 json 的模块，该模块与标准库中的 json 模块发生冲突，导致用户在导入时无法正常使用软件包功能^[115]。该案例表明，即便标准库模块无法被覆盖，其命名冲突仍可能导致解释器加载路径发生歧义。

$$\exists p \in P, m \in M_p \wedge m_l \in M_{Lib} \wedge m \xleftarrow{\text{conflict}} m_l \quad (3-1)$$

模块与第三方库之间冲突。模块与第三方库之间冲突（Module-to-TPL）指两个不同软件包之间存在同名或同路径模块冲突。形式化而言，若存在 $p, p' \in P$ 且 $p \neq p'$ ，并分别存在 $m \in M_p$ 与 $m' \in M_{p'}$ 满足 $m \xleftarrow{\text{conflict}} m'$ ，则认为软件包 p 与 p' 之间存在模块冲突。其定义如式 3-2 所示。例如，在 Cookiecutter-udata-plugin 项目的 Issue #3 中指出，python-slugify (v8.0.0) 与 awesome-slugify (v1.6.5) 均包含名为 slugify.py 的模块。当二者被安装于同一运行环境时，将发生冲突并导致其中一个软件包功能异常^[116]。

$$\exists p, p' \in P, p \neq p' \wedge m \in M_p \wedge m' \in M_{p'} \wedge m \xleftarrow{\text{conflict}} m' \quad (3-2)$$

依赖图内部软件包之间的模块冲突。依赖图内部软件包之间的模块冲突（Module-in-Dep）是指在某一根软件包 r 的依赖图 $DG(r)$ 中，存在两个不同依赖包 p 与 p' ，且其模块 $m \in M_p$ 与 $m' \in M_{p'}$ 之间发生冲突。此时，即便根软件包本身未直接声明冲突模块，其整体依赖结构仍可能引发冲突风险。其形式化定义如式 3-3 所示。例如，在项目 emoca 的 Issue #44 中指出，其 v1.0 版本的依赖图同时包含 opencv-python (v4.5.5) 与 opencv-python-headless (v4.5.5)。由于二者均包含 cv2 模块，导致依赖图内部产生模块冲突，从而影响根项目的正常运行^[117]。该类冲突具有较强隐蔽性，尤其在间接依赖情形下，开发者往往难以感知其存在。

$$\exists(p, p') \subseteq DG(r), p \neq p', m \in M_p \wedge m' \in M_{p'} \wedge m \xleftarrow{\text{conflict}} m' \quad (3-3)$$

3.5.2 模块冲突影响与新型攻击面

基于对 97 个真实 Issue 的系统化实证分析，本小节进一步归纳 Python 生态在代码管理机制层面存在的两项结构性缺陷。在此基础上，本文提出一种面向软件应用层的潜在攻击范式——模块替换攻击。该攻击并非依赖传统漏洞利用，而是借助模块命名冲突与加载优先级机制，在合法安装流程中实现模块替换，从而影响程序语义与执行结果。

首先，Python 采用统一的模块导入语法 `import`，允许从标准库、第三方库以及项目自身目录中加载模块。然而，解释器在解析导入请求时遵循首次匹配即停止策略，即先次查询 `sys.modules` 中的缓存记录，再按照 `sys.path` 中记录的模块路径顺序，一项项查找，而未对模块来源进行显式区分。该机制导致当不同位置存在同名或同路径模块时，解释器可能加载与开发者预期不一致的模块，从而引发导入混淆威胁。其次，`pip` 默认将来自 PyPI 的第三方库统一安装至 `site-packages` 目录，且不同软件包之间缺乏物理隔离机制。虽然虚拟环境可以隔离不同项目之间的依赖版本，但在单一项目内部，直接依赖与间接依赖仍被混合存储于同一目录层级。当存在同名或同路径模块时，后安装模块可能覆盖先前模块，从而引发模块覆盖威胁。上述两类机制性缺陷分别作用于软件包的安装、升级与运行阶段，对项目依赖完整性与运行可靠性构成严重威胁。下文结合典型案例，对两类威胁进行详细分析。

威胁一：模块覆盖。模块覆盖威胁源于 Python 生态缺乏包级物理隔离机制。在默认安装模式下，不同软件包的模块被写入同一 `site-packages` 目录。当两个原本无直接关

联的软件包包含相同相对路径的模块时，后安装模块将覆盖先安装模块，从而破坏项目的依赖完整性与功能一致性。该威胁主要由模块与第三方库之间冲突以及依赖图内部模块冲突触发。

在模块与第三方库之间冲突场景下，`pip` 并不会对不同软件包进行命名空间级别隔离。例如，在 `pypa/pip` 项目的 Issue #4625 中，开发者同时安装 `pyjwt`(v1.5) 与 `jwt`(v0.5.2)，二者在 `jwt/exceptions.py` 模块上发生冲突。由于安装顺序差异，后安装的软件包覆盖先前模块，导致运行时行为异常^[118]。该案例表明，即便两个软件包功能不同，只要存在同路径模块，其并存即可能破坏项目运行环境。在大小写不敏感（Case-Insensitive）的文件系统环境（如 Windows）中，该问题进一步加剧。文件系统对目录名不区分大小写，导致不同命名的模块可能被解析为同一物理路径。例如，在 `pycrypto` 项目的 Issue #156 中，开发者先安装 `crypto`（小写），随后安装包含 `Crypto`（大写）目录的 `pycrypto`。由于文件系统无法区分大小写，`pip` 无法创建新的 `Crypto` 目录，而是将模块写入既有的 `crypto` 目录中，不仅覆盖原有模块，而且改变模块层级结构，导致所有 `import Crypto` 语句失效，最终破坏项目功能^[119]。该现象表明，模块冲突与底层文件系统特性存在一定关联，从而扩大了威胁影响范围。

在依赖图内部冲突情形下，模块覆盖通常发生于直接依赖与间接依赖之间。例如，在 `Albumentations` 项目的 Issue #841 中，项目同时依赖 `opencv-python`（间接依赖）与 `opencv-python-headless`（直接依赖），二者在 `cv2` 模块上存在冲突^[120]。尽管官方文档已明确指出两者不可同时安装，但由于间接依赖对开发者而言缺乏可见性，其安装过程具有“黑盒”特征，冲突仍可能在自动依赖解析过程中被触发^[121]。此外，模块覆盖亦可能在软件升级阶段产生。例如，在 Issue #8509 中，`ansible@2.9.10` 升级后出现文件缺失问题。其根本原因在于 `pip` 的升级流程包括：首先安装新版本依赖（期间可能发生模块覆盖），随后卸载旧版本（可能误删已被覆盖的新模块，因为系统将其识别为旧版本文件），最后安装新版本主包。该过程破坏了文件一致性，导致软件包完整性受损^[122]。该案例说明，模块覆盖不仅影响安装阶段，还可能在版本更新过程中存在风险。

威胁二：导入混淆。导入混淆威胁源于 Python 导入解析机制未区分模块来源，而是基于优先级顺序进行匹配。当存在同名模块时，解释器可能加载与开发者预期不一致的模块，从而改变程序语义。

