中国科学技术大学

博士学位论文

基于语言接口的多语言软件的安全性及其静态分析

作者姓名：　胡明哲

学科专业：　网络空间安全

导师姓名：　熊焰 教授　张昱 教授

完成时间：　二〇二三年十一月十七日

University of Science and Technology of China

A dissertation for doctor's degree

Safety and Static Analysis of Language Interface-based Multi-language Software

Author: Mingzhe Hu

Speciality: Cyberspace Security

Supervisors: Prof. Yan Xiong, Prof. Yu Zhang

Finished time: November 17, 2023

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

控阅的学位论文在解密后也遵守此规定。

☑ 公开 ☐ 控阅（____ 年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘　要

随着数字化转型的不断推进，软件已经成为影响国民经济和社会发展的战略支点，软件安全的重要程度也日益凸显，事关国计民生。在众多软件中，涉及编程语言互操作的多语言软件（如 Python 调用 C/C++ 的 PyTorch 等）越来越多地应用于数据科学、深度学习等领域，推动了人工智能、智能制造等新兴技术产业的发展。但是，由于语言特性差异和语言接口设计对互操作性的影响，多语言软件相较单语言软件更加易错且难以调试。常见漏洞披露 CVE 收录了不同语言互操作的多语言软件的安全问题，相关漏洞可能导致内存泄漏、未定义行为等，影响智能系统的安全性和可靠性。作为保障软件安全的重要技术手段，静态分析技术可以在不运行程序的情况下识别程序的正确性、安全性等特性，并应用于漏洞检查、程序理解等场景。然而，现有的静态分析研究成果主要针对单语言程序，研究支持编程语言互操作的多语言软件的静态分析技术势在必行。

本文围绕基于语言接口的多语言软件的安全性及其静态分析，系统地研究了编程语言互操作中语言接口的设计使用与漏洞模式、外部函数的类型推断、跨语言的调用图构建等问题，分析总结了多语言软件的漏洞模式，设计实现了一系列支持编程语言互操作的静态程序分析技术，在大量使用的开源仓库中发现了多种漏洞，提高了多语言软件的安全性、可靠性和可维护性。本文的具体研究工作包括：

（1）**Python/C 语言接口的提取分析与漏洞模式**。外部接口是编程语言互操作的编程接口，它的设计实现和使用模式反映了语言间不同语言特性的协调交互，是理解互操作性、分析跨语言程序潜在漏洞的基础。互操作语言间特性的差异体现在异常处理、类型系统、内存管理等多个方面，本文以语言特性差异更大、近年来广泛使用但研究较少的 Python 和 C/C++ 互操作为研究对象，首先基于编译前端设计实现了 Python/C 跨语言编程的语言接口 Python/C API 的提取分析工具，分析了 Python/C API 在 Python 编译器中的设计迭代和在主流 Python/C 多语言软件中的使用行为，包括 Python/C API 的数量、增删改过程中的二进制兼容性、使用热点和稳定性等。然后，本文结合 Python 和 C/C++ 语言特性的差异，分析总结了 Python/C 多语言软件潜在的 9 类漏洞模式，包括异常处理错误、不完整的错误检查、内存管理错误、整型溢出、迭代兼容性、缓冲区溢出等。最后，本文通过语法模式匹配等方法设计实现了其中 6 类漏洞的轻量级的漏洞检查工具，并在大量使用的 Python/C 多语言软件中发现了漏洞实例。

（2）**动态类型语言的外部函数的静态类型推断**。跨语言程序的类型误用作为本文总结的 9 类多语言漏洞之一，难以在语法层面进行有效的漏洞检查。同

时，静态类型推断作为维护动态类型语言的安全性的基础技术，也无法直接作用于包含外部函数调用的跨语言程序。为此，本文提出了一种不依赖类型标注的、确定性的 Python 的 C 外部函数的静态类型推断方法。首先，本文形式建模了 Python/C 跨语言程序的静态语义，包括抽象语法、类型系统和子定型规则。然后，本文把外部函数类型推断的前提表示为三类可组合的假设判断，分别对应外部函数声明、参数类型转换、返回类型转换三类语言接口，根据其具体组合推断外部函数的函数签名。最后，作为类型推断的副产物，本文的方法可以检查外部函数声明与实现不一致的漏洞，该漏洞会导致无参外部函数可以接收任意类型的参数。在大量使用的 Python/C 多语言软件上的实验表明，本文的方法可以无误报地推理外部函数的类型签名并检查声明与实现不一致的漏洞，本文在不同多语言软件中发现了多个漏洞实例。作为对 Google 的 Python 单语言静态类型推断工具 Pytype 的增强，本文的方法可以将其类型推断召回精度平均提高 27.5%。

（3）**支持多种宿主语言的跨语言调用图构建**。调用图构建是安全扫描、错误定位、代码跳转、补全与重构、程序理解等多种程序分析任务的基础。然而，已有的调用图构建方法大多针对单语言软件设计与实现，少量支持多语言软件的方法从外部语言出发或依赖专家知识，对于不同的宿主语言和互操作接口缺乏泛化能力。本文通过抽象描述不同互操作机制下外部函数声明的语义，把跨语言调用图的构建转换为语言无关的外部映射语义提取和图变换与节点融合算法。在 Python/C 语言接口 Python/C API、接口生成工具 pybind11 和 JavaScript/C 语言接口 Node.js C++ addons 上的实验表明，本文的方法在具有泛化性的同时，取得了较依赖专家知识的语法匹配方法更好的精度，并保持了较好的时间效率。

关键词：编程语言互操作；程序分析；漏洞检查；类型推断；调用图构建

ABSTRACT

With the steady promotion of digital transformation, software has become a strategic fulcrum that affects the national economy and social development. The importance of software security has become increasingly prominent, and it is related to the country's stability and the people's well-being. Among plenty of software, multi-language software involving programming language interoperation (such as Python calling C/C++ in PyTorch) is increasingly used in fields like data science and deep learning, and it promotes the development of emerging technologies and industries like artificial intelligence and smart manufacturing. However, due to the impacts of language feature discrepancies and language interface designs on interoperability, multi-language software is more error-prone and more difficult to debug than single-language software. Common Vulnerabilities and Exposures (CVE) program catalogs the safety and security issues of multi-language software with different interoperating languages. The bugs can lead to memory leaks, undefined behaviors, etc., and affect the security and reliability of intelligent systems. As an important technique to ensure the software security, static analysis can identify the correctness, safety and other properties of a program without running it, and can be used in tasks like bug detection and program comprehension. However, existing static analysis research mainly focuses on single-language programs. It is imperative to study the static analysis of multi-language software that supports programming language interoperability.

Focusing on the safety and static analysis of language interface-based multi-language software, this dissertation systematically studies the design and use of language interface and related bug patterns, type inference of foreign functions, and cross-language call graph construction in programming language interoperability. This dissertation analyzes and summarizes the bug patterns of multi-language software, designs and implements a series of static program analysis technologies that support programming language interoperability, finds a variety of bugs in widely used open source repositories, and improves the safety and reliability of multi-language software. The specific research work of this dissertation is as follows:

(1) **Extraction, analysis, and bug patterns of Python/C language interface**. The foreign interface is the programming interface for language interoperability. Its design, implementation, and usage reflect the incorporation and interaction of different language features between languages. It is the basis for understanding interoperability and ana-

lyzing potential bugs of multi-language programs. Features of interoperating languages differ from each other in aspects of exception handling, type system, memory management, etc. This dissertation focuses on the interoperability between Python and C/C++, which is widely used in recent years and has significant differences in language features. First, based on the compiler front-end technologies, this dissertation designs and implements an extraction and analysis tool for the Python/C language interface Python/C API, analyzes the design evolution of the Python/C API in the Python compiler and its usage behavior in mainstream Python/C multi-language software. Research findings include the number of Python/C API, binary compatibility during its additions, deletions and changes, usage hotspots and stability, etc. Second, this dissertation analyzes and summarizes nine bug patterns of multi-language software based on different language features between Python and C/C++, including mishandling exceptions, insufficient error checking, memory management flaws, integer overflow, evoluation compatibility, buffer overflow, etc. Finally, this dissertation designs and implements lightweight bug detection tools for 6 of the 9 bug patterns through syntactic pattern matching and other methods, and finds bug instances in widely used Python/C multi-language software.

(2) **Static type inference for foreign functions of dynamically typed languages**. Type misuse of cross-language programs is one of the nine multi-language bugs summarized in this dissertation, and it is difficult to detect effectively at the syntactic level. At the same time, static type inference, as a basic technology for maintaining the safety of dynamically typed languages, cannot directly perform on cross-language programs that contain foreign function calls. To this end, this dissertation proposes a type-annotation-independent and deterministic method for static type inference of C foreign functions of Python. First, this dissertation formally describes the static semantics of the Python/C cross-language programs, including their abstract syntax, type system, and subtyping rules. Second, this dissertation represents the premise of foreign function type inference as three combinable hypothetical judgments, corresponding to foreign function declarations, parameter type conversions, and return type conversions, respectively, and infers the signature of foreign functions according to their specific combinations. Finally, as a by-product of the type inference, the system can check the inconsistency between the declaration and the implementation of foreign functions, which can cause parameter-free foreign functions to accept arguments of arbitrary types. Experiments on widely used Python/C multi-language software show that the proposed method can infer the type signature of foreign functions without false positives and check the inconsistency bug, this dissertation finds several bug instances in different multi-language software. As an

enhancement to Google's Python single-language static type inference tool Pytype, the method can improve its recall accuracy by 27.5%.

