软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(7):2719−2741 [doi: 10.13328/j.cnki.jos.007584] [CSTR: 32375.14.jos.007584]

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

Python 软件包库中 C/C++外部语言调用的安全性分析*

胡明哲[1], 丁秋然[1], 张子涵[1], 谢金言[1], 于 乐[1], 韩丽萍[1,2]

[1](南京邮电大学 计算机学院、软件学院、网络空间安全学院, 江苏 南京 210023)

[2](计算机软件新技术全国重点实验室 (南京大学), 江苏 南京 210093)

通信作者: 于乐, E-mail: yulele08@njupt.edu.cn

摘　要: 软件包库的安全性是软件供应链分析的重要一环, 但是现有研究和工具往往缺乏对软件包库中外部语言调用的有效分析. PyPI 是 Python 语言的官方软件包库, 其中存储海量不同应用领域的 Python 软件包. 这些软件包除了包含 Python 语言编写的程序外, 还常包含通过 Python 的外部接口 Python/C API 调用的 C/C++外部语言程序. 分析 Python 软件包库中外部语言调用的安全性对于保障软件供应链的安全可靠具有重要意义. 通过分析互操作官方文档以及互操作程序分析的相关方法和工具, 构建 Python-C/C++互操作程序的漏洞基准套件, 其中包括 9 小类行为共计 15 种漏洞模式的基准测试程序, 覆盖内存、类型、异常、并发和数值这 5 大类语言特性以及安装量最大的 16 个包含 C/C++外部调用的 Python 软件包中的互操作程序漏洞. 通过在漏洞基准套件上评估已有的先进 Python-C/C++互操作漏洞检查工具, 对比分析现有研究和工具的可靠性、完备性和可扩展性, 分析总结 Python-C/C++互操作安全分析的现状和不足. 通过分析超过 700 个漏洞警告, 确认在 6 个 PyPI 库中新发现 3 种共计 21 个实际漏洞.

关键词: 软件供应链分析; 跨语言互操作; 漏洞检查; 程序分析; 漏洞基准库

中图法分类号: TP311

中文引用格式: 胡明哲, 丁秋然, 张子涵, 谢金言, 于乐, 韩丽萍. Python软件包库中C/C++外部语言调用的安全性分析. 软件学报, 2026, 37(7): 2719–2741. http://www.jos.org.cn/1000-9825/7584.htm

英文引用格式: Hu MZ, Ding QR, Zhang ZH, Xie JY, Yu L, Han LP. Safety Analysis of C/C++ Foreign Language Calls in Python Software Package Repositories. Ruan Jian Xue Bao/Journal of Software, 2026, 37(7): 2719–2741 (in Chinese). http://www.jos.org.cn/1000-9825/7584.htm

Safety Analysis of C/C++ Foreign Language Calls in Python Software Package Repositories

HU Ming-Zhe[1], DING Qiu-Ran[1], ZHANG Zi-Han[1], XIE Jin-Yan[1], YU Le[1], HAN Li-Ping[1,2]

[1](School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China)

[2](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210093, China)

Abstract: The safety of software package repositories is a critical aspect of software supply chain analysis, but existing research and tools often lack effective analysis of foreign language calls in the package repositories. As the official package repository for Python, PyPI stores a vast amount of Python software packages from various application domains. In addition to programs written in Python, these packages often include C/C++ foreign language programs that are called via Python's foreign interface—the Python/C API. Analyzing the safety of foreign language calls in Python package repositories is crucial for ensuring the safety and reliability of software supply chains.

* 基金项目: 国家自然科学基金 (62502226, 62202406); 计算机软件新技术全国重点实验室 (南京大学) 开放课题 (KFKT2025B66); 江苏省高等学校基础科学 (自然科学) 研究项目 (25KJB520029); 南京邮电大学引进人才科研启动基金 (自然科学) (NY224026, NY224023, NY224001, NY223164); 南京邮电大学校级自然科学基金 (NY224143)
本文由"智能化基础软件可信构造和供应链安全"专题特约编辑王林章教授、司徒凌云研究员、钟浩研究员、 向剑文教授、张路教授推荐.
收稿时间: 2025-09-06; 修改时间: 2025-10-20; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-26
CNKI 网络首发时间: 2026-06-02

By analyzing official documentation for interoperability and relevant methods and tools for interoperability program analysis, a bug benchmark suite for Python-C/C++ interoperability programs is established. This suite includes benchmark test programs for 15 bug patterns across nine categories, covering five language features of the memory, type, exception, concurrency, and numerical issues, as well as interoperability program bugs in the 16 most installed PyPI software packages that involve C/C++ foreign calls. By evaluating the existing state-of-the-art Python-C/C++ interoperability bug checkers on the bug benchmark suite, a comparative analysis of the reliability, completeness, and scalability of existing research and tools is conducted, with the current status and limitations of Python-C/C++ interoperability safety analysis analyzed and summarized. By analyzing more than 700 bug warnings, 21 new real-world bugs across three bug patterns are found in six PyPI repositories.

Key words: software supply chain analysis; cross-language interoperability; bug detection; program analysis; bug benchmark

Python 凭借高效的开发效率、对领域应用的生态支持、编译辅助的安全机制等优势吸引了越来越多的开发者, 其官方软件包库 PyPI 包含超过 63 万个项目, 日平均安装数超过 6 亿项目次 (2025 年 5 月 PyPI Stats 数据). 保障 Python 软件包库的安全性对于软件供应链安全至关重要. 研究揭示 PyPI 库存在安全和隐私问题[1,2], 已有的检查方法仅针对 PyPI 库中的 Python 语言程序[3−5].

然而, 除了 Python 宿主语言程序外, PyPI 库还允许包含 C/C++外部语言程序, 以及桥接 Python-C/C++的跨语言互操作程序. 外部语言 C/C++包含大量经过测试和优化的存量代码, 能够避免重复的开发劳动并提供更优的时空性能. Python-C/C++互操作的软件架构被应用于诸多流行 Python 软件包如深度学习框架 TensorFlow 和 PyTorch、科学计算库 NumPy、图像处理库 Pillow 等. 相对于单语言软件, 跨语言互操作可能引入更多的程序漏洞且更加难以分析[6], 其中 Python-C/C++互操作由于语言特性差异更大而更加易错[7], 并且互操作程序中 C/C++外部调用的安全问题难以被单语言检查工具有效分析[8,9]. 分析 Python 软件包库中 C/C++外部语言调用的安全性是对 Python 软件供应链安全的重要补足.

现有的 Python-C/C++互操作程序的安全性分析研究一方面扩展分析 Java-C/C++互操作程序的漏洞模式分类在 Python-C/C++上的行为, 一方面针对具体的 Python-C/C++互操作程序漏洞设计实现检查工具. Hu 等人[10]扩展研究了 Java-C/C++互操作程序的漏洞模式分类[8], 分析了相关漏洞模式在 Python-C/C++互操作程序中的表现形式. 针对引用计数错误的漏洞检查, Pungi[11]利用仿射变换与分析来检查内存泄漏、悬空指针等漏洞, RID[12]基于不一致路径对检查提升了 Pungi 的精度. PyRefcon[13]提出基于生命周期建模的检测方法, 结合符号执行引擎对对象状态进行路径敏感追踪, 以准确检查包括引用泄漏与释放后使用在内的漏洞. Li 等人[14,15]先后提出 PolyCruise 和 PolyFuzz, 通过轻量级静态分析结合动态的选择性插桩、信息流分析与灰盒模糊测试, 检查诸如缓冲区溢出、整数溢出、除零错误等漏洞. Mopsa[16]构建共享抽象域的联合抽象解释框架以检查类型一致性错误、整数溢出、异常处理错误等漏洞. PyCType[17]基于跨语言的类型推断对外部函数的声明与实现进行一致性检查. CPyChecker 基于 GCC 构建数据流分析插件, 检测 Python/C API 及其使用, 识别引用计数错误、异常处理错误、格式化字符串不匹配等漏洞.

然而, 这些研究普遍存在漏洞模式归纳不系统、漏洞检查能力评估不全面的问题. 首先, 由于 Java 和 Python 作为宿主语言在内存管理、类型系统、并发控制等语言特性上有较大的差异, 扩展的漏洞模式分类[10]和针对特定漏洞的测试集[14]仍有较多缺漏, 不能系统地分类总结 Python-C/C++互操作程序的漏洞模式. 其次, Python 语言社区活跃且迭代迅速, 特定语言版本的测试用例不能反映漏洞模式的变化, 例如 Python 3.7 在外部函数声明时引入新的调用约定标志 METH_FASTCALL、Python 3.12 引入引用计数不会改变的永生对象等, 这些变更影响 Python-C/C++互操作程序的行为, 导致现有工具在类型一致性、引用计数等漏洞检查时存在误报、漏报等问题. 最后, 已有研究缺乏针对真实软件供应链的全面评估, 无法多维度地反映现行语言版本下不同先进工具的漏洞检查能力, 例如 PyRefcon 的实验评估中与 Pungi 和 RID 对比时使用的包大多过于老旧且规模较小, 而与 CPyChecker 对比使用的包则存在 CPyChecker 无法完全编译的问题.

针对上述问题, 本文基于 Python 最新稳定版本 Python 3.13 的互操作官方文档和先进 Python-C/C++互操作程序分析方法和工具的对比分析, 设计实现涵盖内存、类型、异常、并发和数值这 5 大类语言特性, 包括 9 小类行

为共计 15 种漏洞模式的漏洞基准测例; 通过识别互操作接口, 在 Python 官方软件包库 PyPI 中选取安装量最大的 16 个互操作软件包, 提取其中的互操作程序; 针对 2010 年以来代表性的先进 Python-C/C++互操作漏洞检查研究, 在漏洞基准测例和互操作软件包上进行详细的对比实验, 评估分析先进工具的可靠性、完备性和可扩展性, 揭示在漏洞基准测例上漏报率高、在供应链软件包库上误报率高、可扩展性差的问题, 通过其中局部较优的解决方案和整体不足启示后续研究的发展方向. 同时在安装量最大的 16 个互操作软件包上, 先进工具评估报告超过 700 个漏洞警告, 经过人工分析检查, 确认在 6 个软件包中新发现 3 种共计 21 个实际漏洞. 漏洞基准套件的构建及评估系统的设计目标如图 1 所示.

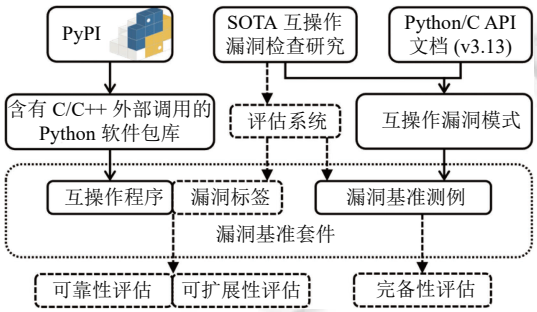

图 1 漏洞基准套件构建及评估系统设计目标

本文主要贡献如下.

● 漏洞分类与基准构建. 本文系统地收集并归纳 Python 的 C/C++外部调用的安全隐患, 并据此构建了覆盖 15 种漏洞模式的基准套件, 包括不同语言特性互操作带来的漏洞测例, 以及 Python 软件供应链中高安装量的互操作软件包中的实际工程漏洞.

● 评估系统与实证发现. 通过设计统一的评估系统, 基于漏洞基准套件, 我们全面评估已有的先进 Python-C/C++互操作安全性分析工具, 确认新发现 3 种漏洞模式共 21 个真实存在于 6 个 PyPI 高安装量软件包的真实漏洞.