在模块与标准库冲突场景下，标准库模块虽然物理上独立存储且无法被覆盖，但若

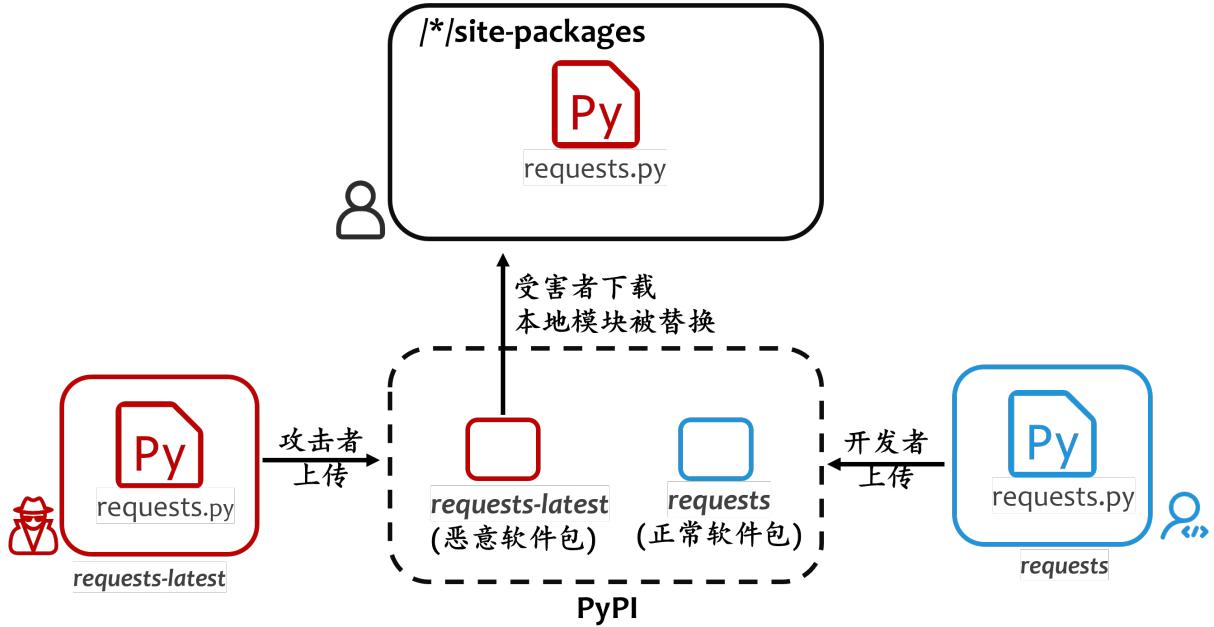


图 3.7 模块替换攻击原理图

第三方库或项目自身模块与标准库同名，解释器可能优先加载缓存或系统路径中的标准库模块。例如，项目 FibexConverter 包含名为 parser.py 的模块，但运行时却被解析为标准库中的 parser 模块^[123]。这是由于 Python 启动时已将部分标准库模块缓存于 sys.modules 全局变量中，当执行 import parser 时，解释器首先查询缓存并直接返回已加载模块，从而导致项目模块未被正确导入。进一步地，当用户首先加载与标准库同名的第三方模块时，该模块将被写入 sys.modules 缓存。此后，即使开发者希望调用标准库模块，解释器仍可能返回已缓存的第三方模块，从而导致内置功能异常，严重情况下可能影响 Python 解释器的正常执行流程。

在模块与第三方库冲突情形下，解释器按照 sys.path 中路径顺序进行搜索，并在首次匹配后停止。当冲突模块位于不同目录时，实际加载结果依赖路径排列顺序。这要求开发者精确控制运行环境配置，否则极易产生不可预测行为。尽管命名空间包机制在一定程度上缓解了模块覆盖问题，但其并未解决由加载优先级带来的导入歧义风险。此外，Python 支持多种安装方式，不同工具具有不同默认路径。例如，通过 apt-get 安装的模块位于 /usr/ 目录，通过 pip 安装的模块位于 site-packages/，而 conda 与 poetry 等工具则维护独立环境路径。多路径共存的生态结构显著增加了模块搜索空间的复杂性，使得导入优先级问题更加难以预测，从而扩大了导入混淆攻击面的潜在影响范围。

模块替换攻击。如图 3.7 所示，模块替换攻击的核心原理在于利用 pip 在安装软件

包时对同名模块的覆盖行为。当目标环境中已存在某一模块，而新安装的软件包中包含相同模块路径的文件时，后安装模块将在默认配置下直接覆盖原有模块，且整个过程对用户缺乏显式提示与风险告警。在此机制下，攻击者无需利用传统漏洞，仅通过构造特定命名与依赖关系，即可实现对既有模块的语义替换。具体而言，攻击者可以构造一个包含高频模块名称的恶意软件包并上传至 PyPI。例如，在恶意软件包中包含名为 `requests.py` 的模块，并在其中嵌入经过混淆处理的恶意代码。当用户在本地环境中安装该软件包后，若其模块路径与现有合法模块发生冲突，则原有模块可能被替换。攻击者可通过多种方式诱导用户下载该恶意软件包。

首先，攻击者可以采用名称混淆策略，例如使用 `requests-latest` 等具有误导性的名称，伪装为 `requests` 软件包的最新更新版本。其次，可以利用拼写近似攻击，上传常用软件包相似名字，在用户拼写错误软件包名称时触发误下载。再次，攻击者可以利用 Python 报错提示机制。当解释器在缺失模块时报出 `ModuleNotFoundError: No module named xxx` 错误时，部分开发者可能误将缺失模块名视为软件包名称并直接执行 `pip install xxx`。例如，实际所需软件包为 `opencv-python`，但报错信息显示缺少 `cv2` 模块，攻击者若提前发布名为 `cv2` 的恶意软件包，即可能诱导用户执行 `pip install cv2` 并下载恶意代码。此外，攻击者亦可发布表面功能正常的软件包，但在其依赖配置文件（如 `requirements.txt` 或 `setup.py`）中引入恶意依赖项。由于 `pip` 在安装软件包时会自动解析并下载其依赖，该机制可被滥用于间接分发恶意模块。在上述任一情形下，一旦恶意软件包被安装，其模块文件将写入本地 `site-packages` 目录，并可能覆盖原有良性模块。

当用户后续通过 `import` 语句加载相关模块时，解释器将优先执行已被替换的恶意代码，从而在用户无感知的情况下触发攻击行为。该类攻击具有隐蔽性强、触发条件自然、执行路径合法等特征，难以通过传统漏洞扫描或异常检测机制发现。同时，当前 PyPI 生态在恶意软件包检测与主动下架机制方面仍存在局限，使得此类攻击在现实供应链环境中具有可行性与持续性风险。

3.5.3 PyPI 生态大规模检测结果

本小节基于 ModuleGuard 框架，对 PyPI 生态系统中的模块冲突问题开展了大规模实证检测分析。研究对象覆盖截至 2023 年 3 月发布的 420 余万个 PyPI 软件包版本，系统性评估了三类模块冲突模式在真实生态中的分布特征与演化趋势。

表 3.4 软件包中冲突频率最高的前 10 个模块路径

模块路径	最新版本包数量	所有版本包数量
src/__init__.py	1,157	8,777
__init__.py	1,083	4,421
utils/__init__.py	410	3,899
distributions/__init__.py	404	448
distributions/Generaldistribution.py	394	431
distributions/Gaussiandistribution.py	394	431
distributions/Binomialdistribution.py	393	428
client/__init__.py	367	1,142
scripts/__init__.py	363	5,336
server/__init__.py	360	796