(3) **Cross-language call graph construction supporting different host languages**. Call graph construction is the basis for various program analysis tasks, such as security scanning, error localization, go-to-definition jump, code completion and refactor, etc. However, existing call graph construction methods are often designed and implemented for single-language software. A few methods that supports multi-language software are based on the foreign language programs or rely on the knowledge of experts, they lack generalization capabilities for different host languages and inter-operation interfaces. In this dissertation, by abstractly describing the semantics of foreign function declarations with different interoperability mechanisms, the construction of cross-language call graphs is represented as language-independent foreign mapping semantics extraction and graph transformation and node fusion algorithms. Experiments on the Python/C language interface Python/C API, the interface generation tool pybind11 and the JavaScript/C language interface Node.js C++ addons show that the method proposed in this dissertation achieves generalization, as well as better accuracy than the syntactic matching method based on experts' knowledge, and good time efficiency.

Key Words: programming languages interoperability; program analysis; bug detection; type inference; call graph construction

目　录

插 图 清 单

表 格 清 单

算 法 清 单

符 号 说 明

C_i	Python 版本 i 的 Python/C API 全集
A_i	Python 版本 i 的 Python/C API 函数集合
M_i	Python 版本 i 的 Python/C API 宏定义集合
$\bar{P}_j$	多语言软件 j 中的 `Py*` 标识符组成的集合
P_j	多语言软件 j 使用的 Python/C API 集合
K_{j_1}, K_{j_2}	多语言软件 j 的两个版本使用到的 Python/C API 集合
J_j	多语言软件 j 两个版本使用到的 Python/C API 集合的 Jaccard 相似性，$J_j = \frac{\lvert K_{j_1} \cap K_{j_2} \rvert}{\lvert K_{j_1} \cup K_{j_2} \rvert}$
f^{p}	外部函数在宿主 Python 侧的调用名
f^{c}	外部函数在外部 C 侧的实现函数
$flag \in \mathbb{F}$	`PyMethodDef` 结构体中的位段标记
$\iota_{\mathrm{ap}} \in \mathbb{I}_{\mathrm{ap}}$	参数解析 Python/C API
$\iota_{\mathrm{vb}} \in \mathbb{I}_{\mathrm{vb}}$	值构建 Python/C API
$\iota_{\mathrm{ec}} \in \mathbb{I}_{\mathrm{ec}}$	显式转换 Python/C API
$\tau, \eta \in \mathbb{T}_{\mathrm{p}}$	Python/C 跨语言程序中 Python 侧的类型
$\alpha, \beta \in \mathbb{T}_{\mathrm{c}}$	Python/C 跨语言程序中 C 侧的类型
$\tau' <: \tau$	τ' 是 τ 的子类型
$\Gamma \vdash e :: \tau$	类型赋值，表达式 e 在定型上下文 Γ 中有类型 τ
$f^{\mathrm{p}} \xhookrightarrow{flag} f^{\mathrm{c}}$	外部函数声明
$\mathbb{P}$	程序属性
$\tau \ {}_{\mathrm{p}}\overset{\mathbb{P}}{\mapsto}_{\mathrm{c}} \alpha$	Python 的 C 外部函数的参数类型转换
$\tau \ {}_{\mathrm{p}}\overset{\mathbb{P}}{\Longmapsto}_{\mathrm{c}} \alpha$	Python 的 C 外部函数的参数类型转换（带有数值溢出检查）
$\alpha \ {}_{\mathrm{c}}\overset{\mathbb{P}}{\mapsto}_{\mathrm{p}} \tau$	Python 的 C 外部函数的返回类型转换
$\{\pi(\mathbb{P})\}?\{J\}$	假设判断 J 仅在关于属性 $\mathbb{P}$ 的谓词 π 为真时成立
$\frac{J_1 \cdots J_n}{J}$	由判断 $J_1, \cdots, J_n$ 可以推导出判断 J
$\mathcal{P}_{\mathrm{PFA}}$	无参分析

$\mathcal{P}_{\text{UPA}}$	未使用形参分析
$\tau_1 \mid \tau_2$	和类型
(τ_1, τ_2)	积类型
$\tau_{in} \to \tau_{out}$	函数类型
$"u_1, \cdots, u_n"$	由 n 个格式化单元组成的格式化串
$(\beta_1) \cdots (\beta_n)e$	类型转换表达式
$\mathcal{T}_{\text{RDA}}$	可达定义分析
$f \to g$	函数 f 调用函数 g
$f \ {}_{\text{h}}\!\!\to_{\text{c}} g$	跨语言的外部函数调用，定义在宿主语言侧的函数 f 调用定义在外部语言侧的函数 g
$\mathcal{M}$	外部映射语义模型
$\mathcal{M}\|_m$	特定跨语言互操作方法（表示为外部映射提取规则 m）参数化后的外部映射语义模型
$\mathcal{B}$	多语言代码库
$\mathcal{S}$	外部映射总结
$\mathcal{M}'$	回调行为的外部映射语义模型

第 1 章　绪　　论

1.1　研究背景及意义

软件产业是国民经济和社会发展的基础性、先导性、战略性和支柱性产业，对经济社会发展具有重要的支撑和引领作用。现代软件系统很多是多语言的，软件系统的不同组件由不同的编程语言编写得到，并且不同语言之间存在着交互关系。多语言的软件架构兼具互操作语言双方的优点。一方面，伴随着编程语言的发展和新兴应用领域的需要，新的编程语言、语言特性和编程范式被设计实现并广泛使用。这些高级语言和编程模型凭借高效的开发效率[1-3]、编译辅助的安全机制[4]、轻量级线程带来的高并发[5]、对领域应用的生态支持[6-9]等优势吸引了越来越多的开发者。根据 JetBrains 公司 2022 年度的开发者生态调查[10]，半数受访开发者计划使用一门新的编程语言。另一方面，存量代码尤其是低级语言（C、C++、汇编等）编写的大量仓库仍然占据重要地位，如基础线性代数子程序[11]（Basic Linear Algebra Subroutines, BLAS）、压缩库 zlib[12]、加密库 OpenSSL、操作系统接口等，它们避免了重复的开发劳动、经过了大量的测试和优化、更贴近机器指令从而提供了更优的时空性能。统计发现（2022 年 12 月数据），GitHub 上星标数最多的 60 个 C++ 项目中，有 24 个存在和其他高级编程语言的互操作，其中包括 Python/C 互操作的深度学习框架 TensorFlow[13] 和 PyTorch[14]、自动驾驶平台 Apollo[15]、加密货币 Bitcoin[16]、数据库 RethinkDB[17]，Java/C 互操作的云原生代理 Envoy[18]、流媒体 AI 套件 MediaPipe[19]、移动 UI 框架 Weex[20]，JavaScript/C 互操作的桌面开发框架 Electron[21]，等等。

虽然多语言的软件架构实用且常见，但是编写安全可靠的多语言软件并不容易。在多语言软件中，宿主语言通过外部接口（Foreign Interface，FI）调用外部语言。同一对互操作语言可能存在不同机制与实现的外部接口，其中基于语言接口的外部语言互操作几乎是所有主流通用编程语言的语言标准的一部分（详见第 2.1 节）。外部接口的复杂性以及不同语言之间语言特性的差异，如内存管理、类型系统、异常处理、并发机制等的不同，增加了跨语言编程的难度，导致在跨语言接口层可能存在多种程序漏洞。同时，低级语言一般是非类型安全、非内存安全的[22-23]，多语言互操作既增加了漏洞传播的风险，又阻碍了错误定位等调试手段的有效应用。相关的实证研究发现，软件系统中不同语言间的依赖越多，存在程序漏洞的风险就越高，大约是单语言系统的 2–3 倍[24]。Li 等人[25]对大量的开源软件及其修复记录的分析也发现，语言和互操作机制越多越复杂，软件系统存在安全隐患的可能就越高。常见漏洞披露（Common Vulnerabilities and Exposures, CVE）由非营利组织 MITRE 下属美国国家网络安全中心（NCF）管理