● 现状与发展方向分析. 评估结果反映互操作安全检查在可靠性、完备性、可扩展性等方面现存的不足, 揭示复杂语言特性互操作分析能力的缺陷, 以及复杂系统动静态分析的构建要求. 漏洞基准套件和评估分析可以作为未来互操作安全性分析方法和技术改进的对照标准与基础设施. 本文首先介绍 Python-C/C++互操作的基本机制 (第 1 节), 然后通过系统的漏洞模式分类 (第 2 节) 和漏洞基准套件构建 (第 3 节), 全面地评估已有的先进 Python-C/C++互操作程序漏洞检查研究和工具 (第 4 节), 最后给出总结与展望 (第 5 节), 揭示 Python-C/C++互操作程序安全性分析的现状和不足.

1 Python-C/C++互操作

Python/C API 是 CPython 编译器核心提供给 C/C++程序的编程接口, 用以实现 Python 程序的 C/C++扩展模块 (包括 CPython 标准库的一部分) 或是将 Python 运行时嵌入 C/C++程序. Python 的 C/C++扩展模块被广泛使用于 Python 生态的软件供应链, 如深度学习框架 TensorFlow 和 PyTorch、自动驾驶平台 Apollo、加密货币 Bitcoin 等. 如图 2 所示是来自 protobuf 项目的 Python-C/C++互操作实例. protobuf (protocol buffer) 是 Google 开源的用以序列化结构化数据的项目, 底层使用 C 实现, 并提供多种语言接口. PyPI 同名软件包 protobuf 是其 Python 接口, 累积下载超过 4 亿次, 是诸多数据密集型项目如深度学习框架 TensorFlow 等的核心组件.

Python/C API 定义了一系列宏、函数和变量, 通过在跨语言互操作程序中包含 Python.h 头文件引入, 是 C/C++程序操作 Python 运行时的接口. 如图 2 中 message.c 第 24–29 行, Python/C API 结构体 PyType_Spec 类型的变量 PyUpb_Message_Spec 定义扩展类 Message; PyUpb_Message_Slots 是一个 PyType_Slot 类型的 Python/C API 结构体数组, 定义在第 20–22 行; PyUpb_Message_Methods 是一个 PyMethodDef 类型的 Python/C API 结构体数组,

定义了扩展类中的外部方法, 以第 15–18 行为例, 调用名为 SerializeToString 的 Python 外部方法指向 C 函数实现 PyUpb_Message_SerializeToString. 如图 2 右下方 proto.py 所示, Python 无法区分 message.SerializeToString 是 C/C++实现的外部函数还是 Python 本地函数; 其实参中的位置参数和关键字参数被分别打包传给对应的 C 实现 PyUpb_Message_SerializeToString (message.c 第 2–12 行), 再由 Python/C API 函数 PyArg_ParseTupleAndKeywords 进行解包和跨语言的类型转换, 其中格式化串"|p"要求 Python 侧接收一个 bool 类型的可选参数, 并转换为 C 中 int 类型的 0/1 值. 在 Python 3.13 版本中, 格式化串支持可嵌套的 37 种格式化单元, Python-C/C++互操作漏洞检查器 CPyChecker 受限于其支持的 Python 版本较低, 无法识别格式化单元 p, 导致会产生一个误报.

```
   // protobuf/python/message.c
1  #include "Python.h"
2  PyObject* PyUpb_Message_SerializeToString(
3      PyObject* _self, PyObject* args, PyObject* kwargs) {
4    PyUpb_Message* self = (void*)_self;
5    if (!PyUpb_Message_Verify((PyObject*)self))
6      return NULL;
7    static const char* kwlist[] = {"deterministic", NULL};
8    int deterministic = 0;
9    if (!PyArg_ParseTupleAndKeywords(
10       args, kwargs, "|p", (char**)(kwlist), &deterministic))
11     return NULL;          CPyChecker: warning
12 ...}
13
14 static PyMethodDef PyUpb_Message_Methods[] = {
15   {"SerializeToString",
16    (PyCFunction)PyUpb_Message_SerializeToString,
17    METH_VARARGS | METH_KEYWORDS,
18    "Serializes the message to a string, ..."},
19   {NULL, NULL}};
```

```
   // protobuf/python/message.c
20 static PyType_Slot PyUpb_Message_Slots[] = {
21   {Py_tp_methods, PyUpb_Message_Methods},
22   {0, NULL}};
23
24 PyType_Spec PyUpb_Message_Spec = {
25   PYUPB_MODULE_NAME ".Message",
26   sizeof(PyUpb_Message),
27   0,
28   Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
29   PyUpb_Message_Slots,};
```

```
   # protobuf/python/google/protobuf/proto.py
1  from google.protobuf.message import Message
2  def serialize(message: Message,
3                deterministic: bool = None) -> bytes:
4    return message.SerializeToString(
5             deterministic=deterministic)
```

图 2 protobuf 互操作实例及 CPyChecker 漏洞检查误报

2 漏洞模式分类

为了系统地分类定义 Python-C/C++互操作程序可能存在的安全漏洞, 本文完整地分析整理 Python/C API 官方文档中提到的安全隐患, 并设计实现对应的漏洞模式和测试用例; 同时针对已有的不同语言的互操作安全性分析研究提及的漏洞, 根据其语言特性和运行时行为的异同对漏洞模式进行了归纳和完整测例设计实现.

漏洞模式分析整理的来源包括: (1) Python 3.13 版本 Python/C API 官方文档; (2) 已有的 Python-C/C++跨语言互操作安全性分析研究和工具[9−17]; (3) 已有的其他语言互操作安全性研究[8,18]; (4) CVE 库及 Python-C/C++互操作软件开源漏洞列表. 计算机专业的一名研究生和一名本科生分别通读上述文献并实现了其中所有示例代码和漏洞, 包括文档中所有提及可能存在隐患的情形, 以及其他语言的外部调用漏洞在 Python 中对应的情形, 同时对 Python 3.13 版本有变更的漏洞模式进行了调整. 在两名具有博士学位和相关研究背景的专家的指导下, 对两位学生实现的漏洞进行了审计、整理和分类. 如图 3 所示为漏洞模式分类流程图.

具体而言, 首先在语言特性层面, 我们根据导致漏洞的语言特性差异将其归纳为 5 大类, 分别是内存模型 (memory model, M)、类型系统 (type system, T)、异常处理 (exception handling, E)、并发控制 (concurrency control, C) 和数值计算 (numerical computation, N). 这一抽象层对应 Python 与 C/C++在语言设计上的本质差异, 是引发安全漏洞的根本原因[9]. 其次, 针对每大类语言特性, 可能存在多小类的漏洞模式, 每个小类对应 Python-C/C++互操作中的一类行为 (Python/C API 集合). 以内存模型大类为例, 其可以进一步分为引用计数错误、内存管理错误和缓冲区溢出这 3 小类. 最后, 根据漏洞实际的运行时行为, 每一小类又包含若干种细粒度的漏洞模式. 每种细粒度漏

洞模式分配唯一编号 (如 M1.1、T2.2 等), 作为后续漏洞检查工具分析评估和漏洞基准测例索引的标签. 该自顶向下的漏洞模式框架提升分类体系的可解释性, 也为漏洞基准套件设计与工具效果评估提供系统化的分类标准, 同时为后续漏洞检查方法扩展和设计提供理论基础.

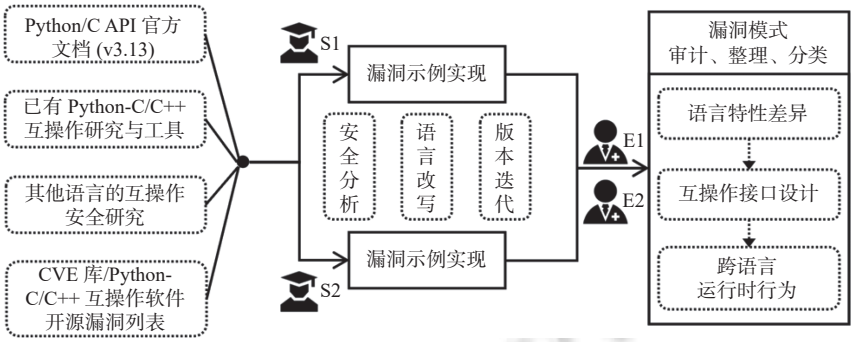

图 3 漏洞模式分类流程图

基于上述分类方法设计, 本文系统地构建了一个覆盖全面、结构清晰的 Python-C/C++互操作程序漏洞模式分类体系 (如表 1 所示).

表 1 漏洞模式分类表

语言特性	漏洞模式小类	小类编号	细粒度漏洞模式	细粒度编号
内存模型	引用计数错误	M1	内存泄漏	M1.1
			悬空引用或对象过早释放	M1.2
	内存管理错误	M2	构造析构不匹配	M2.1
			非法内存分配与释放	M2.2
	缓冲区溢出	M3	缓冲区越界访问	M3.1
类型系统	整数溢出	T1	跨语言传参溢出	T1.1
			接口层运算溢出	T1.2
	类型一致性错误	T2	格式化字符串不匹配	T2.1
			调用约定不匹配	T2.2
异常处理	异常处理错误	E1	异常后非预期控制流	E1.1
			异常重复抛出	E1.2
			异常状态与返回值不一致	E1.3
	不完整的错误检查	E2	未检查接口函数返回值	E2.1
并发控制	数据竞争	C1	GIL导致的临界区阻塞	C1.1
数值计算	除零错误	N1	接口层整数除零操作	N1.1

下面将围绕上述漏洞模式分类, 系统地介绍每种漏洞模式并分析其语言特性差异及带来的 Python-C/C++外部语言调用的安全性问题.

2.1 内存模型

2.1.1 引用计数错误 (M1)

Python 是带有自动垃圾收集机制的语言, 解释器通过引用计数跟踪每个 Python 对象的生命周期; 而在 C/C++中, 内存管理则依赖显式的分配与释放操作. 当 C/C++扩展模块持有 Python 对象的引用时, 开发者需通过调用特定的 Python/C API 如 Py_INCREF 和 Py_DECREF 手动增减引用计数. 由于借引用和偷引用行为[19]的存在, 手动管理引用计数并不容易, 引用计数错误可能导致内存泄漏 (M1.1)、悬空引用或对象过早释放 (M1.2) 等错误, 进而引发运行时错误乃至安全攻击.

在代码 1 的测例 1 中, 外部函数实现 leaky_creation 通过 Python/C API 函数 PyLong_FromLong 创建一个新

Python 对象, 其引用计数在生命周期结束时仍然为 1, 导致其无法被回收, 形成内存泄漏 (M1.1). 代码 1 的测例 2 中, 外部函数实现 bad_return_ref 通过 Python/C API 函数 PyList_GetItem 获取列表对象中的元素, 由于该函数返回一个借引用, 在将其返回至 Python 侧时应增加其引用计数, 以避免原列表对象被垃圾回收而使得 Python 侧持有一个悬空引用 (M1.2), 该漏洞可能导致段错误或未定义行为.

代码 1. 引用计数错误测例.

```
/*测例 1: 引用计数内存泄漏 (M1.1)*/
static PyObject* leaky_creation(PyObject *self, PyObject *args) {
  PyObject *num = PyLong_FromLong(42);
  //错误: 创建对象但未正确管理引用, 导致内存泄漏
  Py_RETURN_NONE;
}
/*测例 2: 引用计数悬空引用 (M1.2)*/
static PyObject* bad_return_ref(PyObject *self, PyObject *args) {
  PyObject *list;
  if (!PyArg_ParseTuple(args, "O!", &PyList_Type, &list)) return NULL;
  PyObject *item = PyList_GetItem(list, 0);
  //错误: 直接返回借引用, 未增加引用计数, 可能导致悬空引用
  return item;
}
```

2.1.2 内存管理错误 (M2)

Python 和 C/C++在内存管理机制上存在显著差异. C/C++依赖标准库函数 (如 malloc 和 free) 进行显式的内存分配与释放, 而 Python 则采用其内建的内存管理器在私有堆上进行分配. Python 和 C/C++通过外部接口 Python/C API 互操作时, 共享地址空间中的 Python 对象需要使用 Python 运行时的内存接口进行分配和释放. 这些内存接口并不直接向操作系统请求资源, 而是通过其内部内存分配策略进行统一调度, 以提升性能与跨平台的一致性. 如表 2 所示, 根据内存域的不同, Python-C/C++互操作程序中包含几族作用不同的内存接口. 构造析构不匹配 (M2.1) 可能破坏共享地址空间, 导致内存泄漏或未定义行为.