针对模块与第三方库冲突，考虑到同一项目的不同版本无法在同一运行环境中并存，且在未显式指定版本约束的情况下 pip 默认安装最新版本，本研究仅提取每个项目截至 2023 年 3 月的最新版本进行分析。在识别出存在冲突的软件包后，假设这些软件包在实际应用场景中可能被同时安装于同一环境，从而评估其潜在冲突风险。针对模块与标准库冲突，本研究首先依据 Python 官方文档收集了 199 个标准库模块名称，并据此对全生态软件包的模块命名进行匹配分析^[124]。鉴于真实用户环境中 sys.path 的优先级配置及标准库加载状态具有不确定性，本研究采用保守假设：即 199 个标准库模块在用户环境中均可用，且已被加载至解释器缓存（sys.modules）中，从而评估最坏情况下的导入混淆风险。针对依赖图内部冲突，本研究对每个项目的所有历史版本进行全面解析，并利用 Envresolver 技术构建其完整依赖图。在获得传递性依赖关系后，逐一检查依赖节点对应的软件包文件结构，判断是否存在同路径模块冲突或内容不一致的同名模块。

总的来说，本研究分析数量级为从 PyPI 提取 4,223,950 个软件包版本，解析得到 177,216,363 条模块路径及 27,678,668 条直接依赖声明，并成功构建 4,223,950 个完整依赖图。其中，处于最新版本的软件包数量为 424,823 个，共涉及 5,419,306 条模块路径。

模块与第三方库冲突分析。针对 424,408 个最新版本项目的量化评估结果表明，共有 91,134 个软件包（占比 21.45%）存在模块与第三方库冲突，涉及 386,595 条模块路径（占总模块数 5,419,306 的 7.13%）。根据这些软件包是否被安装至相同目标路径，其

冲突可能演化为模块覆盖或导入混淆威胁。进一步分析发现，在 27,851 个软件包（占比 6.56%）中，冲突模块位于大小写不敏感的文件系统环境下可能被解析为相同路径，共牵涉 3,517 条模块路径。这表明底层文件系统特性会进一步放大模块覆盖风险。

研究发现，开发者在发布软件包时，往往将运行时非必需的冗余模块一并打包。例如，包含测试模块（如 `test(s)/__init__.py`）的软件包多达 41,095 个，包含示例模块（如 `example(s)/__init__.py`）的软件包达 14,877 个。这些模块通常仅用于本地开发或调试阶段，将其分发至生产环境不仅增加命名空间污染风险并诱发导入错误，还额外占用仓库存储资源。更为重要的是，在依赖解析阶段，这些冗余元数据会显著扩大回溯算法在回溯时下载软件包的大小，从而增加 `pip` 在复杂依赖场景下的解析时间开销，降低整体安装效率。

如表 3.4 所示，本研究识别出生态中冲突频率最高的前 10 个模块路径。其中，超过 1,000 个最新版本软件包包含缺乏明确功能语义的 `src/__init__.py` 与 `__init__.py` 文件。该现象主要源于开发者对 `src-layout` 与 `flat-layout` 目录结构配置不当，导致初始化文件被错误地置于项目根目录并随包一并分发^[125]。

此外，本研究观察到，存在模块冲突的软件包在命名上往往具有较高语义相似性。例如，在包含 `distributions/__init__.py` 模块的 404 个软件包中，有 290 个项目名称包含 `distribution` 子串。这表明功能相近或处于同一领域功能的软件包更有可能被终端用户组合使用，从而显著提高实际环境中冲突触发的概率。通过对相关 GitHub Issue 的进一步挖掘，本研究发现，即便维护者意识到其模块命名与其他流行库存在冲突，通常也不愿主动重命名。原因在于核心模块重命名将破坏向后兼容性，引发大规模重构，并增加用户开发成本和下游用户的使用更新成本。

模块与标准库冲突分析。在对 420 万个软件包的全量扫描中，共识别出 345,068 个软件包（占比 8.17%）存在模块与标准库冲突。在 199 个标准库模块中，有 182 个（占比 91.96%）至少与一个第三方软件包发生重名。冲突频率最高的三个标准库模块分别为 `types`、`io` 与 `logging`，分别涉及 69,940、47,214 和 35,694 个第三方软件包。

研究揭示，本地开发环境与实际部署环境之间的路径优先级差异是引发此类导入混淆的重要因素。在本地调试阶段，当前工作目录通常在 `sys.path` 中具有较高优先级；而当项目被打包并通过 `pip` 安装后，其模块所在的 `site-packages` 目录优先级低于系统标准库。这种优先级反转可能导致在开发环境中开发者加载的是本地开发目录下的模块，而

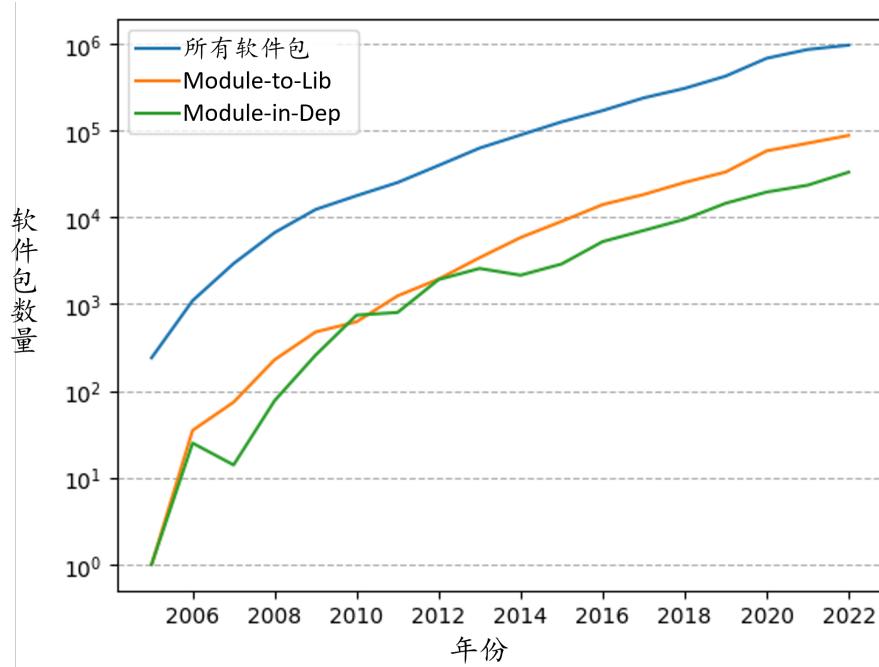


图 3.8 历年发布的软件包总数与存在模块冲突的软件包数量统计

在用户部署环境中，其加载的是标准库模块，而非预期的第三方模块，从而引发运行时异常。为解决该问题，开发者通常采取模块重命名或使用相对导入等方式。然而，这些方案可能破坏向后兼容性，并降低代码的可读性与跨平台移植性。