维护，是权威的索引信息安全漏洞披露的数据库。CVE 中收录了 Python、Java、Rust、Ruby、PHP 等不同编程语言通过语言接口和外部语言互操作时由于跨语言的内存管理、异常处理、外部漏洞代码等原因导致的多语言软件系统的安全问题[26-32]。以 Python/C 多语言软件为例，相关漏洞[27-28,33-34]影响了 TensorFlow 等广泛使用的深度学习框架，可能导致 Python/C 跨语言程序存在内存泄漏、缓冲区溢出、未定义行为等问题，并可能被利用形成相关的安全攻击，危害智能车、智能机器人等新兴智能系统的安全性和可靠性。研究人员也在多语言软件中发现了跨语言的异常处理[35-38]、内存管理[33-34,39]、类型检查[40-41]、并发机制[37,42]等不同方面的程序错误。这些多语言软件漏洞可能导致系统崩溃、非预期的行为、安全攻击等多种问题，给社会生产生活带来极大的危害和损失，影响数字化转型的速度和质量。

作为漏洞检查的一种有效手段，程序分析[43]是指对计算机程序进行自动化的处理，推理或跟踪程序的行为，以确认或发现安全性、正确性、性能等特性，其结果可以用于编译优化、程序理解等。静态程序分析无需运行程序，能够覆盖更多的执行路径，提供更可靠的安全保障，已经被用于广泛应用于生产生活中单语言软件的安全分析，在包括飞机飞行控制软件[44]、安卓手机操作系统[45]等关键系统中发现了程序错误，避免了潜在影响巨大的安全风险。然而，已有的静态分析技术主要针对单语言程序设计，难以有效应用于多语言软件[38,46]。随着多语言软件在不同应用领域的广泛使用，以及多语言软件缺陷的不断出现[33-37,39-41,47]，对多语言软件的静态分析的需求也日益增加。随着编程语言的发展和新兴语言的流行，对当下广泛使用但语言特性差异较大的多语言软件（如 Python/C）的漏洞模式缺乏系统的分析总结，复杂语言特性如动态类型系统对互操作的影响也缺少有效的静态分析检查技术，同时日益增多的多语言软件架构也对程序分析技术的泛化性提出了更高要求。考虑到应用背景的发展，多语言软件的静态分析需要解决以下三类问题：

（1）**新兴多语言软件的漏洞模式**。一方面，针对单语言程序定义的漏洞模式在多语言软件中的表现并不相同，例如整数溢出、缓冲区溢出等经典程序漏洞，它们在多语言软件中有着不同的表现形式，单语言视角下正确的程序仍然可能由于跨语言互操作而导致相应的安全漏洞；另一方面，互操作带来了一些跨语言程序所特有的漏洞模式，例如语言特性差异所带来的异常处理错误、内存管理错误等漏洞。多语言软件的漏洞模式分析需要考虑互操作双方语言特性的差异和外部接口的使用行为。对于近年来随着深度学习的流行而大量使用的 Python/C 多语言软件，动态类型、解释执行的宿主语言 Python 在语言特性上和 Java、C、C++ 等静态类型的编译型语言差异更大，这也影响并反映在 Python 和 C/C++ 互操作的外部接口 Python/C API 的设计和使用中。提取并分析外部接口在编译器

和主流多语言软件中的使用行为、研究不同语言特性对多语言软件安全性的影响是利用程序分析提高多语言软件安全性的基础。

（2）**复杂语言特性的跨语言程序分析**。大量的程序分析问题都是不可判定的[48]，对于涉及复杂语言特性如类型、内存、并发等系统的跨语言程序漏洞，其程序分析方法需要精细的设计，以提高漏洞检查的精度。典型地，Python 等语言的动态类型系统增加了类型错误的风险[49]，同时互操作可能触发非类型安全的 C 代码中的未定义行为[50]，但是已有的类型检查技术却不能直接作用于动态类型语言和静态类型语言互操作的跨语言程序。检查多语言软件的类型误用需要设计外部函数的类型推断方法，包括通过形式化方法描述跨语言程序的静态语义，并给出跨语言类型推断的分析规范。外部函数的类型推断方法能够增强多语言软件的类型检查能力，提高多语言软件的安全性和可维护性。

（3）**跨语言基础程序分析**。程序分析技术包含一些基础理论和关键技术，它们回答关于程序的某些基本性质，并为许多客户分析（client analysis）提供依赖信息。例如，调用图构建记录程序控制流中的函数调用关系，是编译优化、安全分析、错误定位等许多程序分析任务的基础。桥接宿主语言和外部语言的接口代码根据互操作语言和外部接口设计的不同，会在外部函数声明、外部对象映射等方面产生不同的计算效果，进而阻碍基础程序分析技术的有效应用。设计支持跨语言互操组的基础程序分析技术能够更好地推理和跟踪跨语言程序的行为，并为构建高精度的多语言软件的程序分析方法提供基础。

本文主要围绕以上多语言软件的静态分析的三类问题展开，系统地研究了编程语言互操作中外部接口的设计使用和漏洞模式、外部函数的静态类型推断、跨语言的调用图构建等科学问题，给出了多语言软件的漏洞模式分类，设计了一系列支持跨语言互操作的静态分析技术，增强了跨语言程序的基础程序分析能力，在大量使用的多语言软件中发现了多种漏洞，提高了多语言软件的安全性、可靠性和可维护性。

1.2　国内外研究现状及挑战

学界和业界对于多语言软件的程序分析已经取得了多方面的研究进展：通过实证分析发现了多语言互操作对软件安全性的影响，通过形式语义描述了编程语言互操作的安全规范，分析总结了特定跨语言互操作和外部接口的漏洞模式，设计实现了针对不同语言特性的跨语言程序分析技术，等等。实证安全分析和互操作语义研究分别从实践和理论层面说明了多语言软件的易错性，漏洞模式分类分析总结了以 Java/C 互操作为代表的多语言软件的常见漏洞，后续工作通过分析具体的语言特性解决了相关漏洞的安全检查，包括异常处理、内存管理

等语言特性差异引发的多语言软件缺陷。然而，这些方法依然存在不足之处，同时对于多样化的多语言软件架构如新兴的 Python/C 多语言软件、复杂的技术方法和语言特性如动态类型的宿主语言、语言无关的跨语言的基础程序分析技术，仍然存在需要研究和解决的问题和挑战。

尤其国内针对多语言软件的程序分析工作尚处于起步和发展阶段，Li 等人及其团队对多语言软件中互操作语言的选择[51]和识别[52]、安全性问题[25]进行了实证分析，提出了跨语言的动态信息流分析方法[53]。Zhang 等人及其团队提出了支持 Python 外部库的测试用例生成方法[54]和内置类型相关漏洞的测试集[55]。Hua 等人及其团队研究了 Python 标准库包括外部语言模块在内的安全问题及其漏洞检查[56-57]，提出了 C 到 Rust 的代码翻译[58]和 Rust/C 跨语言程序的程序分析框架[59]。国内多语言软件的程序分析研究在多语言软件的形式语义建模、复杂语言特性的程序分析、基础程序分析等方面仍然较为薄弱。

1.2.1　多语言软件的安全性分析的研究现状

多语言软件的安全性分析首先通过实证分析和语义研究分别从实践和理论两方面发现和证明了多语言软件的易错性，并在此基础上分析总结了多语言软件的漏洞模式。随着编程语言的发展，实证安全分析已逐渐覆盖了新兴的多语言软件架构，互操作语义研究也对不同的语言特性逐渐给出了更具一般性的结果，这启发着现有的漏洞模式总结扩展到 Java/C 多语言软件以外的新兴架构，以及泛化能力更好的跨语言基础程序分析的研究。

1. 多语言软件的实证安全研究

实证研究（empirical study）通常包括数据和经验的收集和分析，通过实验、案例分析、研究实际工程项目、采访或问卷等手段定性、定量地分析软件开发的工具、技术、实践过程等因素，实证结果描述、评估和揭示软件工程某些方面的现状，形成一个知识体系以引导相关理论和技术的研究。Mayer 等人[60]通过对 139 名专业开发者的问访，发现平均每个项目会使用 7 种编程语言（含 shell、make 等脚本语言），其中包括 3 对互操作语言。开发者们认为多语言互操作提供了快速原型化的能力，但是也影响了软件的可理解性和可重构性。超过 90% 的开发者在跨语言互操作中遇到过问题，认为辅助工具尤其是漏洞检查器能够为他们带来帮助。Sultana 等人[61]分析了 20 个 C/Fortran 互操作的多语言软件、超过 1200 万行代码，发现：（1）由于历史代码（早于 Fortran 2003 标准[62]）在 C/Fortran 互操作方法上与特定编译器实现相关，目前仅有 3% 的互操作实例在描述跨语言接口时遵循现行的国际标准化组织（International Organization for Standardization, ISO）标准；（2）部分外部函数的参数类型也是编译器相关的，且不和现行标准兼容，可能导致程序漏洞[63]；（3）23% 的外部函数参数是只读的，却按引用传