表 2 Python-C/C++互操作程序内存接口

内存域	分配函数	分配函数作用	释放函数	释放函数作用
Raw Memory接口 (线程安全/无需GIL)	PyMem_RawMalloc	分配原始内存块, 返回未初始化的内存指针, 请求0字节返回非NULL指针	PyMem_RawFree	释放Raw域分配的内存块
	PyMem_RawCalloc	分配并清零的连续内存块, 适合数组初始化	PyMem_RawFree	
	PyMem_RawRealloc	调整内存块大小, 支持原地扩展或新分配	PyMem_RawFree	
通用内存接口 (需GIL)	PyMem_Malloc	从Python堆分配通用内存, 默认使用pymalloc优化小对象	PyMem_Free	释放Mem域分配的内存
	PyMem_Calloc	分配清零内存块, 用于初始化场景	PyMem_Free	
	PyMem_Realloc	重新分配通用内存块大小	PyMem_Free	
对象内存接口 (需GIL)	PyObject_Malloc	专为Python对象设计的内存分配接口	PyObject_Free	释放Object域分配的对象内存
	PyObject_Calloc	分配并初始化为0的对象内存	PyObject_Free	
	PyObject_Realloc	调整已分配对象内存的大小	PyObject_Free	
类型安全宏	PyMem_New	类型安全的封装, 等价于((type*)PyMem_Malloc(n*sizeof(type)))	PyMem_Del	等价PyMem_Free(p), 类型安全

代码 2 包含两个漏洞测例. 外部函数实现 demo_pymem_malloc_free 通过 PyMem_Malloc 从 Python 堆申请了

32 字节内存, 错误地使用 C 标准库的 free 函数释放该内存将绕过 Python 内存管理器, 可能破坏其内部状态导致未定义行为甚至内存损坏. 类似地, 外部函数实现 demo_rawmalloc_pymemfree 通过 PyMem_RawMalloc 分配原始内存块, 但错误地使用 PyMem_Free 进行释放. 不同族内存接口的内部实现机制不同, 不匹配的使用会导致内存回收异常, 引发崩溃或堆结构破坏.

代码 2. 构造析构不匹配测例.

```
//测例 1: PyMem_Malloc 错配 free (M2.1)
static PyObject* demo_pymem_malloc_free(PyObject *self, PyObject *args) {
  char *buf = PyMem_Malloc(32);
  if (!buf) return PyErr_NoMemory(), NULL;
  free(buf); //错误: PyMem_Malloc 应对应 PyMem_Free
  Py_RETURN_NONE;
}
//测例 2: PyMem_RawMalloc 错配 PyMem_Free (M2.1)
static PyObject* demo_rawmalloc_pymemfree(PyObject *self, PyObject *args) {
  void *buf = PyMem_RawMalloc(64);
  if (!buf) return PyErr_NoMemory(), NULL;
  PyMem_Free(buf); //错误: PyMem_RawMalloc 应对应 PyMem_RawFree
  Py_RETURN_NONE;
}
```

除了不同域之间内存操作 Python/C API 错配的问题外, 在表 2 中同一族内存接口内部, 重复释放、释放后使用、内存泄漏、非法释放、偏移释放等 C 语言经典内存管理错误仍然存在, 我们将其统一记为非法内存分配与释放 (M2.2). 漏洞测例如代码 3 所示. 代码 3 的测例 1 会破坏堆内存分配器的空闲链表结构或分配元数据, 进而引发程序崩溃. 代码 3 的测例 2 堆中释放的内存可能已被回收到空闲池或分配给其他对象, 后续的写操作将产生未定义行为, 可能带来数据破坏、非法访问和内存劫持的风险.

代码 3. 非法内存分配与释放测例.

```
//测例 1: 重复释放 (M2.2)
static PyObject* demo_double_free(PyObject *self, PyObject *args) {
  char *buf = PyMem_Malloc(16);
  if (!buf) return PyErr_NoMemory(), NULL;
  PyMem_Free(buf);
  PyMem_Free(buf); //错误
  Py_RETURN_NONE;
}
//测例 2: 释放后使用 (M2.2)
static PyObject* demo_use_after_free(PyObject *self, PyObject *args) {
  char *buf = PyMem_Malloc(16);
  if (!buf) return PyErr_NoMemory(), NULL;
  PyMem_Free(buf);
  buf[0] = "X"; //错误: 释放后写入
  Py_RETURN_NONE;
}
```

2.1.3 缓冲区溢出 (M3)

Python 语言提供包含编译辅助的边界检查机制, 开发者无需关心缓冲区边界, 甚至可以通过负数来倒序索引; 而 C/C++不具备运行时的边界检查能力. 当 Python 向 C/C++外部函数传递字符串、字节流、对象指针等数据时, 若互操作程序未对数据类型或长度做充分校验, 使用如 strcpy、memcpy、PyArg_ParseTuple 等直接操作指针的函数, 便可能在处理不受信任输入时写入超出目标缓冲区的内存, 触发缓冲区越界访问 (M3.1).

如代码 4 所示, 外部函数实现 unsafe_copy 使用 16 字节固定大小的缓冲区接收来自 Python 的字符串输入, 但在未检查输入长度的情况下直接调用 strcpy 复制内容, 导致当传入字符串超过 16 字节时发生栈缓冲区溢出.

代码 4. 缓冲区越界访问 (M3.1) 测例.

```
static PyObject* unsafe_copy(PyObject* self, PyObject* args) {
  const char* input;
  char buffer[16];
  if (!PyArg_ParseTuple(args, “s”, &input)) return NULL;
  strcpy(buffer, input); //不安全的字符串复制 (未检查长度)
  return PyUnicode_FromString(buffer);
}
```

2.2 类型系统

2.2.1 整数溢出 (T1)

C/C++的整数类型 (如 int、long 等) 具有固定的字长和取值范围; 而 Python 则支持统一的长整型表示, 能够动态扩展以适应不同的数值大小. 这一类型系统上的差异导致数据从 Python 传至 C/C++侧时可能引发整数溢出漏洞. 若外部函数从 Python 侧得到的实参超出目标 C/C++类型的表示范围, 就需要跨语言接口程序进行有效的边界检查或类型转换, 否则便可能导致截断、符号翻转、未定义行为等安全性漏洞.

如代码 5 所示, 外部函数实现 add_one 在调用 Python/C API 函数 PyLong_AsLong 时, 将 Python 侧传入的任意精度整数转换为 C 中的 long 类型对象. 若传入的整型值超出 long 类型的表示范围, 将引发数值截断或转换失败, 导致运行时错误或隐式信息丢失. 这一漏洞被归类为跨语言传参溢出 (T1.1).

代码 5. 整数溢出测例.

```
static PyObject* add_one(PyObject* self, PyObject* args) {
  PyObject* input_obj; long num; int x;
  if (!PyArg_ParseTuple(args, “O”, &input_obj)) return NULL;
  //将 Python 整数转换为 C long 类型, 此时若 Python 端传入数据过大可能发生截断 (T1.1)
  num = PyLong_AsLong(input_obj);
  if (num == −1 && PyErr_Occurred()) return NULL;
  x = (int)num; //显式地转换为 int(可能导致溢出)
  x += x; //接口层运算可能导致溢出, 无法返回有效值 (T1.2)
  return PyLong_FromLong(x);
}
```

其次, 即便跨语言传参过程中转换成功, 由于接口代码可以进行 C/C++的强制类型转换和整数运算, 仍存在因整数宽度不足而导致接口层运算溢出 (T1.2), 进而影响外部函数返回的正确性.

2.2.2 类型一致性错误 (T2)

在跨语言传参过程中, PyArg_ParseXXX 族 Python/C API 函数被用于外部调用的参数解析, 其依赖格式化字

符串对从 Python 侧传入的 Python 对象实参进行解包和跨语言类型转换. 跨语言的类型一致性建立在 3 个要素的一致性上: 格式化字符串的声明、Python 侧传入的对象、C/C++侧解包后接收参数的变量类型. 这种复杂性源自 Python 动态类型系统与 C/C++静态类型系统之间的本质差异, 并可能存在多种不匹配的漏洞.

如代码 6 所示是格式化字符串不匹配 (T2.1) 的漏洞测例, 在代码 6 的测例 1 中, 格式化字符串“i”要求传入参数为 Python 整型对象并将其转换为 C 的 int 类型对象, 若 Python 侧传入其他类型对象, PyArg_ParseTuple 无法完成类型转换, 返回 false 并设置 TypeError 异常. 在代码 6 的测例 2 中, 格式化字符串“ii”声明解析两个整型位置参数, 调用者仅提供一个参数时, 跨语言传参在尝试解包并转换第 2 个变量时发生错误.

代码 6. 格式化字符串不匹配 (T2.1) 测例.

```
//测例 1: 格式字符串与实际参数类型不匹配 (要求整型, 传入其他类型)
static PyObject* wrong_format_int(PyObject *self, PyObject *args) {
  int value;
  //格式字符串声明为“i” (整型), 传入其他类型触发 TypeError
  if (!PyArg_ParseTuple(args, “i”, &value)) return NULL;
  return PyLong_FromLong(value * 2);
}
//测例 2: 参数数量不匹配 (要求 2 个位置参数, 实际 C/C++侧接收变量只有 1 个)
static PyObject* wrong_arg_count(PyObject *self, PyObject *args) {
  int a;
  //格式字符串声明为“ii” (两个整型位置参数), 但 C/C++侧只有一个变量 a 去接收}
  if (!PyArg_ParseTuple(args, “ii”, &a)) return NULL;
  return PyLong_FromLong(a);
}
```

在使用 Python/C API 编写扩展模块时, 每一个外部函数 (或方法) 都必须通过 PyMethodDef 结构体进行注册. 该结构体包含 4 个字段 (如图 2 中 message.c 第 15–18 行), 其中第 3 个字段是一个比特位域, 用于指明该外部函数的调用约定. 该位域的值决定解释器在运行时如何封装参数并调用外部函数实现. 因此, 该位域的值必须与实际外部函数实现的参数类型相匹配, 调用约定不匹配 (T2.2) 将导致调用栈破坏、参数传递错误等安全漏洞. 所有合法的位域组合及其含义和约束如表 3 所示.

表 3 PyMethodDef 结构体调用约定位域标志及其适配的外部函数实现类型

标志组合	标志值 (十六进制)	用法	参数数量	适配外部函数实现类型
METH_VARARGS	0x0001	处理位置参数 (元组形式)	2	PyCFunction
METH_VARARGS | METH_KEYWORDS	0x0003	处理位置和关键字参数	3	PyCFunctionWithKeywords
METH_FASTCALL	0x0080	快速调用 (仅位置参数)	3	PyCFunctionFast
METH_FASTCALL | METH_KEYWORDS	0x0082	快速调用 (支持关键字参数)	4	PyCFunctionFastWithKeywords
METH_METHOD | METH_FASTCALL | METH_KEYWORDS	0x0282	支持定义式类方法	5	PyCMethod
METH_NOARGS	0x0004	无参数	2	PyCFunction
METH_O	0x0008	单个对象参数	2	PyCFunction
METH_CLASS	0x0010	类方法 (绑定到类)	2	PyCFunction
METH_STATIC	0x0020	静态方法 (无实例绑定)	2	PyCFunction

如代码 7 所示, 测例 1 的外部函数实现 wrong_flags_noargs 使用 METH_NOARGS 标志, 表明该函数调用应仅接受代表外部函数所属模块或外部方法所属类对应的实例对象 self 作为隐藏参数而不包含实际参数元组 args, 但

外部函数实现却尝试接收并解析实际参数. 这种标志位与外部函数实现类型的不一致将导致解释器未传递预期的参数结构, 引发栈错位或非法内存访问.