如图 3.8 所示是历年发布的软件包总数与存在模块冲突的软件包数量统计图，其结果表明模块与标准库冲突的软件包数量呈逐年增长趋势。随着 Python 生态规模的持续扩张及标准库功能的不断丰富，该类冲突已演化为一种长期存在且不断累积的供应链风险。

表 3.5 向 GitHub 开源社区报告的模块冲突漏洞工单及修复状态

开源项目	工单号	已回复	已修复
wzpan/wukong-robot	#286	✓	-
RVC-Project/Retrieval-based-Voice-Conversion-WebUI	#908	-	✓
google/BIG-bench	#946	-	-
google-research/text-to-text-transfer-transformer	#1106	-	-
asymly/texar	#303	-	-
invoke-ai/InvokeAI	#4078	✓	✓
Picsart-AI-Research/Text2Video-Zero	#63	-	-
sail-sg/EditAnything	#51	✓	✓
AIGC-Audio/AudioGPT	#89	-	-
Sygil-Dev/sygil-webui	#1808	-	-
riffusion/riffusion	#151	✓	✓
Anjok07/ultimatevocalremovergui	#702	✓	✓
pjialin/py12306	#454	-	-
PaddlePaddle/PaddleSpeech	#3432	✓	✓
marqo-ai/marqo	#556	✓	✓
nbei/Deep-Flow-Guided-Video-Inpainting	#113	-	-
saleor/saleor	#13554	-	✓
jantic/DeOldify	#482	✓	✓
PaddlePaddle/PaddleRec	#940	-	-
kohya-ss/sd-scripts	#684	✓	✓
bmaltais/kohya_ss	#1267	✓	✓
sensity-ai/dot	#98	✓	✓
chatchat-space/Langchain-Chatchat	#952	✓	✓
PaddlePaddle/PaddleOCR	#10490	-	-
microsoft/TaskMatrix	#434	-	-
weecology/DeepForest	#464	✓	✓
OpenGVLab/InternGPT	#53	-	-
Rudrabha/Wav2Lip	#536	-	-
lucasjinreal/weibo_terminator	#62	-	-
airbnb/streamalert	#1345	-	-
jupyter/jupyter_core	#350	✓	✓
microsoft/TaskMatrix	#345	-	-
qpzzk/zz_spider	#1	-	-
NickHugi/PyKotor	#3	✓	✓

依赖图内部冲突分析。对 420 万个软件包的依赖图拓扑分析显示，共有 129,840 个软件包（占比 3.07%）存在依赖图内部冲突，涉及 11,666 个项目。其中，38,371 个软件

包（关联 4,516 个项目）在依赖树中存在路径完全相同但文件内容哈希值不同的冲突模块。这种情况在 pip 安装软件包时会自定触发覆盖行为引发功能异常。部分冲突模块会随着上游依赖版本迭代而发生内容变更，从而在下游系统中形成潜在功能退化风险。即便某些冲突模块在当前主执行路径中未被直接调用，其静态覆盖行为已破坏软件包的物理完整性。在复杂 AI 系统中，无法保证这些模块不会在未来版本或异常处理分支中被触发。

对冲突项目供应链关系的深入分析表明，依赖图内部冲突主要源于三类根因：其一，不同维护者实现相似功能的竞争性软件包采用相同或相近模块命名，在同时作为间接依赖引入时产生命名空间重叠，例如，在 saleor 的依赖图中，python-magic-bin 是 python-magic 的一个由不同维护者维护的分支；其二，生态演进过程中的包名迁移，即旧项目被弃用后由新名称的分支替代，在过渡期被不同组件分别依赖，例如在 riffusion 项目中，soundfile 是 pysoundfile 的演进重命名版本；其三，同一项目的不同变种或面向不同环境的分发版本被同时引入依赖树，例如 opencv-python 与 opencv-python-headless 则是 opencv 分别面向桌面 GUI 与服务器环境的两个互相冲突的平行分发版。

此外，冲突在项目生命周期中表现出显著的潜伏性。在 4,516 个明确存在内容冲突的项目中，冲突集中分布于早期连续版本区间（2,342 个项目）或贯穿全部历史版本（1,819 个项目）。统计结果显示，此类未决冲突平均跨越 6.5 个版本发布周期。该现象表明，维护者往往仅在出现显性功能故障后才进行修复，而大量冲突在未触发异常的情况下长期潜伏，削弱系统整体可靠性。例如，aniposelib 项目在 0.3.7 版本之前长期处于 opencv-python 与 opencv-contrib-python 的依赖冲突状态中，直至终端用户在 Issue 中报告了由依赖图内部冲突引发的严重运行时错误后，维护者才在后续版本中予以纠正^[126]。如图 3.8 从时间维度观察，依赖图内部冲突的受影响项目数量亦呈增长趋势。部分早期发布时依赖结构健康的软件包，在上游依赖迁移、分裂或废弃后，重新解析时触发新的依赖图中模块冲突问题，这一现象凸显了软件供应链依赖关系的长期脆弱性与动态退化特征。

3.5.3.1 GitHub 热门项目检测结果

本研究选取 GitHub 平台中具有较高影响力的开源 Python 项目开展系统性实证分析，检测其中潜伏的依赖图内部冲突。实验结果显示，共有 519 个项目（占比 13.93%）

的 10,850 个版本标签（占比 11.61%）存在模块覆盖风险。相较于 PyPI 生态的宏观统计结果，依赖图内部冲突在 GitHub 实际工程项目中的发生比例更高。其原因在于，面向真实应用场景的项目通常构建了更为复杂且规模更大的依赖拓扑结构，因而更易产生传递性的模块命名空间重叠。即便覆盖前后模块文件的内容哈希值一致，在语义层面未直接引发显性运行时异常，这类静默覆盖行为仍会破坏本地环境中软件包的物理完整性，并显著增加环境异常的定位与调试难度，因为并没法保证后续该文件是否会在某一个版本进行修改，从而诱发潜在威胁。

进一步分析表明，在 108 个项目的 2,569 个版本标签中，覆盖前后模块文件存在实质性内容差异，从而带来直接的功能性错误风险。在这 108 个受影响项目中，有 65 个项目的最新版本仍遗留该隐患，其余 43 个项目则在后续版本迭代中完成了修复。统计结果显示，每个未修复冲突平均跨越 23 个历史版本，反映出该类问题具有显著的潜伏期与隐蔽性。通常仅当终端用户在特定执行路径下触发显性崩溃并提交错误报告后，维护者才会意识到冲突的存在并采取修复措施。针对上述 65 个包含未决冲突的最新版本项目，本研究开展了细粒度人工代码审计，并向相关维护者提交了 35 个安全漏洞工单（因为部分项目的冲突原因相同，因此本研究向依赖的维护者提交工单而非项目本身维护者）。如表 3.5 所示，截至目前，已有 15 个项目团队作出正式回应，其中 16 个项目已得到了修复以消除模块冲突（部分项目静默修复了问题，并未回复工单），还有部分项目正在持续跟进中。对于尚未回应的项目，鉴于其冲突触发机制与已确认问题在结构上完全同构，本研究合理推断，这些遗留冲突同样可能对受限系统环境造成实质性的功能破坏。