递，这违反了最小权限原则并可能导致安全攻击[64]。Grichi 等人[24]设计了两种多语言依赖关系的分析方法，分析了 10 个多语言软件，发现：（1）程序中跨语言的依赖越多，引入漏洞的风险就越高，而对于单语言则维持常量；（2）跨语言的依赖中非直接依赖是直接依赖的 2.7 倍，而非直接依赖难以被静态程序分析工具发现；（3）跨语言依赖引入程序漏洞的概率约是单语言内依赖的 3 倍，而引入可能导致安全攻击的漏洞的概率约是 2 倍。Li 等人[25]分析了 GitHub 上 4001 个多语言软件项目 3 年内超过 2000 万条提交的演化历史，发现：（1）多语言软件的易错性和语言的选择是相关的，在超过 20 组多语言组合中，Python/C 互操作导致的软件漏洞增多是最多的；（2）除了互操作语言本身，互操作机制也会影响多语言软件的易错性，基于语言接口的互操作相比于基于通用运行时的互操作更容易导致多语言软件漏洞。通过测试与分析语言无关的动态程序分析工具 ORBS[65]在 10 个不同语言组合的多语言软件上的表现，Yang 等人[66]发现缺乏语法知识的语言无关的程序分析技术在效率和可扩展性上存在着较大的不足。

多语言软件的实证安全研究通过对现实软件工程中的多语言项目的实证分析，发现并总结了软件开发过程中编程语言互操作的使用情况，通过实证展现并分析了多语言软件相比单语言软件的易错性，体现了工程实践对多语言软件的程序分析的实际需求。

2. 编程语言互操作语义

互操作性的形式语义使用形式化方法描述多语言软件的安全规范，为严格地推理和证明如类型安全、上下文等价等程序性质提供了可靠的数学基础。尽管我们容易知道在安全的语言中操作非安全的语言的数据结构是不安全的，这也是一些工作的基本假设[67]。然而实际上，由于不同的语言特性带来的计算效果的差异，安全的语言之间的互操作也可能是不安全的。Trifonov 和 Shao[46]说明了这一点，他们基于效果系统[68-70]（effect system）形式化地描述了在可变存储、异常等特性上不相同的静态类型语言之间的互操作语义。抽象理论[71]（abstraction theorem）通过证明源代码到目标代码的编译满足上下文等价（contextual equivalence）性质，使得程序分析和验证技术可以在源码层面推理程序的安全性和可靠性。但是跨语言互操作导致目标语言可能存在缺少源语言层面等价性的特性，这种破坏编译过程的全抽象（full abstraction）属性的特性已经在 CLR（Common Language Runtime）、JVM（Java Virtual Machine）等支持多种语言及其互操作的通用运行时中被确认存在[72-73]。Ahmed 和 Blume[74]提出了一种保持源代码和目标代码的上下文等价性的续延传递风格（Continuation Passing Style，CPS）翻译，该 CPS 翻译基于一种以类型转换选择为核心的多语言语义，要求源类型保持程序行为的同时，在翻译到目标类型时也具有语义等价的程序行为。进一步地，考虑到外部库的非安全性，其甚至可能包含恶意代码，Larmuseau 等人[75]提出了一种包

含内存隔离机制的多语言操作语义（operational semantics），限制目标语言的语言特性保持多语言系统的安全编译（全抽象）属性。Matthews 和 Findler[76] 提出了一种被称作边界（boundary）的构造用以描述多语言系统的语义，并在源语言的语法层面证明了基于边界的编程语言互操作语义的类型安全性。然而，该方法要求互操作双方的求值规则能够相互嵌入，无法作用于语言特性差异很大的多语言系统。Patterson 等人[77] 通过可实现性模型[78-79]（realizability model）将源语言的类型解释为目标语言中具有相同行为的项，从而证明互操作双方的类型之间的可转换性，提供多语言系统的语义类型安全。

编程语言互操作语义通过形式语义严格地描述了多语言软件考虑特定语言特性时的安全规范，证明了互操作语言的语言特性和程序性质对多语言软件的安全影响。编程语言互操作语义为分析和验证多语言软件的安全性提供了理论基础，同时语义抽象模型在一定程度上揭示了基础程序分析语言无关的一般性。

3. 多语言软件的漏洞模式分类

研究人员通过实证分析和形式语义已经发现并证明了，受互操作语言及其语言特性、互操作机制等因素的影响，多语言软件系统在实践和理论上都存在着更大的漏洞隐患。Java 是一种广泛使用的静态类型、自动垃圾收集、面向对象的编程语言，作为 Java 和本地代码（低级语言编写的本地应用或库）之间的外部接口，Java 本地接口（Java Native Interface，JNI）相关的漏洞模式得到了较好的分类总结。Liang[80] 给出了 JNI 设计的文字规范，涵盖类型、异常等多种语言特性。同时，作为 Java/C 跨语言程序的编程指南，该规范覆盖了 15 类 JNI 编程中可能遇到的错误。随着 Java 语言的发展，Kondoh 和 Onodera[37] 补充了一类并发相关的错误。针对该并发错误和 Liang 总结的 15 类漏洞中的 3 类，他们设计了基于类型状态数据流分析[81] 和语法检查的轻量级的检查工具。Java 开发套件（Jave Development Kit，JDK）由 Java 编译工具链和 Java 运行时环境（Java Runtime Environment，JRE）组成，其中包含大量通过 JNI 连接的本地代码。Tan 和 Croft[38] 通过分析 JDK 中的外部 C 代码，给出了 Java/C 多语言软件的漏洞模式分类，同时发现已有的针对单语言软件设计的程序分析工具在检查这些多语言漏洞时有着很高的误报率。

多语言软件的漏洞模式分类分析总结了特定互操作语言的多语言软件的安全隐患，是对实证和语义安全研究的具体归纳总结。实证和语义安全研究已经覆盖了更多更新的多语言软件架构，漏洞模式分类仍以 Java/C 互操作的多语言软件为主要研究目标。

1.2.2　不同语言特性的跨语言程序分析的研究现状

互操作语言的语言特性差异既影响着外部接口的设计，又反映在很多多语言软件的漏洞模式中。针对多种语言特性，相关工作已经在不同的互操作语言和机制中设计实现了跨语言程序分析的方法。其中已经包括了一些复杂语言特性的处理，如垃圾收集语言和手动内存管理语言互操作时的内存安全等，同时也反映出针对一些新兴编程语言的复杂语言特性的不足，如在分析多语言软件的类型安全时仍以静态类型的宿主语言为主要研究目标，而动态类型语言的流行性和易错性已在在实证分析中被发现。

1. 内存管理的跨语言程序分析

常见的外部语言 C/C++ 采用手动内存回收的动态内存管理机制，在时空性能优化空间更大的同时，容易存在内存泄漏、多次回收等内存安全问题[82]。宿主语言一般是使用自动垃圾收集的高级语言，由基于不同算法的垃圾收集器自动回收不再使用的对象。对于共享内存空间的多语言互操作机制，多语言软件在需要正确地手动回收外部对象的同时，也可能需要手动维护跨语言接口层中宿主语言对象的引用关系。Python 使用基于引用计数算法[83]的自动垃圾收集，而在 Python/C 多语言软件中，传递到 C 侧或在 C 侧创建的 Python 对象不在 Python 编译器自动垃圾收集的范围内，需要程序员利用特定的外部接口手动增减其引用计数，同时由于借引用、偷引用等特性的存在[84]，人工维护对象的引用计数是易错的。Li 和 Tan[33]提出了一种检查 Python/C 跨语言接口代码中引用计数内存错误的方法，他们通过仿射分析[85]（affine analysis）跟踪如下不变式：在变量的引用不逃逸出其作用域时，一个对象引用计数的变化量在其作用域结束时应该为零；在存在逃逸时（通过返回值或写堆内存），则应该等于其引用的逃逸数。Mao 等人[34] 使用不一致路径对（Inconsistent Path Pairs，IPP）方法检查引用计数错误，一个不一致路径对是同一函数内两条具有不一样的引用计数变化量的路径。IPP 方法不关心函数被调用的上下文从而简化了分析，同时泛化了对操作引用计数的外部接口的语义模型，使得问题可以扩展到多线程环境中对系统资源使用的跟踪。即便宿主语言在设计时就考虑了内存错误的避免机制，如 Rust 基于所有权（ownership）的内存管理系统，外部调用仍然可能导致多语言系统的内存安全问题，Li 等人[39]在 Rust/C 多语言软件中发现了多种内存漏洞。为了避免 C/C++ 外部程序的内存安全漏洞影响整个多语言软件，同时使得 Java 等高级语言已有的内存安全检查能够作用于外部对象，已有的研究提出了把外部对象分配在宿主堆上[86-87]、重新划分宿主堆[88-89]等方法。

内存管理的跨语言程序分析主要关心不同内存管理机制的语言和以 C/C++ 为代表的手动内存管理语言互操作时的内存安全，不同的内存管理机制包括基

于引用计数的垃圾收集、基于所有权的内存管理等，它们分别是 Python、Rust 等不同新兴编程语言在内存管理方面的语言特性，也都给多语言软件的内存安全带来了不同的影响。