类似地, 在代码 7 的测例 2 中, 外部函数实现 wrong_flags_keywords 在外部函数实现的参数中包含关键字参数 kwargs 并调用 PyArg_ParseTupleAndKeywords 解析关键字参数, 但对应的调用约定位域却不包含 METH_KEYWORDS 标志, 导致解释器在构造外部函数参数时不会提供关键字参数字典. 这种调用约定与外部函数实现中解析行为的矛盾, 将导致外部函数访问未定义指针并可能引发运行时崩溃.

代码 7. 调用约定不匹配 (T2.2) 测例.

```
//测例 1: 使用 METH_NOARGS 标志但尝试接收参数
static PyObject* wrong_flags_noargs(PyObject *self, PyObject *args) {
  int value;
  //尝试解析参数
  if (!PyArg_ParseTuple(args, “i”, &value)) return NULL;
  return PyLong_FromLong(value * 2);
}
{“py_wrong_flags_noargs”, (PyCFunction)wrong_flags_noargs, METH_NOARGS, “docstring”}
//测例 2: 使用 METH_VARARGS 标志但尝试解析关键字参数
static PyObject* wrong_flags_keywords(PyObject *self, PyObject *args, PyObject *kwargs) {
  int a, b; static char *kwlist[] = {“a”, “b”, NULL};
  //尝试解析关键字参数
  if (!PyArg_ParseTupleAndKeywords(args, kwargs, “ii”, kwlist, &a, &b)) return NULL;
  return PyLong_FromLong(a + b);
}
{“py_wrong_flags_keywords”, (PyCFunction)wrong_flags_keywords, METH_VARARGS, “docstring”}
```

2.3 异常处理

C/C++缺乏统一的、语言级别的异常处理机制 (C 不具备异常处理机制, C++与 Python 的异常处理机制不完全兼容), 因此在 Python 的 C/C++外部扩展中, 异常处理依赖基于特定 Python/C API 的显式异常状态管理: 当外部函数出现逻辑或运行错误时, 必须通过调用 PyErr_SetString、PyErr_SetObject 等 Python/C API 函数设置异常状态, 且在抛出异常后须显式结束外部控制流返回 Python 侧, Python 的异常处理机制才能检查异常状态并进行处理.

如代码 8 所示, 在外部函数实现 overwrite_error 中, 当 fopen 函数调用失败时通过 Python/C API 函数 PyErr_SetString 设置异常. 由于 Python/C API 异常不会中断控制流, 因此未立即通过 return 返回会导致异常后非预期控制流 (E1.1). 外部函数继续执行并在 invalid_condition 为真时再次调用 PyErr_SetString 覆盖原异常, 导致异常重复抛出 (E1.2), 破坏异常链的完整性.

代码 8. 异常处理错误测例.

```
//测例: 非预期控制流中覆盖已有异常 (E1.1、E1.2)
static PyObject* overwrite_error(PyObject *self) {
  FILE *file = fopen(“nonexistent.txt”, “r”);
  if (!file) PyErr_SetString(PyExc_FileNotFoundError, “文件不存在”);
  //错误: 非预期控制流
  int invalid_condition = 1;
```

```
 if (invalid_condition) {
  PyErr_SetString(PyExc_ValueError, “参数无效”); //错误: 异常重复抛出
  return NULL;
 }
 fclose(file);
 Py_RETURN_NONE;
}
```

由于 C/C++可以使用返回值标识异常, 外部函数的返回 NULL 应与异常状态设置保持一致, 否则导致异常状态与返回值不一致 (E1.3) 漏洞. 如代码 9 所示, 测例 1 中外部函数实现 return_null_without_error 返回 NULL 却未调用抛出异常的 Python/C API, 可能引发运行时错误. 在代码 9 测例 2 的外部函数实现 set_error_but_return_object 中, 调用 PyErr_SetString 设置异常却返回非空对象 Py_NONE, 此时解释器误判函数成功执行, 导致异常状态丢失.

代码 9. 异常状态与返回值设置不一致测例.

```
//测例 1: 返回 NULL 但未设置异常 (E1.3)
static PyObject* return_null_without_error(PyObject *self) {
 return NULL; //错误: 未设置异常返回 NULL
}
//测例 2: 设置异常返回非 NULL 对象 (E1.3)
static PyObject* set_error_but_return_object(PyObject *self) {
 PyErr_SetString(PyExc_RuntimeError, “发生严重错误”);
 Py_RETURN_NONE; //错误: 设置异常后返回有效对象
}
```

2.4 并发控制

Python 解释器通过全局解释器锁 (global interpreter lock, GIL) 实现对内部状态的并发控制 (不考虑 Python 3.13 新引入允许关闭 GIL 的实验特性). 然而, 该机制导致即便在多线程环境下, 同一时刻也仅有一个线程可执行 Python 字节码. 当在 C/C++外部扩展中执行诸如文件读写、网络通信或系统调用等可能发生阻塞的操作时, 若未显式释放 GIL, 将导致当前线程在阻塞期间独占解释器资源, 从而阻碍其他线程的调度与执行, 形成 GIL 导致的临界区阻塞 (C1.1). 该漏洞会显著限制 Python 多线程程序在 I/O 密集型任务上的并发性能.

如代码 10 所示, 外部函数实现 blocking_io 在执行阻塞操作 sleep(seconds) 时未使用 Python/C API 宏 Py_BEGIN_ALLOW_THREADS 和 Py_END_ALLOW_THREADS 释放 GIL, 导致当前线程在阻塞期间仍持有 GIL, 阻碍其他线程运行, 进而引发临界区阻塞.

代码 10. GIL 导致的临界区阻塞 (C1.1) 测例.

```
//在持有 GIL 期间执行阻塞操作
static PyObject* blocking_io(PyObject *self, PyObject *args) {
 long seconds;
 if (!PyArg_ParseTuple(args, “l”, &seconds)) return NULL;
 //释放 GIL, 进入阻塞 I/O
 //Py_BEGIN_ALLOW_THREADS
 sleep(seconds); //阻塞操作
```

```
 //Py_END_ALLOW_THREADS
 Py_RETURN_NONE;
}
```

2.5 数值计算

Python 对于数值计算中的非法操作 (如除零) 会自动抛出异常, 以保障控制流的可预测性与错误的可追踪性. 而在 C/C++中, 整数除零、浮点除零、空指针解引用等运行时错误属于未定义行为, 不会自动抛出异常, 执行结果取决于编译器、平台甚至硬件指令集, 可能导致程序崩溃或产生不符合预期的返回值. 当外部函数的数值型参数参与数值计算时, 接口层整数除零操作 (N1.1) 等漏洞也可能存在, 如代码 11 所示.

代码 11. 接口层整数除零操作 (N1.1) 测例.

```
//a 和 b 是来自 Python 侧的整型参数, a 除以 b, 未检查 b 是否为零
static PyObject* do_division(PyObject *self, PyObject *args) {
 int a, b;
 if (!PyArg_ParseTuple(args, “ii”, &a, &b)) return NULL;
 double result = a/b;
 return PyFloat_FromDouble(result);
}
```

3 漏洞基准套件构建

为了系统地评估 Python-C/C++互操作程序的漏洞检查工具, 本文构建了一套覆盖性强、结构清晰的漏洞基准套件. 该套件由两大部分组成, 分别是基于上述漏洞模式分类构建的漏洞基准测例, 以及面向真实工程实践的来自 Python 软件包库 PyPI 的高安装库中存在漏洞的互操作程序.

作为漏洞基准套件的两部分, 模式驱动的漏洞测例子集与软件供应链真实漏洞子集两者相互补充并承担不同的评估职责: 模式驱动的漏洞测例子集主要用于准确评估工具对完整漏洞模式的检出能力 (漏报率); 软件供应链真实漏洞子集则侧重反映工具在复杂工程项目中的实际表现, 包括误报率和可扩展性. 其中软件供应链真实漏洞子集的构建是一个同步进行工具评估和漏洞标注的过程, 考察复杂软件环境下的误报率, 收集软件供应链中的真实漏洞, 不作为漏报率的评估基准.

该漏洞基准套件具有覆盖全面、结构清晰、易于扩展等优点. 不仅涵盖真实 Python 软件包中广泛使用的 Python-C/C++互操作场景, 体现出良好的代表性与多样性; 同时依托系统的漏洞模式分类, 为每类典型漏洞设计实现可控、可复现的标准化测例, 支持静态与动态分析工具的统一评估. 该漏洞基准套件具备统一接口、版本兼容、源汇点标注、支持先进安全性分析技术等优势, 为 Python-C/C++互操作程序的安全性分析研究提供高质量、可持续演进的评估测试平台. 漏洞基准套件构建的总体流程如后文图 4 所示.

3.1 模式驱动的漏洞测例子集构建

为了提高漏洞模式的代表性, 构建表 1 所示漏洞测例子集时混合了以下 4 个来源: (1) Python 3.13 版本 Python/C API 官方文档; (2) 已有的 Python-C/C++跨语言互操作安全性分析研究和工具[9−17]; (3) 已有的其他语言互操作安全性研究[8,18]; (4) CVE 库及 Python-C/C++互操作软件开源漏洞列表. 对来源 (1)–(3) 中的文档和文献进行通读, 对来源 (4) 进行以表 1 细粒度漏洞模式列为关键字的搜索和筛选, 实现了其中所有示例代码和漏洞, 包括文档中所有提及可能存在隐患的情形. 随后对复现得到的大量候选样例进行人工审查, 去除重复及与漏洞无关的业务逻辑, 基于第 2 节定义的漏洞模式手工提炼与构造每个模式对应的最小可复现测试单元.

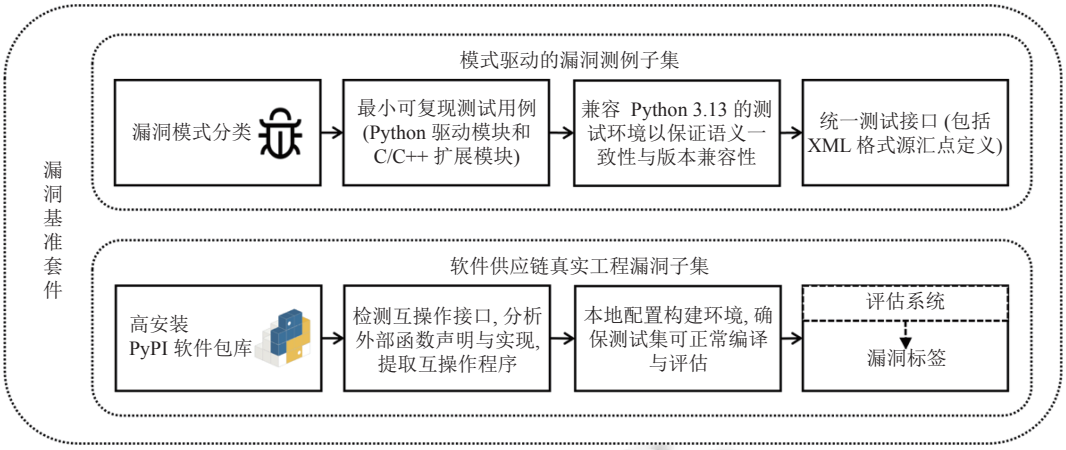

图 4 漏洞基准套件构建总体流程图

针对第 2 节中定义的每一种细粒度漏洞模式, 构造对应的最小可复现测试单元, 统一由一个 Python 测试脚本与一个配套的 C 扩展模块构成, 形成“模式-测例”一一映射关系. 所有测例均提供一致的测试接口, 并通过统一的测试脚本调用目标测例, 以支持自动化批量测试与结果收集. 事实上, 我们在漏洞测例子集中构建的测例要远多于在第 2 节中展示的测例, 即一个细粒度漏洞模式会对应多个具体漏洞情形. 以代码 5 的整数溢出漏洞为例, 接口层运算溢出 (T1.2) 除了来自加法外, 还可能来自其他运算如乘法、移位等.