实证分析显示，模块冲突问题在 AI 相关项目中呈现出显著高发态势。其典型诱因源于底层计算机视觉库的交叉依赖。开发者在构建 AI 系统时，通常显式声明依赖四种 opencv-python 变体发行版中的某一版本，同时还会引入其他 AI 相关第三方组件。然而，这些组件在其传递性依赖树中往往再次声明其他互斥版本的 opencv 基础库，例如 opencv-python-headless。尽管官方文档已明确指出这些变体共享同一 cv2 模块命名空间，无法在同一运行环境中共存，但此类冲突在实际工程中仍广泛存在^[121]。其根本原因在于，应用层开发者通常只能显式约束直接依赖，而对传递性依赖缺乏可见性与控制能力。由此产生的底层命名空间重叠，直接导致多个 AI 衍生项目在同一生产环境集成部署时出现严重的不兼容与环境破坏问题。

研究进一步发现，部分开发者在顶层依赖中同时引入功能重叠且存在模块级冲突的软件包。通过与开源社区维护者的沟通，本研究观察到一种风险较高的开发模式，即当开发者遭遇由底层模块覆盖引发的异常行为时，往往通过增加新的依赖项以期“覆盖”原有环境或缓解错误，而非从依赖拓扑结构层面定位冲突根因。这种做法表明，部分开发实践过度关注程序是否能够暂时运行，而忽视了依赖图结构的健康性与文件系统层面的完整性约束。引入功能冗余依赖不仅显著提升工程复杂度，也增加了下游用户在构建与部署环境时的失败概率。大量项目 Issue 讨论记录进一步表明，开发者普遍缺乏对“模块冲突”这一系统级架构缺陷的整体认知。多数修复行为依赖反复试错机制，通过增删依赖配置维持程序的脆弱可运行状态。实证数据亦支持该观察：在本研究追踪的 16 个已确认修复的最新版本项目中，有 12 个项目的最终修复方案仅为移除冗余依赖声明。综上所述，本研究的量化分析揭示了模块冲突在软件供应链内部的传播机制与潜在破坏路径，并为开发者在开发与调试阶段规范依赖声明、识别环境风险及提升供应链稳健性提供了实证基础。

3.6 本章小节

针对 AI 系统软件应用层日益严峻的模块冲突威胁，本章提出并设计了一套名为 ModuleGuard 的自动化分析与检测框架。具体而言，该框架的整体工作流主要包含三个核心阶段：

在第一阶段为元数据采集阶段，其为突破全生态规模下模块与依赖分析中存在的效率瓶颈与准确率偏低的难题，本章创新性地提出了 InstSimulator 与 EnvResolver 两项关键技术。这两项技术能够在不执行实际物理安装的前提下，高效、高保真地提取整个 PyPI 生态系统的模块拓扑路径与传递性依赖图信息，进而构建了大规模的本地依赖知识库。基于真实 pip 安装结果构建的基准测试集评估表明，上述两项技术在节点级与图级等多个验证维度下均实现了 95% 以上的解析准确率。

在第二阶段为模块冲突模式的实证溯源阶段，本研究通过对 GitHub 和 StackOverflow 开源社区中的真实 Issue 进行系统性的实证挖掘与溯源分析，科学归纳了三类典型的模块冲突模式，并严格界定了其引发的两类核心安全威胁。基于底层包管理器缺乏物理隔离以及导入机制缺乏上下文校验的结构性缺陷，本章首次提出并定义了一种针对

AI 系统软件应用层的新型软件供应链攻击向量——模块替换攻击。

在第三阶段为模块冲突检测阶段，为了有效防御模块替换攻击并缓解模块冲突带来的潜在系统风险，本研究依托前期构建的本地知识库，利用 ModuleGuard 框架对整个 PyPI 仓库及 GitHub 上的热门 AI 开源项目进行了全量自动化扫描检测。在挖掘出大量真实世界中隐蔽的模块冲突缺陷后，本研究积极向相关项目维护者披露漏洞工单，并协助其完成依赖配置的修复，从而在 AI 系统软件应用层成功建立了一套闭环的供应链安全防御与生态治理机制。

4 针对模型框架层的新型恶意模型攻击与检测框架

4.1 引言

参考文献

- [1] 比亚迪. 比亚迪获全国首张 L3 自动驾驶高快速路测试牌照, 全面加速智能化布局[EB/OL]. 2023. <https://www.byd.com/cn/news/2023/detail496>.
- [2] Apple Inc. Siri[EB/OL]. 2025. <https://www.apple.com/siri/>.
- [3] Huawei. 小艺助手[EB/OL]. 2025. <https://xiaoyi.huawei.com/chat/>.
- [4] OpenAI. ChatGPT[EB/OL]. 2022. <https://openai.com/zh-Hans-CN/index/chatgpt/>.
- [5] ABADI M, BARHAM P, CHEN J, et al. {TensorFlow}: a system for {Large-Scale} machine learning [C]//12th USENIX symposium on operating systems design and implementation (OSDI 16). 2016: 265-283.
- [6] PASZKE A, GROSS S, MASSA F, et al. Pytorch: An imperative style, high-performance deep learning library[J]. Advances in neural information processing systems, 2019, 32.
- [7] BRADSKI G, the OpenCV team. opencv-python: OpenCV library for Python[EB/OL]. 2025. <https://pypi.org/project/opencv-python/>.
- [8] INC. H F. Hugging Face Hub: A Platform for Sharing Machine Learning Models, Datasets and Demos[EB]. 2026.
- [9] Various Contributors. Model Zoo: A Collection of Pre-trained Deep Learning Models[EB]. 2026.
- [10] Google Research. TensorFlow Hub: A Repository of Trained Machine Learning Models[EB]. 2026.
- [11] Sonatype. State of the 2021 Software Supply Chain[J/OL]. Sonatype Blog, 2021. <https://www.sonatype.com/blog/software-supply-chain-2021>.
- [12] AKHOUNDALI J, NOURI S R, RIETVELD K, et al. MoreFixes: A large-scale dataset of CVE fix commits mined through enhanced repository discovery[C]//Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering. 2024: 42-51.
- [13] CHENG W, ZHU X, HU W. Conflict-Aware Inference of Python Compatible Runtime Environments with Domain Knowledge Graph[C/OL]//ICSE '22: Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022: 451-461. <https://doi.org/10.1145/3510003.3510078>. DOI: 10.1145/3510003.3510078.
- [14] MUKHERJEE S, ALMANZA A, RUBIO-GONZÁLEZ C. Fixing dependency errors for Python build reproducibility[C]//Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis. 2021: 439-451.
- [15] SMITH J. pipreq[EB/OL]. 2023. <https://github.com/bndr/pipreqs/>.
- [16] YE H, CHEN W, DOU W, et al. Knowledge-based environment dependency inference for Python programs[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1245-1256.
- [17] MAHON J, HOU C, YAO Z. PyPitfall: Dependency Chaos and Software Supply Chain Vulnerabilities in Python[J]. arXiv preprint arXiv:2507.18075, 2025.
- [18] ALFADEL M, COSTA D E, SHIHAB E. Empirical analysis of security vulnerabilities in Python packages[J/OL]. Empirical Softw. Engng., 2023, 28(3). <https://doi.org/10.1007/s10664-022-10278-4>. DOI: 10.1007/s10664-022-10278-4.
- [19] LADISA P, PLATE H, MARTINEZ M, et al. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains[C/OL]//2023 IEEE Symposium on Security and Privacy (SP). 2023: 1509-1526. DOI: 10.1109/SP46215.2023.10179304.
- [20] BOGAERTS F C G, IVAKI N, FONSECA J. A Taxonomy for Python Vulnerabilities[J/OL]. IEEE Open Journal of the Computer Society, 2024, 5: 368-379. DOI: 10.1109/OJCS.2024.3422686.
- [21] RÍOS F. Hugging Face's two million models and counting[EB/OL]. AI World. 2025. <https://aiworld.eu/stories/hugging-face-two-million-models>.
- [22] JFrog Security Research Team. Examining Malicious Hugging Face ML Models with Silent Backdoor[EB/OL]. JFrog Security Research. 2025. <https://research.jfrog.com/examining-malicious-hugging-face-ml-models-with-silent-backdoor/>.
- [23] JI Y, ZHANG X, WANG T. Backdoor attacks against learning systems[C/OL]//2017 IEEE Conference on Communications and Network Security (CNS). 2017: 1-9. DOI: 10.1109/CNS.2017.82286