2. 类型系统的跨语言程序分析

类型系统既表现了很多语言特性，又反映了值的内存布局。协调两个语言的类型系统是互操作性的重要设计，跨语言的类型转换一方面解释一个语言的语言特性如何在另一个语言中表达，另一方面也指明不同大小和对齐方式的值如何在语言之间传递。由于语言特性的移除和转换，多语言软件可能存在运行时无法捕获的跨语言类型误用。同时，外部函数调用可能在类型安全的宿主语言中引入违反安全的行为。Furr 和 Foster[40] 提出了一种多语言的类型推断方法来检查跨语言的类型安全，通过支持 OCaml 类型和 C 类型相互嵌入的类型系统来跟踪跨语言的类型信息。由于许多 OCaml 类型传递到 C 程序中后有着相同的物理表示，Furr 和 Foster 利用一个表征类型来表示多个 OCaml 类型。此外，由于 C 程序拥有 OCaml 对象的底层视角，类型推断系统得以基于数据流分析来跟踪内存偏移和标签信息。Furr 和 Foster[90] 进一步设计了一种支持 JNI 程序的跨语言类型推断方法，通过在类型系统中增加字符串变量解决 JNI 使用参数化的格式化字符串表示多态的（polymorphic）外部函数签名的问题。通过外部接口 JNI，外部 C 代码可以绕过 Java 的类型安全机制。Tan 等人 [41] 设计了一种多语言的类型安全检查方法，通过静态程序分析保证 Java 指针在 C 程序中的不透明性和 JNI 函数指针的只读性，通过动态程序分析在运行时检查跨语言程序的安全性，包括外部函数的参数类型检查、访问控制、数组边界检查、异常检查等等。

类型系统的跨语言程序分析研究多语言软件在跨语言值传递时应当满足的类型约束，已有的研究主要关注静态类型语言如 Java 和 C/C++ 互操作时的类型安全。

3. 异常处理的跨语言程序分析

异常处理的差异影响着跨语言互操作。在多语言软件中，对于不支持异常或使用错误码替代的低级外部语言，需要由外部接口负责抛出异常。但是外部接口抛出的异常不能被宿主语言的异常检查规则检查，并且不能立即终止跨语言程序的执行，而是需要等待外部函数返回，这使得多语言软件易错且难以调试。Kondoh 和 Onodera[37] 定义了和 JNI 异常处理相关的三类状态及其之间的转换，分别是有异常挂起的 Thrown 状态、无异常挂起的 Cleared 状态和未知是否抛出异常的 Unchecked 状态。例如，JNI 函数 `ExceptionClear` 触发 Thrown 到 Cleared 的异常状态转移，`CallVoidMethod` 触发 Cleared 到 Unchecked 的异常状态转移，等等。基于类型状态数据流分析 [81] 可以求解异常状态停留在 Thrown 或 Unchecked 的异常处理错误，Kondoh 和 Onodera 在 Java Gnome 和 Mozilla Firefox

等大量使用的 Java/C 多语言软件中发现了错误实例。对于外部接口抛出的异常无法立即终止跨语言程序的控制流的问题，Tan 和 Croft[38] 通过语法模式匹配和人工检查在 JDK 中发现了错误实例。Li 和 Tan[35] 设计实现了一个包含过程间数据流分析和静态污点分析的异常分析框架，提高了跨语言程序异常漏洞检查的精度。Li 和 Tan[36] 进一步提出了一个静态分析框架使得宿主语言的类型系统能够作用于外部接口抛出的异常，从而检查外部函数异常声明与实现不一致的问题。

异常处理的跨语言程序分析研究带有异常机制的高级语言和没有异常机制的低级语言互操作时的安全问题，尽管现有的工作主要关注以 Java 为宿主语言的情形，但是不同高级语言之间异常机制的设计往往是相似的，使得异常处理的跨语言程序分析具有较好的一般性。

4. 并发机制的跨语言程序分析

并发机制的差异影响着跨语言互操作。当使用外部接口与多线程的外部语言程序进行互操作时，根据宿主语言是非多线程的语言，如带有全局解释器锁（Global Interpreter Lock，GIL）的 Python，还是提供轻量级线程的语言，如支持协程 goroutine 的 Go，可能产生不同的影响。为了支持高并发，语言层面的轻量级线程和操作系统提供的本地线程之间一般是多对一的映射关系，但是发起外部调用的宿主语言线程则需要和本地线程一一对应。Marlow[91] 等人提出了兼容多路映射和一一映射的语义模型，在 Concurrent Haskell 和 Haskell FI 上实现了非阻塞的外部调用，支持多线程应用的调用和回调等特性。Li 等人[42] 分析 Java 对象被宿主线程和外部本地线程操作的情况，设计了一种通过求解需要在外部函数中加锁的 Java 对象以实现强制原子性的分析框架。

并发机制的跨语言程序分析初步研究了宿主线程和本地线程间的映射关系，以及 Java/C 多语言软件中的原子性问题，仍然缺乏足够的理论基础如单线程语言与并发语言互操作的语义模型，以及并发错误的分析总结与漏洞检查。

1.2.3　多语言软件的程序分析面临的挑战

多语言软件的程序分析需要考虑互操作双方语言特性的差异和外部接口的设计实现，虽然国内外的研究工作已经通过实证分析、形式语义、漏洞分类等研究说明了多语言软件存在更多安全问题，并且针对不同语言特性设计实现了跨语言的程序分析技术，但是多语言软件的程序分析仍然面临以下一些挑战：

（1）**外部接口的差异性影响了漏洞模式分析总结的有效性**。互操作语言的选择和互操作机制的设计受到新兴应用领域和编程范式的影响。Java 在服务器端编程中被大量使用，Java/C 多语言软件架构也得到了较多的研究，包括其漏洞模式的分类总结，以及针对 Java 的内存、类型、异常、并发等特性的跨语言程序

分析。然而，随着数据科学、深度学习等领域的流行，Python/C 的多语言软件架构被越来越多地使用，甚至成为了深度学习框架事实上的标准架构，是几乎所有主流框架的默认架构。但是不同于 Java 和 C/C++ 同样是编译型、静态类型的语言，作为宿主语言 Python 和 Java 在语言特性上的差异更大，这种差异一方面可能导致新的漏洞模式，一方面可能阻碍已有程序分析方法的应用。Li 等人[25]已经通过实证分析揭示多语言软件的易错性和语言的选择是相关的，在超过 20 组多语言组合中，Python/C 互操作导致的软件漏洞增多是最多的。以跨语言的内存漏洞为例，Python 使用引用计数算法[83]进行垃圾收集，Java 则使用标记清除算法[92]，这使得 Python/C 和 Java/C 多语言软件中跨语言的内存泄漏有着不同的漏洞模式和检查方法。语言特性的差异不仅使得多语言软件更加易错，也给分析总结漏洞模式带来了挑战，除了内存问题外，Python/C 多语言软件在其他语言特性上的问题仍未得到分析和检查。因此，分析外部接口的设计和使用、分析语言特性差异并总结潜在漏洞模式是理解并分析新兴多语言软件的安全性的基础。

（2）**缺少针对动态类型语言的外部函数的类型推断技术**。跨语言的类型转换是互操作的关键因素之一，但是跨语言的类型约束也是多语言软件中复杂且难以分析的性质。已有的工作主要关注静态类型语言之间的互操作，以 C/C++ 作为外部语言，Java、OCaml 等作为宿主语言，这些语言使用静态类型检查，程序分析工具在编译期能够直接得到显式的类型信息。然而，动态类型的宿主语言如 Python、JavaScript 等和 C/C++ 互操作时，类型系统的差异更大，导致跨语言的类型转换更加复杂，同时动态类型的宿主语言也会增大跨语言类型误用的风险。结合动态类型语言的类型推断和跨语言的类型分析，设计动态类型语言的外部函数的类型推断方法能够提高多语言软件的类型安全性和可靠性。

（3）**跨语言程序分析适用不同互操作语言和机制的泛化能力不足**。除了缺少对新兴的多语言软件架构的研究，跨语言的程序分析技术对于不同的互操作语言和外部接口也表现出较差的泛化能力，导致在异构的多语言软件上有较低的分析精度或需要重新设计实现跨语言的程序分析方法。Li 等人[25]已经通过实证分析说明除了互操作语言本身，互操作机制也会影响多语言软件的易错性。这也给程序分析方法针对不同互操作机制的泛化能力提出了挑战。同时，复杂的多语言软件可能混合使用不止一种互操作方法。例如，深度学习框架 PyTorch 除了基于 Python/C API 手工编写静态的跨语言接口，还使用了接口生成工具 pybind11。互操作语义研究为跨语言方法调用、类型转换等提供了一般性的抽象模型，但是如何将理论的形式模型应用到跨语言基础程序分析上仍然是一个挑战。增强跨语言程序分析技术的泛化能力能够提高在复杂多语言软件上的分析精度。基础程序分析技术的跨语言扩展能够回答跨语言程序的基本性质，其分析性质具有一般性，分析结果可以服务于多种程序分析任务。