为了保证在不同工具和平台下的可执行性与稳定性, 所有测试用例均基于 Python 3.13 环境进行构建, 确保语义一致性与版本兼容性. 同时, 考虑到 Mopsa 与 PolyCruise 等程序分析工具的构建需求, 测试模块在构建时内嵌相关辅助标注与兼容支持, 并配套提供以 XML 格式编写的 criterions.xml 源点/汇点定义文件, 标注输入变量与潜在传播路径, 以满足静态污点分析与动态信息追踪的需要.

漏洞基准测例按照漏洞模式编号组织目录结构, 并采用模块化设计以支持后续对测试集的动态增补与工具适配的迭代更新, 具备良好的可维护性与可扩展性.

3.2 软件供应链真实工程漏洞子集构建

在基于 Python/C API 构建 C/C++扩展模块的过程中 (参考图 2 代码), 跨语言程序通常包含#include<Python.h>这一关键性的预处理指令. 该头文件作为 Python-C/C++互操作外部接口 Python/C API 的核心入口, 集中声明了所有与 Python 解释器交互所必需的函数、数据类型、宏定义等. 因此, 本文以是否包含#include<Python.h> 作为一种项目级的静态特征, 用于初步判定项目中是否存在 Python-C/C++互操作行为, 从而作为后续筛选的依据.

具体的项目筛选流程如图 4 所示. 首先, 我们参考第三方统计平台提供的安装数据, 获取了 PyPI 平台上安装量排名前 100 的软件包. 这一选择策略的初衷在于确保所筛选项目具有较高的实际使用频率和社区影响力, 能够代表主流工程实践中 Python-C/C++互操作程序的开发现状.

随后, 我们编写自动化脚本, 对上述项目源码进行递归遍历, 检测各源码文件是否包含 Python.h 头文件. 该检测作为一个快速过滤步骤, 能够有效排除完全未使用 C/C++扩展机制的项目. 然而, 该策略并非充分条件, 因为某些文件虽包含该头文件, 但可能仅用于文档示例、测试代码或注释, 尚不足以确认实际互操作行为. 为进一步提升识别准确性, 本文实现了一个基于 Clang 的轻量级静态分析器, 用于深入分析源码中是否存在实际的 Python-C/C++互操作结构. 该分析器能够从源码中识别出声明外部函数的 Python/C API 结构体 PyMethodDef (图 2 第 14–19 行), 并进一步提取其定义的 Python 侧调用函数名及其对应的 C/C++外部函数实现.

通过以上的静态结构分析与筛选, 我们最终得到 16 个安装量排名前列且实际包含 Python/C API 调用的 PyPI 软件包, 这些项目具有代表性且在 Python 软件供应链中使用广泛. 除了用于编写 C/C++扩展外, Python/C API 还可以用于将 Python 运行时嵌入 C/C++程序以实现 Python-C/C++调用, 这种用法在 Python 软件供应链中非常少见.

安装量最高的 16 个 PyPI 互操作包及上述漏洞测例子集都是 Python 调用 C/C++的情形, 其中允许通过 PyObject_Call 等 Python/C API 函数由 C/C++侧回调 Python 对象.

进一步地, 为确保后续静态分析工具能够准确解析各项目源码, 我们对每一个候选项目在本地环境中进行依赖配置并测试构建流程. 该步骤对于静态分析工具的成功构建和分析准确性至关重要. 一方面, 先进分析工具往往依赖项目构建过程中生成的中间文件, 以及系统头文件路径、预处理宏展开以及符号链接信息等, 只有在项目被完整构建后, 这些上下文信息才可被工具有效解析. 若项目未能成功构建, 将严重阻碍静态分析完整的语法语义信息, 导致分析不准确或失败.

不同项目在构建系统与依赖管理方式上的差异也对分析环境配置提出更高要求. 例如, numpy、scipy、pandas 等高性能计算项目已普遍引入 mamba 等现代化依赖解析器, 以适配复杂的编译依赖与平台特性; 相比之下, 其他中小型项目仍采用 setuptools 或 distutils 等传统构建方式. 前者往往依赖 CMake、Meson、cythonize 等二级构建系统, 并可能在构建阶段动态生成部分 C/C++文件和头文件, 而后者则相对结构简单. 这些差异直接影响构建流程的完整性和构建产物的可访问性, 因此也对静态分析工具的兼容性与适配能力提出了挑战.

为了保证对不同项目的构建链的统一调度与评估复现, 我们在本地实现了一个统一驱动脚本, 作为自动化构建与评估的入口. 该驱动脚本调用不同项目的原生构建脚本, 根据评估任务相应地设置环境变量并执行分析命令, 同时调用针对被测项目的配置文件. 确保每个项目的源码在分析前都已成功通过其原生构建系统处理, 最大程度还原其语法结构和语义信息, 以保障分析评估阶段的完整性与准确性. 不同工具的主要分析命令、环境变量和配置文件如表 4 所示.

表 4　不同工具的主要分析命令、环境变量和配置文件

工具	分析命令	环境变量	配置文件
CPyChecker	python setup.py build_ext --inplace	CC="/path-to/gcc-python-plugin/ gcc-with-cpychecker"	setup.py
PyRefcon	scan-build -disable-checker core -enable-checker alpha.core.PythonReferenceCountingChecker python setup.py build	—	setup.py
PyCType	python evaluate.py target_dir	—	ppconfig.yml
PolyCruise	python setup-sda.py build sda -file $bc -criterion <your-path>/criterion.xml difaEngine &python -m pyinspect -C <criterion.xml> -t <case>	CC="clang -emit-llvm -Xclang -load -Xclang llvmSDIpass.so" CXX="clang -emit-llvm -Xclang -load -Xclang llvmSDIpass.so" LDSHARED="clang -flto -pthread -shared -lDynAnalyze"	setup-instrm.py, setup-sda.py, criterion.xml
Mopsa	python3 setup.py build --force	CC="/path-to/mopsa-wrappers/ x86_64-linux-gnu-gcc" LDSHARED="/path-to/mopsa-wrappers/ x86_64-linux-gnu-gcc -shared"	setup.py

最后, 作为漏洞基准套件的一部分, 由 PyPI 库得到的高安装量软件包在构建成功并分离出互操作程序后, 由先进安全分析工具进行评估 (评估过程和结果见第 4 节), 发现其中尚未报告和修复的漏洞, 并根据第 2 节中的漏洞模式分类进行漏洞标注后加入漏洞基准套件.

3.3　其他互操作安全漏洞基准套件与数据集

已有的互操作安全研究大多没有提供系统而全面的漏洞基准套件, 仅针对漏洞模式给出示例代码, 或根据常见漏洞来源选取部分测试项目. Tan 等人[8]分析总结了 Java-C/C++互操作的漏洞模式分类, 并在 Java 开发套件 (Java development kit, JDK) 中使用单语言工具进行了漏洞检查, 反映出单语言工具检查互操作漏洞时极高的误报率. Hu 等人[9,10]分析讨论了 Java-C/C++互操作的漏洞模式在 Python-C/C++互操作程序中的表现形式, 设计实现了基于模式匹配的轻量漏洞检查方法, 其误报率也较高. 这两个工作反映出 Python-C/C++互操作的漏洞检查需要定制的、精细的程序分析工具. 同时所有已知的数据集都无法评估漏报率, 而只能评估特定检查器针对少数漏洞模

式的误报率. PolyCruise[14]和 Mopsa[16]针对其分析的特定漏洞模式分别提供了测试用例, 以评估方法在特定语言特性上的局限, 它们的数据集仅支持少数漏洞模式 (见第 4.3 节), 并且已经在归纳后包含在本文的漏洞模式分类中.

4 实验评估

4.1 C/C++单语言漏洞检测工具分析互操作程序的局限

首先说明 C/C++单语言漏洞检测工具在分析 Python-C/C++互操作程序漏洞上的局限. 考虑以下 6 个代表性的分析工具: Clang Static Analyzer (CSA) 是基于 LLVM/Clang 编译器前端的符号执行框架; Cppcheck 采用规则驱动的轻量数据流分析; Infer 基于抽象解释与路径推理, 分析内存与资源管理问题; Frama-C[20]基于抽象解释, 支持函数不变式求解与可达性分析; Splint 基于注释约定与类型检查实现接口一致性检查; Flawfinder 是基于规则匹配的快速安全审计工具, 检测已知的危险函数模式.

从定性角度看 (表 5 中 C/C++分析工具可检测性列, △表示部分可检测), 现有 C/C++分析器的检测能力主要聚焦于算术错误 (如整数溢出、除零)、内存错误 (如空指针解引用、资源泄漏、越界访问) 等传统 C/C++漏洞. 它们通常通过数据流分析检查变量的取值范围与生命周期, 对 malloc/free 等标准内存操作具有较高的检测精度. 然而, 这类工具的分析止步于 C/C++语言及 C 系统调用这一边界, 缺乏对跨语言生命周期及异常传播机制的建模和分析, 因而在 Python-C/C++互操作场景中存在显著的检测盲区.

表 5 C/C++单语言漏洞检测工具分析 Python-C/C++互操作程序漏洞的表现

细粒度漏洞模式			C/C++分析工具可检测性	测例通过率			
编号	名称	测例数量		CSA	Cppcheck	Infer	Flawfinder
M1.1	内存泄漏	4	×	—	—	—	—
M1.2	悬空引用或对象过早释放	5	×	—	—	—	—
M2.1	构造析构不匹配	9	△	—	—	—	—
M2.2	非法内存分配与释放	6	△	—	—	—	—
M3.1	缓冲区越界访问	6	△	4/6	—	—	4/6
T1.1	跨语言传参溢出	3	△	—	—	—	—
T1.2	接口层运算溢出	6	△	—	1/6	—	—
T2.1	格式化字符串不匹配	12	×	—	—	—	—
T2.2	调用约定不匹配	9	×	—	—	—	—
E1.1	异常后非预期控制流	3	×	—	—	—	—
E1.2	异常重复抛出	2	×	—	—	—	—
E1.3	异常状态与返回值不一致	3	×	—	—	—	—
E2.1	未检查接口函数返回值	5	△	—	—	—	—
C1.1	GIL引发的临界区阻塞	4	×	—	—	—	—
N1.1	接口层整数除零操作	5	△	1/5	1/5	—	—

定量实验在前文介绍的模式驱动的漏洞测例子集上进行, 4 个能够完成互操作程序分析流程的工具的评估结果如表 5 所示, 结果基本验证了上述的定性分析. C/C++单语言漏洞检测工具在算术错误与局部内存错误上表现出一定的检测能力, 但在涉及宿主语言对象状态和跨语言约束的测例中普遍失效.