- 56.
- [24] GU T, LIU K, DOLAN-GAVITT B, et al. Badnets: Evaluating backdooring attacks on deep neural networks[J]. Ieee Access, 2019, 7: 47230-47244.
 - [25] TURNER A, TSIPRAS D, MADRY A. Label-consistent backdoor attacks[J]. arXiv preprint arXiv:1912.02771, 2019.
 - [26] KURAKIN A, GOODFELLOW I, BENGIO S. Adversarial machine learning at scale[J]. arXiv preprint arXiv:1611.01236, 2016.
 - [27] HUANG S, PAPERNOT N, GOODFELLOW I, et al. Adversarial attacks on neural network policies [J]. arXiv preprint arXiv:1702.02284, 2017.
 - [28] MADRY A, MAKELOV A, SCHMIDT L, et al. Towards deep learning models resistant to adversarial attacks[J]. arXiv preprint arXiv:1706.06083, 2017.
 - [29] YIN M, ZHANG J, SUN J, et al. LoBAM: LoRA-Based Backdoor Attack on Model Merging[J]. arXiv preprint arXiv:2411.16746, 2024.
 - [30] LIU H, LIU Z, TANG R, et al. Lora-as-an-attack! piercing llm safety under the share-and-play scenario[J]. arXiv e-prints, 2024: arXiv-2403.
 - [31] HUA J, WANG K, WANG M, et al. MalModel: Hiding Malicious Payload in Mobile Deep Learning Models with Black-box Backdoor Attack[J]. arXiv preprint arXiv:2401.02659, 2024.
 - [32] HITAJ D, PAGNOTTA G, DE GASPARI F, et al. Do You Trust Your Model? Emerging Malware Threats in the Deep Learning Ecosystem[J]. arXiv preprint arXiv:2403.03593, 2024.
 - [33] WANG Z, LIU C, CUI X. Evilmodel: hiding malware inside of neural network models[C]//2021 IEEE Symposium on Computers and Communications (ISCC). 2021: 1-7.
 - [34] WANG Z, LIU C, CUI X, et al. Evilmodel 2.0: bringing neural network models into malware attacks [J]. Computers & Security, 2022, 120: 102807.
 - [35] LIU T, LIU Z, LIU Q, et al. StegoNet: Turn Deep Neural Network into a Stegomalware[C/OL]// ACSAC '20: Proceedings of the 36th Annual Computer Security Applications Conference. Austin, USA: Association for Computing Machinery, 2020: 928-938. <https://doi.org/10.1145/3427228.3427268>. DOI: 10.1145/3427228.3427268.
 - [36] Splinter0. Tensorflow Remote Code Execution with Malicious Model[EB/OL]. GitHub. 2024. <https://github.com/Splinter0/tensorflow-rce>.
 - [37] CERT Coordination Center. Vulnerability Note VU#253266[EB/OL]. CERT/CC. 2025. <https://kb.cert.org/vuls/id/253266>.
 - [38] The Hacker News. New Attack Technique 'Sleepy Pickle' Targets Machine Learning Models [EB/OL]. The Hacker News. 2024. <https://thehackernews.com/2024/06/new-attack-technique-sleepy-pickle.html>.
 - [39] Pjcampbe11. Pickle-File-Attacks[EB/OL]. GitHub. 2024. <https://github.com/pjcampbe11/Pickle-File-Attacks>.
 - [40] Trail of Bits. Exploiting ML Models with Pickle File Attacks (Part 1)[EB/OL]. Trail of Bits. 2024. <https://blog.trailofbits.com/2024/06/11/exploiting-ml-models-with-pickle-file-attacks-part-1/>.
 - [41] Python Software Foundation. pickletools — Tools for pickle developers[EB]. 2023.
 - [42] Of BITS T. Fickling @ DEFCON AI Village 2021[EB]. 2021.
 - [43] Mmaitre314. Python Pickle Malware Scanner[EB]. 2024.
 - [44] Protectai. ModelScan: Protection against Model Serialization Attacks[EB]. GitHub. 2024.
 - [45] University of Toronto. How three U of T researchers discovered a GPU vulnerability that threatened AI models[EB/OL]. University of Toronto. 2024. <https://www.utoronto.ca/news/how-three-u-t-researchers-discovered-gpu-vulnerability-threatened-ai-models>.
 - [46] NVIDIA Corporation. Product Security[EB/OL]. NVIDIA. 2025. https://nvidia.custhelp.com/app/answers/detail/a_id/5630.
 - [47] NVIDIA Corporation. Product Security[EB/OL]. NVIDIA. 2025. https://nvidia.custhelp.com/app/answers/detail/a_id/5703.
 - [48] NVIDIA Corporation. Product Security[EB/OL]. NVIDIA. 2025. https://nvidia.custhelp.com/app/answers/detail/a_id/5670.
 - [49] NVIDIA Corporation. Vulnerability Analysis for Container Security[EB/OL]. NVIDIA. 2024. <https://build.nvidia.com/nvidia/vulnerability-analysis-for-container-security>.