1.3　主要研究内容

基于前述多语言软件的程序分析的研究背景与国内外研究现状，本文围绕三类关键问题展开，包括多语言软件的漏洞模式、复杂语言特性的的跨语言程序分析、跨语言基础程序分析。本文从特殊到一般地，从近年来大量使用但研究较少且易错的 Python/C 跨语言互操作入手，首先分析了 Python/C 互操作的外部接口 Python/C API 在 Python 编译器中的设计迭代和在 Python/C 多语言软件中使用行为，结合 Python 和 C/C++ 的语言特性差异分析总结了 9 类 Python/C 多语言软件的漏洞模式，设计实现了其中 6 类轻量级的漏洞检查工具。接着，针对其中需要精细设计的跨语言类型分析，本文基于形式化方法严格描述了 Python/C 跨语言程序的静态语义，以及 Python/C 外部函数的类型推断规范，设计实现了一种不依赖类型标注且没有误报的 Python 的 C 外部函数的静态类型推断方法，同时能够检查外部函数声明与实现不一致的漏洞。最后，本文更具一般性地设计了跨语言的调用图构建这一跨语言的基础程序分析技术，将外部函数类型推断的推理前提之一的外部函数声明抽象到语言泛化的外部映射语义，通过语言无关的图算法构建完整的调用图，支持多种宿主语言甚至是接口生成工具。

本文具体的研究内容和贡献如下:

（1）**Python/C 语言接口的提取分析与漏洞模式**。外部接口是编程语言互操作的编程接口，它的设计实现和使用模式反映了不同语言特性的协调交互，是理解互操作性、分析跨语言程序潜在漏洞的基础。以往多语言软件的漏洞模式分类主要关注静态类型的编译型语言 Java 和 C/C++ 之间的互操作。本文以语言特性差异更大且近年来广泛使用的 Python 和 C/C++ 互操作为研究对象，设计实现了其外部接口 Python/C API 的提取、分析、漏洞检查的工具链 PyCEAC。首先，PyCEAC 基于编译前端定制了预处理器、宏提取器、解析器，从不同版本的 Python 编译器实现中提取 Python/C API 定义，分析其随编译器迭代的设计变化。接着，PyCEAC 通过跨语言接口分离器、分词器、筛选器等部件从主流的 Python/C 多语言软件，包括 Pillow、NumPy、PyTorch、TensorFlow 等，提取并分析 Python/C API 的使用行为。然后，结合 Python/C API 的设计迭代、使用行为，以及对 Python/C 语言特性差异的分析，本文分析总结了 Python/C 多语言软件潜在的 9 类漏洞模式。最后，PyCEAC 通过语法模式匹配、集合运算、接入第三方工具的方法设计实现了其中 6 类漏洞的轻量级的漏洞检查工具，并在大量使用的 Python/C 多语言软件 Pillow、NumPy 等中发现了漏洞实例。

（2）**动态类型语言的外部函数的静态类型推断**。跨语言程序的类型误用作为 PyCEAC 总结的 9 类 Python/C 多语言软件漏洞之一，难以在语法层面进行有效的漏洞检查。同时，静态类型推断作为维护动态类型语言的安全性的基础技术，

也无法直接作用于包含外部函数调用的跨语言程序。为此，本文提出了一种不依赖类型标注的、确定性的 Python 的 C 外部函数的静态类型推断方法 PyCType。首先，PyCType 形式建模了 Python/C 跨语言程序的静态语义，包括抽象语法、类型系统和子定型规则。然后，PyCType 把外部函数类型推断的前提表示为三类可组合的假设判断，分别对应外部函数声明、参数类型转换、返回类型转换三类外部接口，根据其具体组合推断外部函数的函数签名。其中，外部函数声明只有一种形式，参数类型转换包含基于未使用参数分析的调用惯例分析和基于格式化串的参数解析分析两种形式，返回类型转换则包含四种不同的分析形式。最后，作为类型推断的副产物，PyCType 可以通过未使用参数分析中门限语义谓词[93]（gated semantic predicate）的真假检查外部函数声明与实现不一致的漏洞，该漏洞会导致无参外部函数可以接收任意类型的参数。在大量使用的 Python/C 多语言软件 Pillow、NumPy、CPython 标准库上的实验表明，PyCType 可以无误报地推理外部函数的类型签名并发现了多个一致性漏洞。同时在多个基于 Pillow 的主流应用程序上的实验表明，作为对 Google 的 Python 单语言静态类型推断工具 Pytype 的增强，PyCType 可以将其召回精度提高 27.5%。

（3）**支持多种宿主语言的跨语言调用图构建**。调用图构建是安全扫描、错误定位、代码跳转、补全与重构等多种程序分析任务的基础。然而，已有的工作往往针对单语言软件设计与实现，少量支持多语言软件的方法从外部语言出发或依赖专家知识，对于不同的宿主语言和互操作接口缺乏泛化能力。本文提出了一种支持多种宿主语言和外部接口的跨语言调用图构建方法 Frog。Frog 是一个两阶段的静态分析：离线的阶段一定义外部映射的语义模型，抽象描述不同互操作机制下的外部函数声明，提取跨语言的映射关系；阶段二基于语言无关的图变换与节点融合算法，在线构建完整的跨语言调用图。在 Python/C 的外部接口 Python/C API、接口生成工具 pybind11、JavaScript/C 的外部接口 Node.js C++ addons 上的实验表明，Frog 在具有泛化性的同时，取得了更好的精度，并保持了较好的时间效率。

本文主要的创新点与研究价值总结如下：

- 本文给出了已知的第一个 Python/C API 在 Python 编译器和主流 Python/C 多语言软件中的设计迭代和使用行为的实证分析，以及相对系统的 Python/C API 相关漏洞模式的分析总结。语言接口的提取和分析是多语言软件的程序分析的基础，本文对 Python/C API 的提取分析方法可以应用于其他编程语言互操作的语言接口，同时本文对漏洞模式的分析总结也可以供其他多语言软件的程序设计和分析工作参考。
- 本文提出了已知的第一个确定性的、不依赖类型标注的 Python 外部函数的静态类型推断方法，并基于该类型推断系统研究了外部函数声明及其实现

不一致的漏洞。本文提出的静态类型推断方法可以应用于其他以动态类型语言为宿主语言的多语言软件，服务于跨语言的类型误用的漏洞检查，以及多种需要类型信息的程序分析任务。

- 本文提出了已知的第一个语言泛化的语义模型以表示不同宿主语言和接口生成器的外部函数声明语言接口的语义，并基于语言无关的图变换和节点融合算法构建跨语言调用图。本文的方法可以支持不同宿主语言和 C/C++ 互操作的多语言软件，以及不同的跨语言接口生成工具，同时调用图构建作为基础程序分析可以服务于多种需要控制流信息的程序分析任务。

1.4　论文结构

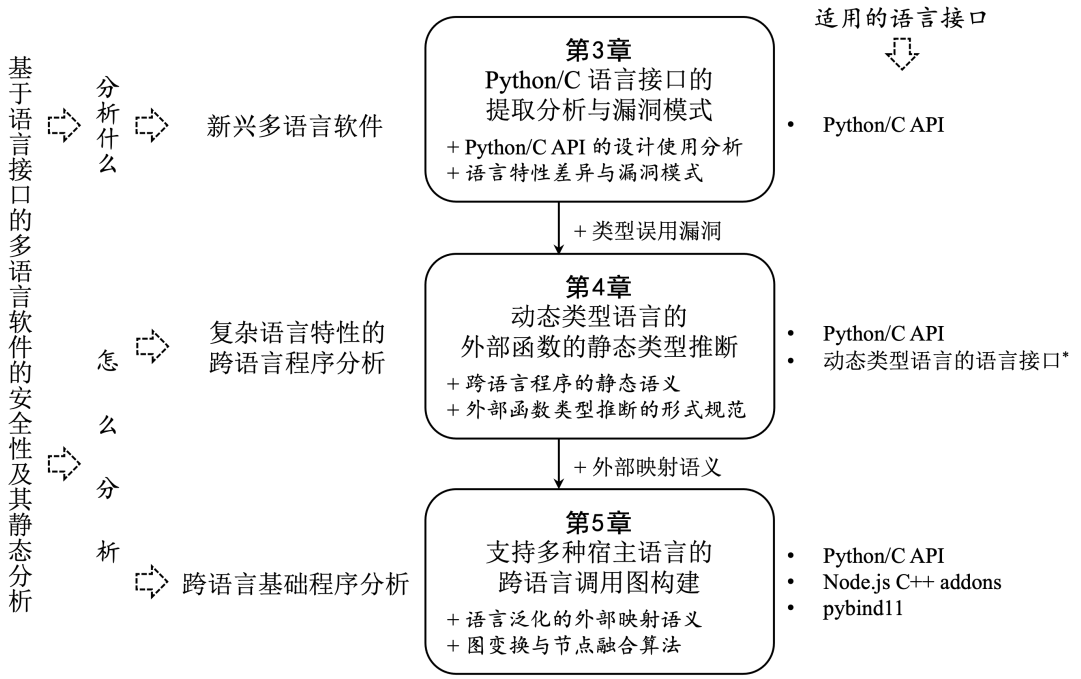


图 1.1　论文组织结构示意图

基于前述主要研究内容，本文的基本组织结构如图 1.1 所示。后续章节的主体内容包括:

第二章简要介绍编程语言互操作和程序分析两方面的背景知识与相关工作。

第三章研究 Python/C 语言接口的提取分析与漏洞模式。针对对新兴的 Python/C 跨语言互操作理解不足的问题，首先设计实现了 Python/C 的语言接口 Python/C API 的提取和分析工具，分析了 Python/C API 在 Python 编译器中的设计迭代和在主流多语言软件中的使用行为。接着结合语言特性差异分析总结了 9 类 Python/C 多语言软件的漏洞模式，并实现了其中 6 类漏洞的轻量级的检查工具，在大量使用的 Python/C 多语言软件 Pillow、NumPy 等中发现了漏洞。

第四章研究动态类型语言的外部函数的静态类型推断。类型误用漏洞是第

三章提出的多语言软件漏洞之一，其漏洞检查需要精细的设计，同时类型推断可以服务于多种需要类型信息的程序分析任务。针对类型这一复杂的语言特性，以及动态类型的宿主语言缺乏有效的跨语言类型分析方法的问题，第四章首先给出了支持 Python/C 跨语言互操作的静态语义，接着形式化地描述了外部函数的类型推断规范，其推理前提包含对外部函数声明、参数类型转换、返回类型转换三类语言接口的分析，最后通过在大量使用的 Python/C 多语言软件上的实验分析了方法的可靠性和完备性，通过对 Google 的单语言类型推断工具 Pytype 的增强实验分析了方法的有效性。

第五章研究支持多种宿主语言的跨语言调用图构建。外部函数声明作为第四章类型推断的前提之一只有一种形式，但是其在不同多语言互操作的语言接口中有着不同的语法和范式，同时调用关系作为基础的控制流信息可以服务于多种程序分析任务。针对跨语言的调用图构建这一基础程序分析难以支持不同的宿主语言和语言接口的问题，第五章首先分析了不同互操作机制下的外部函数声明，将其抽象描述为具有一般性的外部映射语义，接着把外部映射关系和调用子图通过语言无关的图变换与节点融合算法构建形成跨语言调用图，最后在基于 Python/C API、pybind11 和 Node.js C++ addons 的多语言软件上实验分析了方法有效性，并和基于专家知识、针对特定深度学习框架设计的外部映射提取工具 ffi-navigator 对比了分析精度。

作为本文的三点研究内容，第三、四、五章分别解决多语言软件的程序分析现存的某方面不足，同时相互之间呈依次递进的关系。第三章考虑语言接口的差异性，分析总结新兴的 Python/C 语言接口的设计和使用，以及 Python/C 多语言软件的漏洞模式。第四章考虑复杂语言特性的跨语言程序分析，针对第三章没有解决的类型误用漏洞，提出跨语言的类型推断方法，弥补了动态类型语言的外部函数的静态类型推断能力的不足。第 5 章考虑跨语言的基础程序分析，针对第四章中类型推断前提之一的外部函数声明的一般性，从特殊到一般地将其泛化表示为不同互操作方法中的外部映射语义，提出支持多种宿主语言的跨语言调用图构建方法。

第六章总结全文的研究内容，并展望未来可以继续开展的研究工作。

第 2 章　背景知识与相关工作

本章从两个方面介绍多语言软件的程序分析的背景知识与相关工作。首先介绍编程语言互操作的基本概念、发展历史、机制与实现，然后介绍程序分析的基本概念，包括分析对象、分析性质、评价指标，以及经典的静态程序分析技术。

2.1　编程语言互操作

2.1.1　基本概念

多语言软件是不同组件由不同编程语言编写的软件系统，本文尤其特指不同语言编写的模块之间存在跨语言互操作的非平凡的多语言软件系统。编程语言的互操作性是两个或更多编程语言在同一系统中交互的能力[94]。互操作性的设计原理一般可以归约到一对编程语言之间的互操作机制与实现，其核心是桥接互操作双方的外部接口（Foreign Interface，FI）。一般地，发起跨语言调用的语言被称为宿主语言（host language），被调用的语言被称为外部语言（foreign language）或宾客语言（guest language）。外部接口也被称作外部函数接口（Foreign Function Interface，FFI），但是两种表述下被调用的外部语言对象都不局限于函数。为避免误导、方便理解，本文统一地使用“外部语言”和“外部接口”进行描述。

2.1.2　发展历史

编程语言互操作在过去几十年间已经成为语言和系统设计的一个重要主题，可以追溯到语言发展的早期。Matthews[95]调研了编程语言互操作性的早期历史。1958 年，IBM 704 数据处理系统的 Fortran II 编译器[96]支持了子程序（subprogram）的概念，允许 Fortran 子程序和汇编语言子程序进行交互。1961 年，Burroughs 220 数据处理系统的 ALGOL 60 编译器[97]引入了 external 关键字。1967 年，IBM OS/360 的 ALGOL 60 编译器[98]设计了分离编译特性以混合 ALGOL 60 和 Fortran 程序。1979 年，Ada 正式将外部函数互操作写入语言标准[99]，标准定义了一个接口以允许 Ada 程序引入其他语言编写的函数，该接口声明外部函数的名字、参数类型和返回类型，并包含一个编译控制指令（pragma）以指明使用的外部语言。自此，主流的通用编程语言一般都会提供一种或多种编程语言互操作方法作为其语言标准的一部分，同时不断有新的方法和工具被设计提出以支持更安全、更简便的编程语言互操作。

2.1.3　机制与实现

在本文中，多语言软件指包含宿主语言程序和外部语言程序相互操作的系统，两者之间通过某种外部接口连接。如图 2.1 虚线上方部分所示，多语言软件可以分为三个部分：宿主语言程序、外部语言程序、基于外部接口编写的跨语言程序。跨语言程序包含宿主侧外部调用、外部侧被调对象求值结果返回，以及核心的接口层。接口层利用外部接口进行外部对象绑定、参数和返回类型转换等跨语言的处理。跨语言接口层可以看作对外部对象的一层封装，因此也常常被称作包裹（wrapper）。

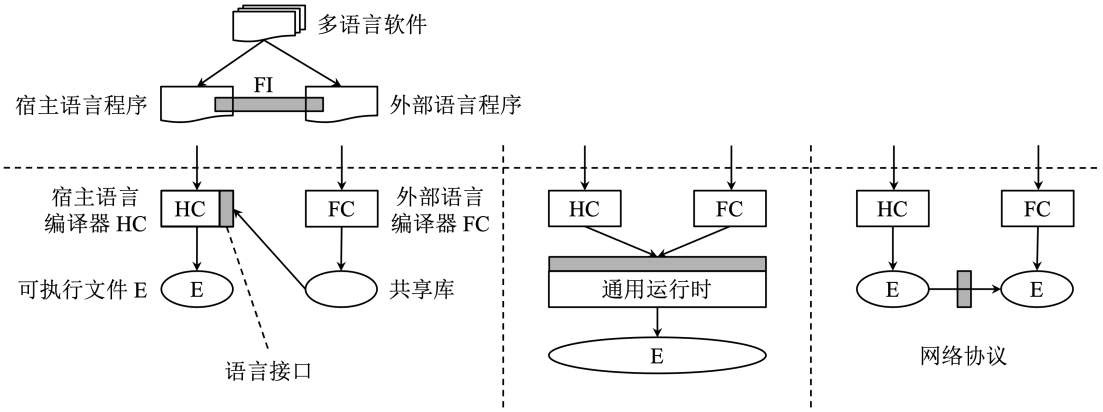


图 2.1　多语言互操作的不同机制

参考已有的研究工作[95,100-101]中的分类标准，多语言互操作在机制上可以分成以下三类：

（1）基于**语言接口**的多语言互操作。宿主语言定义语言接口以模拟编译后代码的调用惯例（calling convention），从而允许链接器把分离编译后的代码组合起来。

调用惯例是编译器对于调用过程的低层约定，包括参数和返回值的存储位置（栈帧、寄存器等）和放置方式（正序、倒序等）。语言接口可以解决不同调用惯例下的互操作问题，既包括不同语言之间的互操作，又包括单语言多编译器之间的互操作，比如使得新的程序能够复用旧的库的后向兼容性问题。C++ 标准委员会在 2014 年讨论了关于可移植的应用二进制接口（Application Binary Interface, ABI）的提案[102]，以解决同一平台上不同版本的 C++ 编译器编译后代码的链接问题和 C++ 编译器对其他语言代码的外部调用问题。由于外部接口在处理一些低级数据结构时可能需要硬编码，外部接口本身也面临着二进制兼容性的问题，导致外部代码需要针对不同平台和不同版本的宿主语言编译器重新编译。为了解决这一问题，Klock[103]提出了一种分层的语言接口以开发可移植的跨语言接口代码，Klock 将 Larceny Scheme 的外部接口分成三层，分别是 Larceny 运行时、依赖于 ABI 的外部接口提供目标架构和操作系统相关的功能、不依赖于 ABI 的