4.2 Python-C/C++互操作程序漏洞检查工具的选择

为了全面评估已有先进工具检查 Python-C/C++互操作程序漏洞的能力, 我们系统地梳理 2010 年至今相关公开文献与工具, 聚焦 Python 外部扩展的安全性分析与建模研究. 依据方法先进性、漏洞覆盖广度、工具可获取性 (开源/闭源)、资料完整度这 4 项标准, 遴选得到表 6 所示的 8 款代表性先进工具. 这些工具设计实现了不同的程序分析技术[21], 包括数据流分析、符号执行、抽象解释、仿射分析、动态污点追踪、跨语言中间表示等, 能够识别 Python-C/C++互操作程序的引用计数错误、类型一致性错误、缓冲区溢出、除零错误等安全漏洞.

表 6 评估的 Python-C/C++互操作程序先进漏洞检查工具

工具	分析方法	能够分析的问题	支持的漏洞模式	是否开源	提出时间
PolyCruise	跨语言中间表示+符号依赖分析+动态信息流追踪	缓冲区越界访问、整数溢出、除零	M3.1/T1.2/N1.1	是	2022
PolyFuzz	动态模糊测试	内存泄漏、悬空引用、越界内存访问	M1.1/M1.2/M3.1	是	2023
PyRefcon	符号执行	内存泄漏、释放后使用	M1.1/M1.2	是	2023
Mopsa	抽象解释	缓冲区越界访问、整数溢出、格式化字符串不匹配、异常处理错误	M3.1/T1.1/T1.2/T2.1/E1.3	是	2021
CPyChecker	基于GCC的数据流分析	引用计数错误(泄漏/过早释放)、格式字符串不匹配、调用约定不匹配、未检查接口函数返回值	M1.1/M1.2/T2.1/T2.2/E2.1	是	2012
PyCType	静态类型推断	调用约定不匹配	T2.2	是	2021
Pungi	仿射变换与分析	内存泄漏、对象过早释放	M1.1/M1.2	否	2014
RID	不一致路径对检查	内存泄漏、对象过早释放	M1.1/M1.2	否	2016

4.2.1 引用计数分析工具

基于引用计数的垃圾收集是 Python 和 C/C++在内存模型上核心的特性差异之一, 导致的互操作程序引用计数错误 (M1) 被研究人员重点关注. 基于 Python/C API 增减引用计数容易引发内存泄漏、悬空引用、重复释放等内存安全问题. 一系列研究工作围绕该漏洞设计实现对应的检查工具.

● Pungi[11]: 采用基于静态单赋值形式的代码转换与仿射分析技术, 将 Python-C/C++互操作程序转换为仿射程序, 并对引用计数行为进行建模与检测. Pungi 能够检查内存泄漏 (M1.1)、悬空引用 (M1.2) 等问题, 主要关注分析精度与路径敏感的引用计数分析, 其并未开源.

● RID[12]: 核心思想为不一致路径对检查, 通过基于 LLVM IR 的符号执行技术枚举路径, 提取路径摘要, 检查不同路径在相同条件下引用计数的变化是否一致. RID 可以分析路径敏感的引用计数内存泄漏 (M1.1)、悬空引用及对象过早释放 (M1.2) 漏洞, 适用于函数内复杂控制流结构下的引用计数错误分析. RID 同样未开源.

● PyRefcon[13]: 提出基于生命周期建模的检测方法, 构建 PyObject (互操作程序中所有 Python 对象的基类) 的状态转换模型, 基于符号执行引擎对对象状态进行路径敏感追踪. PyRefcon 对 384 个 Python/C API 接口的引用计数行为进行建模, 能够检查包括内存泄漏 (M1.1) 与悬空引用 (M1.2) 在内的引用计数安全漏洞. PyRefcon 已开源, 具备良好的理论价值与实用性.

Pungi、RID 和 PyRefcon 分别和 CPyChecker (见第 4.2.2 节) 的引用计数错误检查进行了对比评估, 其中 PyRefcon 作为最新的工具, 宣称具有最好的效果.

4.2.2 类型与语义建模工具

相关研究将单语言的类型推断、抽象解释等程序分析技术扩展到 Python-C/C++互操作场景. 通过跨语言类型转换的语义建模和跨语言抽象域的设计, 提出通用的 Python-C/C++互操作静态分析框架, 具有较好的扩展性, 能够支持类型相关跨语言安全漏洞的检查.

● Mopsa[16]: 一个支持跨语言互操作抽象解释的静态分析平台. 针对 Python-C/C++互操作程序, 构建共享抽象域的联合抽象解释框架. 分析过程包括互操作语义建模、边界函数抽象、数值与指针状态联合建模, 以及边界条件的不变式生成. Mopsa 能够检查整数溢出 (T1.1/T1.2)、类型一致性错误中的调用约定不匹配 (T2.2)、异常处理错误中的异常状态与返回值不一致 (E1.3) 等漏洞. Mopsa 支持模块化配置与扩展, 当前已开源.

● PyCType[17]: 通过构建跨语言的静态类型推断系统, 基于 Python/C API 的类型转换语义对 Python 的 C/C++外部函数进行类型推断, 推断结果可以增强 Python 单语言的静态类型推断工具. 作为类型推断的副产品, PyCType 可以检查外部函数的类型一致性漏洞 (T2.1/T2.2), 但作为类型推断系统的扩展应用示例, PyCType 只支持无参外部函数标志 (表 3 中 METH_NOARGS) 的检查, 该工具已开源.

4.2.3 数据流分析工具

● CPyChecker: 基于 GCC 的静态数据流分析工具, 用于检测 Python 的 C 扩展模块中常见的 Python/C API 误用问题. CPyChecker 能够检查引用计数错误 (M1.1/M1.2)、异常处理错误 (E1.2/E1.3)、格式化字符串不匹配 (T2.1) 等安全漏洞. CPyChecker 在 GCC 中 C 程序静态分析的基础上, 建模并检查 Python/C API 的使用规范 (较早版本), 可以集成于 GCC 编译工具链的构建流程进行静态审计, 支持主流 GCC 版本且已开源.

4.2.4 基于跨语言中间表示的动态分析与测试工具

动态分析与测试可以弥补静态分析在动态特性上的误报和效率问题, 相关研究基于跨语言的中间表示将动态分析与测试扩展到支持 Python-C/C++互操作程序.

● PolyCruise[14]: 设计了一种跨语言的统一中间表示 LISR (language-independent symbolic representation), 通过轻量级静态分析提取不同语言模块间的符号依赖关系, 进而实现动态的选择性插桩与信息流分析. PolyCruise 在运行时使用动态信息流图 (dynamic information flow graph, DIFG) 追踪变量依赖关系, 以检查诸如缓冲区溢出 (M3.1)、整数溢出 (T1.1/T1.2)、除零错误 (N1.1) 等安全漏洞. 该工具已开源, 是目前少有的支持动态信息流计算的跨语言漏洞检测框架之一.

● PolyFuzz[15]: 面向多语言系统设计的一种灰盒模糊测试框架, 通过统一中间表示 SAIR (static abstract intermediate representation) 简化语言间语义差异, 结合支配树分析、种子敏感性建模和路径感知输入生成等技术提高模糊测试效率. 该方法能够检查内存泄漏 (M1.1)、空指针解引用 (M1.2)、缓冲区越界访问 (M3.1) 等安全漏洞, 目前已开源.

4.3 完备性分析

完备性是指分析能力覆盖目标范围内所有可能的安全问题, 具备完备性意味着检查器没有漏报. 完备性评估衡量方法或系统分析能力的边界. 为系统评估分析已有先进工具在 Python-C/C++互操作程序漏洞检测时的完备性, 我们提出以下两个研究问题.

● RQ1: 已有漏洞检测工具能够覆盖哪些 Python-C/C++互操作程序漏洞模式?

● RQ2: 已有漏洞检测工具在不同漏洞模式上的漏报率如何? 产生漏报的主要原因是什么?

基于第 3 节提出的漏洞基准套件, 我们系统地评估第 4.2 节选择的工具在第 2 节提出的漏洞模式上的适用性与能力边界. 我们选取 Mopsa、CPyChecker、PyRefcon、PyCType 和 PolyCruise 对基准测试程序中每种漏洞模式对应的若干测例逐一检查并统计其分析能力的覆盖情况. RID 与 Pungi 未开源, PolyFuzz 依赖输入驱动模糊测试, 因此被排除在外. 余下工具在不同漏洞模式下的表现汇总于表 7.

表 7 Python-C/C++互操作漏洞检查工具在基准测例上的表现

细粒度漏洞模式			分析工具测例通过率				
编号	名称	测例数量	CPyChecker	PyRefcon	PyCType	PolyCruise	Mopsa
M1.1	内存泄漏	4	0/4	*2/4	—	—	—
M1.2	悬空引用或对象过早释放	5	0/5	*4/5	—	—	—
M2.1	构造析构不匹配	9	—	—	—	—	—
M2.2	非法内存分配与释放	6	—	—	—	—	—
M3.1	缓冲区越界访问	6	—	—	—	*4/6	2/6
T1.1	跨语言传参溢出	3	—	—	—	—	*1/3
T1.2	接口层运算溢出	6	—	—	—	2/6	*2/6
T2.1	格式化字符串不匹配	12	*6/12	—	—	—	4/12
T2.2	调用约定不匹配	9	3/9	—	*4/9	—	—
E1.1	异常后非预期控制流	3	—	—	—	—	—
E1.2	异常重复抛出	2	—	—	—	—	—
E1.3	异常状态与返回值不一致	3	—	—	—	—	0/3
E2.1	未检查接口函数返回值	5	0/5	—	—	—	—

表 7 Python-C/C++互操作漏洞检查工具在基准测例上的表现 (续)

细粒度漏洞模式			分析工具测例通过率				
编号	名称	测例数量	CPyChecker	PyRefcon	PyCType	PolyCruise	Mopsa
C1.1	GIL引发的临界区阻塞	4	—	—	—	—	—
N1.1	接口层整数除零操作	5	—	—	—	*3/5	—

注: *为表现最好的工具

表 7 中画横线的单元格表示该工具不支持对应漏洞模式的检查, 因此未通过任何测例. 其余单元格内分母表示漏洞模式的测例总数, 分子表示其中正确检出的测例数量.

● Answer 1. 已有 Python-C/C++互操作程序漏洞检查工具对漏洞模式的覆盖能力存在明显不足, 仅能支持 15 种漏洞模式中的 8 种, 此外还有 2 种漏洞模式尽管有检查工具宣称覆盖但未检出有效结果.

如表 7 所示, 在漏洞基准测例包含的 15 种常见漏洞模式中, 7 种漏洞模式 (占比 46.7%) 在所有工具上均未报告有效检出. 例如, 对于内存管理错误 (M2), 其两个漏洞模式构造析构不匹配 (M2.1) 和非法内存分配与释放 (M2.2) 均未有检查工具提供分析能力. 此外, 异常后非预期控制流 (E1.1), 异常重复抛出 (E1.2), GIL 引发的临界区阻塞 (C1.1) 等漏洞类型也没有工具可以支持. 异常状态与返回值不一致 (E1.3), 未检查接口函数返回值 (E2.1) 这两个漏洞模式尽管有工具宣称覆盖, 但均未检出有效报错.

● Answer 2. 在能够分析的 8 种漏洞模式中, 在对每个漏洞都选用表现最好的工具的情况下, 整体漏报率仍然高达 52% (26/50).

即使在有工具报告检出的 8 种漏洞模式中, 覆盖表现亦呈现显著局限性. 以在每种漏洞上检查能力表现最好的工具为例, PyRefcon 对内存泄漏 (M1.1) 漏报 50%, 对悬空引用或对象过早释放 (M1.2) 漏报 20%. PolyCruise 对缓冲区越界访问 (M3.1) 漏报 33%. Mopsa 对跨语言传参溢出 (T1.1) 漏报 67%. PolyCruise 和 Mopsa 对接口层运算溢出 (T1.2) 均漏报 67%, CPyChecker 对格式化字符串不匹配 (T2.1) 漏报 50%, PyCType 对调用约定不匹配 (T2.2) 漏报 55%, PolyCruise 对接口层整数除零操作 (N1.1) 漏报 40%. 即使对于每个能够检查的漏洞都使用表现最好的工具 (表 7 中*标记), 整体的漏报率依然高达 52%, 即综合运用 5 个工具在支持的 8 个漏洞模式的 50 个测例上仅通过了其中的 26 个.