- [50] AIMonks. The NVIDIA Scape Vulnerability: A Wake-Up Call for Closed-Source AI Infrastructure [EB/OL]. Medium. 2025. <https://medium.com/aimonks/the-nvidiascape-vulnerability-a-wake-up-call-for-closed-source-ai-infrastructure-b07a745bdac2>.
- [51] NVIDIA Corporation. Security Bulletin: NVIDIA Container Toolkit - July 2025[EB/OL]. NVIDIA. 2025. https://nvidia.custhelp.com/app/answers/detail/a_id/5659/~security-bulletin%3A-nvidia-container-toolkit---july-2025.
- [52] Wiz Research. NVIDIA AI Vulnerability CVE-2025-23266 (NVIDIA Scape)[EB/OL]. Wiz. 2025. <https://www.wiz.io/blog/nvidia-ai-vulnerability-cve-2025-23266-nvidiascape>.
- [53] MITRE Corporation. CVE-2024-53870[EB/OL]. CVE.org. 2024. <https://www.cve.org/CVERecord?id=CVE-2024-53870>.
- [54] MITRE Corporation. CVE-2024-53871[EB/OL]. CVE.org. 2024. <https://www.cve.org/CVERecord?id=CVE-2024-53871>.
- [55] MITRE Corporation. CVE-2024-53872[EB/OL]. CVE.org. 2024. <https://www.cve.org/CVERecord?id=CVE-2024-53872>.
- [56] JIA Z, MAGGIONI M, STAIGER B, et al. Dissecting the NVIDIA volta GPU architecture via microbenchmarking[J]. arXiv preprint arXiv:1804.06826, 2018.
- [57] JIA Z, MAGGIONI M, SMITH J, et al. Dissecting the nvidia turing t4 gpu via microbenchmarking [J]. arXiv preprint arXiv:1903.07486, 2019.
- [58] ABDELKHALIK H, ARAFA Y, SANTHI N, et al. Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis[C]//2022 IEEE High Performance Extreme Computing Conference (HPEC). 2022: 1-8.
- [59] JARMUSCH A, GRADDON N, CHANDRASEKARAN S. Dissecting the NVIDIA Blackwell Architecture with Microbenchmarks[J]. arXiv preprint arXiv:2507.10789, 2025.
- [60] LUO W, FAN R, LI Z, et al. Dissecting the NVIDIA Hopper Architecture through Microbenchmarking and Multiple Level Analysis[J]. arXiv preprint arXiv:2501.12084, 2025.
- [61] NAGHIBIJOUBARI H, NEUPANE A, QIAN Z, et al. Rendered insecure: GPU side channel attacks are practical[C]//Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. 2018: 2139-2153.
- [62] ZHANG Z, ALLEN T, YAO F, et al. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG[C]//Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. 2023: 960-974.
- [63] NAYAK A, B P, GANAPATHY V, et al. (mis) managed: A novel tlb-based covert channel on gpus [C]//Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. 2021: 872-885.
- [64] DUTTA S B, NAGHIBIJOUBARI H, ABU-GHAZALEH N, et al. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems[C]//2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). 2021: 972-984.
- [65] GUO Y, ZHANG Z, YANG J. GPU Memory Exploitation for Fun and Profit[C/OL]//33rd USENIX Security Symposium (USENIX Security 24). Philadelphia, PA: USENIX Association, 2024: 4033-4050. <https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan>.
- [66] MITTAL S, ABHINAYA S B, REDDY M, et al. A Survey of Techniques for Improving Security of GPUs[J]. Journal of Hardware and Systems Security, 2018, 2(3): 266-285.
- [67] MIELE A. Buffer Overflow Vulnerabilities in CUDA: A Preliminary Analysis[J]. Journal of Computer Virology and Hacking Techniques, 2016, 12(2): 113-120.
- [68] PARK S O, KWON O, KIM Y, et al. Mind Control Attack: Undermining Deep Learning with GPU Memory Exploitation[J]. Computers & Security, 2020, 102: 102115.
- [69] SORENSEN T, KHLAAF H. LeftoverLocals: Listening to LLM Responses Through Leaked GPU Local Memory[J]. arXiv preprint arXiv:2401.16603, 2024.
- [70] ROELS J, JACOBS A, VOLCKAERT S. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World[C/OL]//EuroSec'25: Proceedings of the 18th European Workshop on Systems Security. Rotterdam, Netherlands: Association for Computing Machinery, 2025: 40-48. <https://doi.org/10.1145/3722041.3723099>. DOI: 10.1145/3722041.3723099.
- [71] GitHub, Inc. GitHub[EB/OL]. GitHub. 2008. <https://github.com>.

- [72] Stack Overflow. Stack Overflow[EB/OL]. Stack Overflow. 2008. <https://stackoverflow.com>.
- [73] Python Software Foundation. Python Package Index (PyPI)[EB/OL]. Python Software Foundation. 2003. <https://pypi.org>.
- [74] Apache Software Foundation. Maven Central Repository[EB/OL]. Sonatype. 2003. <https://central.sonatype.org>.
- [75] Rust Foundation. Crates.io[EB/OL]. Rust Foundation. 2014. <https://crates.io>.
- [76] GNU Project. The GNU C Library (glibc)[EB/OL]. Free Software Foundation. 1992. <https://www.gnu.org/software/libc/>.
- [77] NVIDIA Corporation. CUDA-X Libraries[EB/OL]. NVIDIA. 2007. <https://developer.nvidia.com/cuda/cuda-x-libraries>.
- [78] SARKAR V, KAMPKÖTTER A, HERMANN B. CoPhi-Mining C/C++ Packages for Conan Ecosystem Analysis[C]//2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR). 2025: 586-590.
- [79] Albumentations Team. AlbumentationsX[EB/OL]. GitHub. 2024. <https://github.com/albumentations-team/AlbumentationsX>.
- [80] Albumentations Team. AlbumentationsX setup.py[EB/OL]. GitHub. 2024. <https://github.com/albumentations-team/AlbumentationsX/blob/main/setup.py>.
- [81] Python Packaging Authority. pip Documentation[EB/OL]. Python Packaging Authority. 2008. <https://pip.pypa.io/en/stable/index.html>.
- [82] WANG C, WU R, SONG H, et al. smartpip: A smart approach to resolving python dependency conflict issues[C]//Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 2022: 1-12.
- [83] FAN G, WANG C, WU R, et al. Escaping dependency hell: finding build dependency errors with the unified dependency graph[C]//Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020: 463-474.
- [84] FreeBuf.COM. 已公开利用的 Linux 内核高危漏洞: CVE-2024-36904[EB/OL]. FreeBuf. 2025. <https://www.freebuf.com/vuls/424945.html>.
- [85] 胡金鱼. 黑客劫持 NPM 包进行供应链攻击[EB]. 2025.
- [86] 绿盟科技. 俄乌热战背景下的 Node-ipc 供应链投毒攻击[EB/OL]. 绿盟科技技术博客. 2022. <https://blog.nsfocus.net/node-ipc-npm/>.
- [87] LOKER D. AI code creates 1.7x more problems[EB/OL]. 2025. <https://www.coderabbit.ai/blog/stat-e-of-ai-vs-human-code-generation-report>.
- [88] PAN Z, SHEN W, WANG X, et al. Ambush From All Sides: Understanding Security Threats in Open-Source Software CI/CD Pipelines[J/OL]. IEEE Transactions on Dependable and Secure Computing, 2024, 21(1): 403-418. DOI: 10.1109/TDSC.2023.3253572.
- [89] CVE-2020-14188: Arbitrary Code Execution via Malicious GitHub Issue[EB]. 2020.
- [90] GUI X, LIU J, CHI M, et al. Analysis of malware application based on massive network traffic[J]. China Communications, 2016, 13(8): 209-221.
- [91] TRAN C. The SolarWinds attack and its lessons[J]. E-International Relations, 2021.
- [92] DING Y, TIAN W, CUI B. Security Analysis of Dependency Confusion for Third Party Components: A Risk Assessment Framework Based on Source Code Tree Matching[J]. 2024.
- [93] TANWAR S, KIM H W. A study on Dirty Pipe Linux vulnerability[J]. International Journal of Internet, Broadcasting and Communication, 2022, 14(3): 17-21.
- [94] 国家信息安全漏洞共享平台. Ollama 未授权访问漏洞 (CNVD-2025-04094)[EB/OL]. 2025. <https://www.cnvd.org.cn/flaw/show/CNVD-2025-04094>.
- [95] National Institute of Standards and Technology. CVE-2024-37032 Detail[EB/OL]. 2024. <https://nvd.nist.gov/vuln/detail/CVE-2024-37032>.
- [96] National Vulnerability Database. CVE-2021-44228 Detail[Z]. <https://nvd.nist.gov/vuln/detail/CVE-2021-44228>. Accessed: 2026-01-29. 2021.
- [97] FOX B. 2026 State of the Software Supply Chain[Z]. https://www.sonatype.com/hubfs/1-2025_Webside-Assets/SSCR_2025/SSCR_2026_final.pdf. Accessed: 2026-02-07. 2026.
- [98] Python Enhancement Proposal Authors. PEP 517 – A build-system independent format for Python source trees[Z]. <https://peps.python.org/pep-0517/>. 2015.

- [99] Python Enhancement Proposal Authors. PEP 518 – Specifying Minimum Build System Requirements for Python Projects[Z]. <https://peps.python.org/pep-0518/>. 2016.
- [100] Python Enhancement Proposal Authors. PEP 621 – Storing project metadata in pyproject.toml[Z]. <https://peps.python.org/pep-0621/>. 2020.
- [101] WANG J, LI L, ZELLER A. Restoring execution environments of Jupyter notebooks[C]//2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 2021: 1622-1633.
- [102] HORTON E, PARNIN C. Dockerizeme: Automatic inference of environment dependencies for python code snippets[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 2019: 328-338.
- [103] Python Packaging Authority. Dependency Resolution in pip[Z]. <https://pip.pypa.io/en/stable/topics/dependency-resolution/>. 26.0.1. Accessed: 2026-02-12. 2025.
- [104] LI S, LIU J, TIAN H, et al. EasyPip: Detect and Fix Dependency Problems in Python Dependency Declaration Files.[C]//SEKE. 2023: 162-168.
- [105] DUAN R, ALRAWI O, KASTURI R P, et al. Towards measuring supply chain attacks on package managers for interpreted languages[J]. arXiv preprint arXiv:2002.01139, 2020.
- [106] GUO W, XU Z, LIU C, et al. An Empirical Study of Malicious Code In PyPI Ecosystem[C]//2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2023: 166-177.
- [107] RUOHONEN J, HJERPPE K, RINDELL K. A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI[C]//2021 18th International Conference on Privacy, Security and Trust (PST). 2021: 1-10.
- [108] VU D L, PASHCHENKO I, MASSACCI F, et al. Typosquatting and combosquatting attacks on the python ecosystem[C]//2020 IEEE European Symposium on Security and Privacy Workshops. 2020: 509-514.
- [109] FOUNDATION P S. Python Documentation[EB/OL]. 2022. <https://docs.python.org/3/>.
- [110] CAO Y, CHEN L, MA W, et al. Towards Better Dependency Management: A First Look at Dependency Smells in Python Projects[J/OL]. IEEE Trans. Softw. Eng., 2023, 49(4): 1741-1765. <https://doi.org/10.1109/TSE.2022.3191353>. DOI: 10.1109/TSE.2022.3191353.
- [111] FOUNDATION P S. Python Dependency Specifiers[EB/OL]. 2022. <https://packaging.python.org/en/latest/specifications/dependency-specifiers/>.
- [112] Sarugaku. resolvelib[EB/OL]. 2022. <https://github.com/sarugaku/resolvelib>.
- [113] Bandersnatch Developers. bandersnatch[EB/OL]. 2022. <https://bandersnatch.readthedocs.io/en/latest/>.
- [114] Dylanhogg. Python Awesome Project[EB/OL]. 2022. <https://awesomepython.org/>.
- [115] Wtsi-hgi. Python-hgjson issue[EB/OL]. 2025. <https://github.com/wtsi-hgi/python-hgjson/issues/14>.
- [116] Opendatateam. Cookiecutter-udata-plugin issue[EB/OL]. 2025. <https://github.com/opendatateam/cookiecutter-udata-plugin/issues/3>.
- [117] Radekd91. Emoca issue[EB/OL]. 2025. <https://github.com/radekd91/emoca/issues/44>.
- [118] Pypy. 2022. <https://github.com/pypy/pip/issues/4625>.
- [119] Pycrypto. Pycrypto issue[EB/OL]. 2022. <https://github.com/pycrypto/pycrypto/issues/156>.
- [120] Albumentations-team. 2022. <https://github.com/albumentations-team/albumentations/issues/841>.
- [121] FOUNDATION P S. opencv-python[EB/OL]. 2022. <https://pypi.org/project/opencv-python/>.
- [122] Pypy. 2022. <https://github.com/pypy/pip/issues/8509>.
- [123] LarsVoelker. 2022. <https://github.com/LarsVoelker/FibexConverter/issues/7>.
- [124] FOUNDATION P S. Python Standard Libraries Documentation[EB/OL]. 2022. <https://docs.python.org/3.10/library/index.html>.
- [125] Pypy. Package Discovery and Namespace Packages[A/OL]. 2023. https://setuptools.pypy.io/en/latest/userguide/package%5C_discovery.html.
- [126] Lambdaloop. aniposelib[EB/OL]. 2022. <https://github.com/lambdaloop/anipose/issues/22>.

附录

A 一个附录

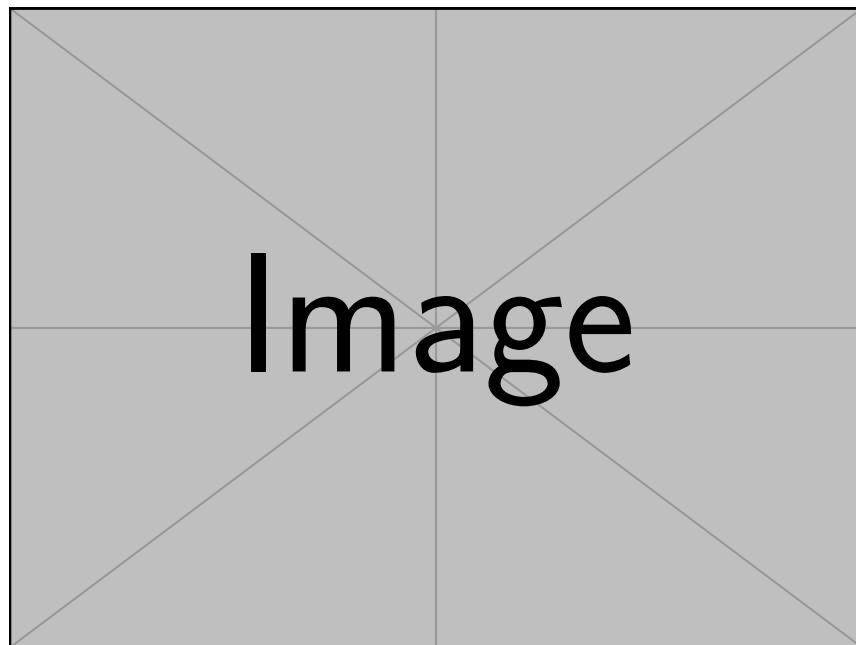


图 A.1 附录中的图片

B 另一个附录