建模粒度较粗或分析策略局限是漏报的主要原因. PyCType 为类型推断工具, 对于调用约定不匹配的类型一致性错误, 仅能检测无参外部函数的情形, 其与 CPyChecker 对参数使用行为的建模能够一定程度形成互补. PyCType 仅检查参数使用, 不检查参数声明与调用约定标志是否匹配, 是其无法通过部分测例的主要原因. 对于传参溢出和运算溢出的测例, Mopsa 和 PolyCruise 都表现出一定的局限性. Mopsa 能捕获部分整数乘法和符号转换过程中的溢出, 但对整数加法、位移以及浮点溢出均不支持, 导致多个测例漏报; PolyCruise 仅支持加法和乘法运算, 忽略其他算术操作, 同时仅分析 C/C++侧运算过程, 对于 Python 侧传参所引发的溢出则无法感知. 此外, PolyCruise 在除零错误检查时也不支持浮点运算和复杂表达式求值的情形. 对于引用计数错误 (M1), PyRefcon 在检测释放后使用 (use-after-free) 等错误方面表现优异, 但对于返回借引用却未显式增加引用计数的情形则存在无法有效分析的测例. 同时, 由于依赖于显式 Py_INCREF、Py_DECREF 调用作为分析起点, PyRefcon 对复杂引用路径的支持仍有不足.

4.4 可靠性分析

一个分析是可靠的意味着其报告的所有结果都是真实存在的漏洞或性质, 即漏洞检查没有误报. 可靠性直接关系到系统的可用性, 能够避免对大规模软件包库产生过多误报而带来高昂的人工检查开销.

● RQ3: 已有漏洞检查工具在真实软件供应链分析中的误报率如何?

我们选取了 PyPI 平台中安装量排名前 16 的包含 Python-C/C++互操作的项目作为分析对象. 由于 PolyCruise[14] 和 Mopsa[16]可扩展性的问题 (详见 RQ4), 我们复用了其论文中部分项目 (python-levenshtein、pyahocorasick 和 numpy) 的数据, 这部分数据仅用于综合评估分析存在可扩展性问题的方法和工具, 其结果不计入本文新发现的

21 个真实漏洞.

● Answer 3. 对于软件供应链中的大型 Python-C/C++互操作项目, CPyChecker 漏洞误报率高达 99.9%; PyRefcon 漏洞误报率高达 91.66%; PyCType 漏洞误报率 5.3%. 整体上, 被测先进工具的漏洞误报率高达 97.17%.

如表 8 所示, CPyChecker 报告格式化字符串不匹配 (T2.1) 漏洞共计 168 处, 仅 1 处为真阳性, 误报率 99.4%; 报告调用约定不匹配 (T2.2) 漏洞共 530 处, 未检出任何真实漏洞, 显示其在软件供应链分析中对类型一致性错误 (T2) 缺乏有效判别能力. PyRefcon 在引用计数错误 (M1) 的检查中报告内存泄漏 (M1.1) 漏洞 2 处, 确认 0 处为真阳性, 误报率 100%; 报告悬空引用或对象过早释放 (M1.2) 漏洞 22 处, 确认 2 处为真阳性, 误报率 91%. 虽然误报率相对 CPyChecker 较好, 但整体误报率仍然高达 91.66%.

表 8 CPyChecker 和 PyRefcon 软件供应链分析的真阳性 (*TP*) 和假阳性 (*FP*) 结果

评估项目	日安装量 (万)	CPyChecker				PyRefcon			
		编译通过	警告	*TP*	*FP*	编译通过	警告	*TP*	*FP*
numpy	1 799	no	0	0	0	yes	1	0	1 (M1.2)
cffi	1 527	yes	40	0	40 (T2.1)	yes	5	0	4 (M1.2)+1 (M1.1)
pandas	1 479	no	0	0	0	yes	0	0	0
protobuf	1 367	yes	4	0	3 (T2.2)+1 (T2.1)	yes	1	0	1 (M1.2)
multidict	904	yes	0	0	0	yes	0	0	0
wrapt	866	yes	6	0	2 (T2.2)+4 (T2.1)	yes	1	0	1 (M1.2)
greenlet	756	no	0	0	0	yes	2	0	2 (M1.2)
psutil	753	no	0	0	0	yes	1	0	1 (M1.2)
grpcio	736	yes	263	0	263 (T2.2)	yes	2	0	2 (M1.2)
pillow	720	yes	133	0	119 (T2.1)+14 (T2.2)	yes	1	0	1 (M1.2)
scipy	707	no	0	0	0	yes	1	0	1 (M1.2)
grpcio-tools	694	yes	0	0	0	yes	1	0	1 (M1.2)
lxml	674	yes	227	0	227 (T2.2)	yes	2	1 (M1.2)	1 (M1.1)
msgpack	576	yes	19	0	19 (T2.2)	yes	2	1 (M1.2)	1 (M1.2)
regex	547	yes	3	0	1 (T2.1)+2 (T2.2)	yes	1	0	1 (M1.2)
coverage	486	yes	1	1 (T2.1)	0	yes	1	0	1 (M1.2)
python-levenshtein	35	yes	0	0	0	yes	1	0	1 (M1.2)
pyahocorasick	4	yes	2	0	2 (T2.1)	yes	1	0	1 (M1.2)
总计		13/18	698	1	697	16/16	24	2	22

如表 9 所示, PyCType 通过保守可靠的类型语义建模表现出较优的误报率. 在测试的 18 个 PyPI 包中报告调用约定不匹配 (T2.2) 漏洞共 19 处, 其中 18 处为真阳性, 仅有 1 处为假阳性, 整体误报率仅 5.3%. 由于较差的可扩展性, PolyCruise[14]和 Mopsa[16]无法在 16 个高下载量的主流软件包上进行有效评估, 我们复用其论文提供的评估数据 (表 9 中*标记): PolyCruise 在 numpy 项目中发现了 3 个实际漏洞, 整体准确率 88.2%; Mopsa 在 C/C++和 Python 侧的测试准确率分别为 93.9% 和 97.2%. 综合表 8、表 9, 共计得到 741 个警报 (不含复用论文数据), 其中 21 个为真实漏洞, 整体漏洞误报率高达 97.17%.

表 9 PyCType、PolyCruise、Mopsa 软件供应链分析的真阳性 (*TP*) 和假阳性 (*FP*) 结果

评估项目	日安装量 (万)	PyCType				PolyCruise		Mopsa		
		可分析文件	警告	*TP*	*FP*	成功触发	*TP*/警告	成功触发	测试准确率 (%)	
									C++	Python
numpy	1 799	732/1 018	2	2 (T2.2)	0	yes	(1 (T1.2)+2 (M3.1))/3	0	0	0
cffi	1 527	15/25	1	1 (T2.2)	0	no	0	0	0	0
pandas	1 479	6/12	0	0	0	no	0	0	0	0
protobuf	1 367	2/73	0	0	0	no	0	0	0	0
multidict	904	0/1	0	0	0	no	0	0	0	0
wrapt	866	0/1	0	0	0	no	0	0	0	0

表 9 PyCType、PolyCruise、Mopsa 软件供应链分析的真阳性 (*TP*) 和假阳性 (*FP*) 结果 (续)

评估项目	日安装量 (万)	PyCType				PolyCruise		Mopsa		
		可分析文件	警告	*TP*	*FP*	成功触发	*TP*/警告	成功触发	测试准确率 (%)	
									C++	Python
greenlet	756	1/1	0	0	0	no	0	0	0	0
psutil	753	7/61	0	0	0	no	0	0	0	0
grpcio	736	45/520	0	0	0	no	0	0	0	0
pillow	720	80/86	5	5 (T2.2)	0	no	0	0	0	0
scipy	707	265/452	0	0	0	no	0	0	0	0
grpcio-tools	694	0/12	0	0	0	no	0	0	0	0
lxml	674	0/6	0	0	0	no	0	0	0	0
msgpack	576	0/1	0	0	0	no	0	0	0	0
regex	547	2/2	6	5 (T2.2)	1 (T2.2)	no	0	0	0	0
coverage	486	4/4	5	5 (T2.2)	0	no	0	0	0	0
python-levenshtein	35	4/19	0	0	0	no	0	17/17	93.1*	98.0*
pyahocorasick	4	3/3	0	0	0	no	0	46/92	79.9*	93.2*
总计		50.8%	19	18	1	—	88.2%*	—	93.9*	97.2*

注: *为文献[14,16]提供的评估数据

综上, 我们在 PyPI 软件包库安装量排名前 16 的含有 Python-C/C++互操作的软件包中新发现 3 种共 21 个漏洞实例. 这些漏洞分布于 PyPI 安装量最高的核心包中, 日安装量在 486 万–1 799 万之间, 漏洞的影响用户数巨大. 此外, 这类软件供应链核心包一旦存在漏洞, 会通过依赖传播影响大量下游项目, 其安全隐患的影响范围不可忽视.

● RQ4: 先进工具在 PyPI 软件供应链中进行互操作安全性分析产生大量误报的主要原因是什么?

对于 CPyChecker, 其误报主要源于版本支持落后, 无法正确识别 Python 新版本中引入的语言特性和对应的 Python/C API. 例如, 从表 8 中的 grpcio、lxml 等项目可以看出, CPyChecker 在调用约定不匹配 (T2.2) 漏洞上产生了大量误报. lxml 中的 227 条误报、grpcio 中的 263 条误报全部源于无法识别 METH_FASTCALL | METH_KEYWORDS 这一 Python 3.7 引入的外部函数调用约定标志. 虽然对应的外部函数确实采用了正确的四参数签名, 但由于 CPyChecker 不支持该调用约定标志, 故错误地认为参数数量不符, 从而发出误报.

此外, CPyChecker 还存在一些分析算法设计上的问题. 例如, METH_NOARGS 调用约定标志规定外部函数不从 Python 侧接收任何参数 (self 对象除外), 但是由于 CPyChecker 仅考虑外部函数类型 PyCFunction 可接收两个参数, 因此其对所有采用 METH_NOARGS 声明的外部函数都会产生错误警告. 这类误报反映了工具在外部函数类型建模上的缺陷.

PyRefcon 的误报主要源于其内建的引用计数状态机在某些语义推理上的不足. PyRefcon 主要通过静态追踪 Py_INCREF 和 Py_DECREF 的调用以推断对象生命周期, 但当进入 Py_DECREF 内部展开进一步分析时, 会沿着调用链进一步跟踪至 Py_Dealloc. 此时在工具视角下, 对象被认为已经释放, 再次调用 Py_Dealloc 即会被视为释放后使用而触发误报. 这种问题实质上并非真实错误, 而是由于工具在对象生命周期建模时未正确处理 Py_DECREF 到 Py_Dealloc 的关系, 导致状态转换超前, 产生误报.

● Answer 4. 误报主要来自: 1) 版本迭代引入未建模的 Python/C API; 2) 互操作的复杂性导致分析算法本身不可靠; 3) 语义建模未考虑 Python/C API 内部实现.

4.5 可扩展性分析

可扩展性通常指当规模、负载、功能需求增长时, 系统或方法仍能高效、稳定地运行, 并且能够方便地扩展功能或容量. 软件供应链往往包含海量的软件包, 其中高安装量的包大多结构复杂、代码量大. 编译时检查漏洞的工具在面对大规模代码时, 或多或少地出现了部分代码无法通过编译、依赖专家知识扩展分析规则的问题. 以互操作为例, 除了基于 Python/C API 编写跨语言接口外, PyTorch 还基于接口生成工具 pybind11 声明了部分的外部函数, 多数工具不支持这一行为[22]. 具备可扩展性意味着能够相对自动化地批量部署和分析复杂软件包, 这在软

件工程实践中可能尤为重要, 定制分析规则并成功执行分析的困难和人工开销有时可能比识别大量误报更大.

● RQ5: 已有漏洞检测工具在大型 Python-C/C++互操作项目上是否具有良好的可扩展性?

PolyCruise 的分析流程依赖于用户显式指定的源汇点对, 同时其与 PolyFuzz 均需构造特定输入以触发项目执行过程中的动态行为, 因而在大型项目中难以进行系统性的评估. Mopsa 是静态分析工具, 但其对 C 扩展文件的结构有特定要求, 且需借助 Python 脚本触发相应执行路径, 因此在分析复杂项目时同样面临可扩展性方面的限制. 在实际应用方面, PolyCruise 所依赖的实现版本较低 (仅支持 Python 3.7), 其依赖分析与插桩追踪的机制需要专家定制, 加之项目本身缺乏持续维护, 限制了其适用性. Mopsa 同样仅支持较低的 Python 3.8 版本, 其静态建模所覆盖的软件包并不完整, 导致一旦遇到未建模或非常规结构, 分析过程则无法继续进行. 由于上述工具在分析条件上存在较大约束, 在实际评估中, 我们发现其难以对高下载量的 16 个软件包进行有效分析. 如表 9 所示, 项目级的 PolyCruise 仅在 numpy 上成功触发分析, 得到 3 个警告, 其都为真阳性 (*TP*), 其中 1 个为接口层运算溢出 (T1.2), 2 个为缓冲区越界访问 (M3.1). 文件级的 Mopsa 对 16 个高下载量软件包的所有文件都未能成功触发, 因此列举了其在两个小型库上的分析结果.

相较而言, CPyChecker 与 PyRefcon 的集成方式更为有效: 仅需在项目的原始编译流程中将编译器替换为支持 Python-C/C++互操作分析插件的 GCC 和 LLVM/Clang 版本, 即可实现对 C/C++扩展代码的自动化静态检测, 无需额外修改源代码或构建逻辑. 这种特性使其更易于在真实软件供应链中部署与应用, 因而我们在全部 18 个 PyPI 包 (16 个高下载, 2 个对比 PolyCruise 和 Mopsa) 上均进行了完整测试 (见表 8). 其中 CPyChecker 由于对版本迭代的支持较差, 在 18 个项目中有 13 个可以顺利编译; PyRefcon 则对 18 个项目全部编译通过, 表现出最好的可扩展性.

PyCType 依赖编译前端解析工具 pycparser, 并不利用项目原生编译脚本, 因此在配置过程中需手动指定头文件路径、宏定义和编译参数. 尽管其可扩展性相较于 CPyChecker 和 PyRefcon 较差, 但相较于 PolyCruise 和 Mopsa 仍表现更优. 如表 9 所示, 对全部 18 个 PyPI 包, 文件级的 PyCType 在所有包中都找到了可分析文件, 并在其中 50.8% 的文件中成功触发了其支持的漏洞检查.

● Answer 5. 对于表 8、表 9 中 5 个开源且非纯动态 (不完全依赖构造输入驱动) 的分析方法, PolyCruise 需要针对项目定义源汇点, Mopsa 需要修改互操作源码. 仅有 CPyChecker、PyRefcon 和 PyCType 具备相对高自动化地检查大型项目的可扩展性.

5 总结与展望

软件包库的安全性是软件供应链分析中的关键问题, 然而现有研究在分析 Python 软件包中涉及 C/C++外部语言调用的安全性时存在明显不足. 为此本文系统地构建了一个面向 Python-C/C++互操作的漏洞基准套件 (开源仓库地址: https://github.com/dynapx/PythonC-Benchmark), 涵盖 5 类语言特性共 15 种典型漏洞模式, 并结合 PyPI 中安装量最高的 16 个含 C/C++扩展的软件包进行了实证安全分析. 通过评估多种现有的先进互操作漏洞检查工具, 本文在 6 个软件包中新发现 3 种共 21 个真实漏洞, 同时分析揭示了当前研究和工具在互操作安全性分析方面的能力局限与改进方向.

(1) 提升完备性, 支持复杂语言特性互操作的安全性分析. 现有先进工具不能检查 Python-C/C++互操作漏洞基准套件 15 种漏洞模式中的 7 种, 尤其针对复杂语言特性的安全性分析存在明显不足, 包括内存管理中引用计数之外的错误, 以及异常处理和并发控制相关的错误. 跨语言互操作时多种内存接口、语言间异常和并发机制的差异是问题的根源, 值得深入分析. 同时, 对于能够支持的 8 种漏洞, 也存在漏报率较高的问题, 安全性分析往往只能覆盖最基础的语言特性.

(2) 提升可靠性, 降低已有安全检查的误报率. 部分工具误报率极高, 难以应用于工业级的供应链安全分析, 确认 21 个真实漏洞排除了超过 700 个误报. 现有外部调用安全性分析方法在建模粒度和分析精度上仍有不足, 此外 Python/C API 的版本迭代更新也是一个重要影响.

(3) 提升可扩展性, 与编译工具链集成. 现有工具在分析大型 Python-C/C++软件包时存在显著的可扩展性问题, 部分工具依赖动态输入、特定项目结构或繁杂的源汇点配置, 难以适应真实工程环境. 基于 GCC、LLVM/Clang 等编译工具链的技术表现出较好的可扩展性, 是比较可取的设计方向.

References

[1] Alfadel M, Costa DE, Shihab E. Empirical analysis of security vulnerabilities in Python packages. Empirical Software Engineering, 2023, 28(3): 59. [doi: 10.1007/s10664-022-10278-4]

[2] Guo WB, Xu ZZ, Liu CW, Huang C, Fang Y, Liu Y. An empirical study of malicious code in PyPI ecosystem. In: Proc. of the 38th Int'l Conf. on Automated Software Engineering. Luxembourg: IEEE, 2023. 166–177. [doi: 10.1109/ASE56229.2023.00135]

[3] Samaana H, Costa DE, Shihab E, Abdellatif A. A machine learning-based approach for detecting malicious PyPI packages. In: Proc. of the 40th ACM/SIGAPP Symp. on Applied Computing. Catania: ACM, 2025. 1617–1626. [doi: 10.1145/3672608.3707756]

[4] Sun XB, Gao XG, Cao SC, Bo LL, Wu XX, Huang KF. 1+1>2: Integrating deep code behaviors with metadata features for malicious PyPI package detection. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Automated Software Engineering. Sacramento: ACM, 2024. 1159–1170. [doi: 10.1145/3691620.3695493]

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

[6] Grichi M, Abidi M, Jaafar F, Eghan EE, Adams B. On the impact of interlanguage dependencies in multilanguage systems empirical case study on Java Native Interface applications (JNI). IEEE Trans. on Reliability, 2021, 70(1): 428–440. [doi: 10.1109/TR.2020.3024873]

[7] Li W, Li L, Cai HP. On the vulnerability proneness of multilingual code. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 847–859. [doi: 10.1145/3540250.3549173]

[8] Tan G, Croft J. An empirical security study of the native code in the JDK. In: Proc. of the 17th USENIX Security Symp. San Jose: USENIX Association, 2008. 365–378.

[9] Hu MZ, Zhang Y. An empirical study of the Python/C API on evolution and bug patterns. Journal of Software: Evolution and Process, 2023, 35(2): e2507. [doi: 10.1002/smr.2507]

[10] Hu MZ, Zhang Y. The Python/C API: Evolution, usage statistics, and bug patterns. In: Proc. of the 27th Int'l Conf. on Software Analysis, Evolution and Reengineering. London: IEEE, 2020. 532–536. [doi: 10.1109/SANER48275.2020.9054835]

[11] Li SL, Tan G. Finding reference-counting errors in Python/C programs with affine analysis. In: Proc. of the 28th European Conf. on Object-oriented Programming. Uppsala: Springer, 2014. 80–104. [doi: 10.1007/978-3-662-44202-9_4]

[12] Mao JJ, Chen Y, Xiao QX, Shi YC. RID: Finding reference count bugs with inconsistent path pair checking. In: Proc. of the 21st Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Atlanta: ACM, 2016. 531–544. [doi: 10.1145/2872362.2872389]

[13] Ma XT, Yan JW, Zhang H, Yan J, Zhang J. Detecting memory errors in Python native code by tracking object lifecycle with reference count. In: Proc. of the 38th Int'l Conf. on Automated Software Engineering. Luxembourg: IEEE, 2023. 1429–1440. [doi: 10.1109/ASE56229.2023.00198]

[14] Li W, Ming J, Luo XP, Cai HP. PolyCruise: A cross-language dynamic information flow analysis. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX Association, 2022. 2513–2530.

[15] Li W, Ruan JY, Yi GB, Cheng L, Luo XP, Cai HP. PolyFuzz: Holistic greybox fuzzing of multi-language systems. In: Proc. of the 32nd USENIX Security Symp. Anaheim: USENIX Association, 2023. 1379–1396.

[16] Monat R, Ouadjaout A, Miné A. A multilanguage static analysis of Python programs with native C extensions. In: Proc. of the 28th Int'l Symp. on Static Analysis. Chicago: Springer, 2021. 323–345. [doi: 10.1007/978-3-030-88806-0_16]

[17] Hu MZ, Zhang Y, Huang WC, Xiong Y. Static type inference for foreign functions of Python. In: Proc. of the 32nd Int'l Symp. on Software Reliability Engineering. Wuhan: IEEE, 2021. 423–433. [doi: 10.1109/ISSRE52982.2021.00051]

[18] Hu MZ, Yu L, Zhang Y, Han LP. A survey of multilanguage interoperability and its program analysis. IEEE Trans. on Reliability, 2025, 74(4): 4944–4958. [doi: 10.1109/TR.2025.3576267]

[19] Simons AJH. Borrow, copy or steal? Loans and larceny in the orthodox canonical form. In: Proc. of the 13th ACM SIGPLAN Int'l Conf. on Object-oriented Programming Systems, Languages, and Applications. Vancouver: ACM, 1998. 65–83. [doi: 10.1145/286936.286948]

[20] Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B. Frama-C: A software analysis perspective. Formal Aspects of Computing, 2015, 27(3): 573–609. [doi: 10.1007/s00165-014-0326-7]

[21] Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. Recent progress in program

analysis. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 80–109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[22] Hu MZ, Zhao Q, Zhang Y, Xiong Y. Cross-language call graph construction supporting different host languages. In: Proc. of the 2023 Int'l Conf. on Software Analysis, Evolution and Reengineering. Macao: IEEE, 2023. 155–166. [doi: 10.1109/SANER56733.2023.00024]

附中文参考文献

[21] 张健, 张超, 玄跻峰, 熊英飞, 王千祥, 梁彬, 李炼, 窦文生, 陈振邦, 陈立前, 蔡彦. 程序分析研究进展. 软件学报, 2019, 30(1): 80–109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

作者简介

胡明哲, 博士, 讲师, CCF 专业会员, 主要研究领域为程序分析与验证.

丁秋然, 本科生, CCF 学生会员, 主要研究领域为软件工程与安全.

张子涵, 硕士生, CCF 学生会员, 主要研究领域为软件工程与安全.

谢金言, 硕士, 主要研究领域为程序分析.

于乐, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为系统安全.

韩丽萍, 博士, 讲师, CCF 专业会员, 主要研究领域为软件测试.