部分提供外部接口的其他功能。语言接口的一个重要设计问题是值的表示。例如，SML# 中 `real array` 类型的值和 C 中 `double[]` 类型的值有着完全相同的大小和对齐方式，这样就减少了跨语言类型转换的开销[95]。语言接口的另一个重要设计问题是协调不兼容的语言特性。例如，不同于 C 的急切求值策略，Haskell 采用惰性求值策略，因此语言双方被限制使用其自身的类型表示并在跨语言传值时进行必要的转换[104]。此外，传递到外部侧的宿主对象可能逃逸出宿主语言的自动垃圾收集机制，导致需要在跨语言接口代码中显式地管理对象的引用，带来了内存安全隐患[33-34,37]。

（2）基于**通用运行时**的多语言互操作。宿主语言编译器和外部语言编译器分别把两种语言的源码编译到由通用运行时定义的同一中间表示（Intermediate Representation，IR），从而共享通用运行时提供的高层和系统层级的特性，如垃圾收集（Garbage Collection，GC）、网络、线程与对应的锁机制等等。特殊地，当外部语言是宿主语言的嵌入语言（embedded language）时，通用运行时即宿主语言运行时，而外部语言编译器可能是一个源到源翻译器或共用宿主语言编译器。

通用运行时的设计核心是中间表示的设计，中间表示是编译器或虚拟机在代码变换和优化过程中使用的一种数据结构或低级中间语言。相关研究在不同语言特性的虚拟机平台上探索了对多种编程语言及其互操作的支持。微软的通用语言运行时（Common Language Runtime，CLR）提供了一种被称作通用中间语言（Common Intermediate Language，CIL）的中间表示。Dowd 和 Henderson[105]、Carlisle 等人[106]、Benton 等人[107]分别描述了逻辑编程语言 Mercury、命令式的编程语言 Ada、带有类型推断的函数式语言 SML 到 CIL 的编译。CLR 还提供了一个通用类型系统，Gordon 和 Syme[108] 形式描述了 CIL 及该通用类型系统在互操作时的类型安全性，Kennedy 和 Syme[109] 为该通用类型系统增加了参数化多态的支持，Yu 等人[110]进一步给出了范型的形式描述。Sun 公司提出的 Java 虚拟机（Java Virtual Machine，JVM）使用 Java 字节码（bytecode）作为其中间语言（称为 JVML）。Tan 和 Morrisett[111]通过在 JVML 中增加模拟 C 语言计算效果的原语，使得在 C 程序编译到 JVML 时能够复用已有的基于 JVML 的程序分析技术。Duboscq 等人[112]设计了图结构的 Java 字节码 Graal IR，以及基于 Graal IR 的即时（Just-in-Time，JIT）编译器 GraalVM。Würthinger 等人[113]提出了一种基于抽象语法树（Abstract Syntax Tree，AST）节点特化重写的虚拟机构建技术，使得 GraalVM 能够以更少的额外工作支持一门新的语言，该语言实现框架被称作 Truffle。Grimmer 等人的系列工作[67,114-115]基于 Truffle 为 GraalVM 增加了高效互操作的通用运行时并支持了 C、JavaScript 等多种语言。通过把外部 C 对象分配在由通用运行时管理的堆上，Grimmer 等人[87]设计了一种 C 程序缓存溢出、悬空指针解引用等漏洞的检查方法。需要指出的是，通用运行时提高了执行环境

的抽象层级，简化了互操作设计，但同时也限制了能够支持的语言特性，即通用运行时扩展支持的语言往往只是该语言的一个子集。例如，基于 JVM 的通用运行时使用 Java 的标记清除算法[92]进行垃圾收集，而 Python 的垃圾收集则基于引用计数算法[83]，将 Python 编译到 JVML 需要基于标记清除算法模拟引用计数垃圾收集[116]。

（3）基于**网络协议**的多语言互操作。宿主语言和外部语言通过共享的网络协议通信并交换数据流和控制流，两个语言的程序在不同的地址空间中作为独立的任务运行。

基于网络协议的多语言互操作源自分布式系统研究，是多系统互操作的一部分。代表性的工作包括远程过程调用[117-118]（Remote Procedure Call，RPC）、CORBA[119]、微软的 COM 与 DCOM[120]等。通过网络协议交互的两种语言的程序运行在不同的地址空间，甚至是不同的机器上，它们独立地维护各自的数据结构，这使得特性差异极大的语言也能轻松地实现互操作。然而，基于网络协议的多语言互操作也存在缺点。其一，在此机制下难以观测多语言软件中的全局不变量，这会影响许多程序分析算法的设计，例如需要使用分布式的垃圾收集算法。其二，每次外部调用都需要经过编码、通信、解码，以及上下文切换的过程，相较于同一地址空间中的本地外部调用而言存在昂贵的额外开销。

宿主语言、虚拟机或外部工具可以为同一对互操作语言提供不同机制的互操作方法及其外部接口设计。此外，从使用外部接口实现跨语言接口层的角度，也可以把多语言软件分成三类：

（1）**静态编码**的跨语言接口实现。可编程的外部接口允许开发者手工编写跨语言接口代码，显式地指明外部调用的构建方式。开发者需要自己处理对象绑定、参数和返回的值传递与类型转换、内存管理、异常处理等细节。静态编码的跨语言互操作有着更好的表达能力和性能，但是易用性较差，也导致跨语言接口编程复杂且易错[37-38]。

（2）**动态创建**的跨语言接口实现。编译器根据宿主侧的外部调用和外部侧被调对象的实现在运行时动态创建跨语言接口。在该方法下，外部接口在编译器内部定义并使用，包括预置的调用惯例和类型转换模板。动态创建的跨语言互操作简单易用，但是表达能力和灵活性较差，只能支持有限的语言特性，性能也较差且调优空间较小。

（3）**自动生成**的跨语言接口实现。辅助或自动生成静态编码的跨语言接口能够简化开发并提高安全性，包括根据外部程序如 C/C++ 头文件生成跨语言接口代码，或者通过接口定义语言（Interface Definition Language，IDL）提供更易编写的辅助接口。Beazley 等人[121]设计实现了 SWIG（Simplified Wrapper and Interface Generator），它能自动生成 C/C++ 与 Tcl、Python、Perl 等脚本语言之间的跨语言

接口代码。SWIG 基于 C/C++ 声明生成跨语言接口，支持大多数 C/C++ 数据类型，包括指针、结构体和类。Dimmich 和 Jacobsen[122] 设计了 SWIG 的 occam-pi 支持模块，occam-pi 是基于进程代数通信顺序过程[123]（Communicating Sequential Processes, CSP）和 π-演算[124]的语言，能够更好地表达并发。Reppy 和 Song[125] 的 FIG（Foreign Interface Generator）以 C 头文件和一个由开发者提供的描述脚本作为输入生成接口代码，额外的描述脚本允许用户设定外部库相关的跨语言转换策略。外部侧的多个低级数据结构和操作可能对应于宿主侧的一个高级特性，如数组参数、资源管理、多返回值等，Ravitch[126] 等人设计了一种基于程序分析的接口生成方法，通过识别外部代码对应的高级特性来生成更高效的 Python/C 跨语言接口代码。Aleksyuk 和 Itsykson[127] 通过 LibSL[128] 增强了基于 RPC 的跨语言互操作，使得接口代码能够自动生成。Zhu 等人[129] 提出了一种针对 Python 调用高性能 C/C++ 内核的跨语言接口生成技术，通过两阶段的程序分析首先静态生成对应的包裹类和函数，然后在运行时完成对象映射，从而生成尽可能少的外部绑定以提高性能。

表 2.1　多语言互操作的外部接口的机制与实现

机制 / 实现	语言接口	通用运行时	网络协议
静态编码	Python/C API Java Native Interface (JNI)† Node.js C++ addons		gRPC Python/C†
自动生成	cgo SWIG Python/C	WebAssembly JS/C	
动态创建	Python ctypes LuaJIT FFI OCaml/C Racket FI	C++/C GraalVM Python/C GraalVM JS/C	

† 提供辅助工具生成一部分的跨语言接口

表 2.1 列举了常见的一些编程语言互操作方法及其对应的机制与实现。同一对语言之间可以有多种互操作方法。例如，Python 的外部函数库 ctypes 提供了 C 兼容的数据类型。加载共享库后，开发者可以直接在 Python 程序中调用共享库中的 C 函数，由 ctypes 根据内置的模版完成参数和返回的传递与转换。Python/C API 则允许开发者访问 Python 编译器，开发者通过这些编程接口编写 Python 和 C/C++ 之间的跨语言接口，自定义对象映射和类型转换等处理。除了 Python 编译器提供的动态创建和静态编码的 Python/C 语言接口，第三方工具如 SWIG 可以基于 Python/C API 从简单的描述性文件自动生成跨语言接口代码。使用 C/C++ 服务端和 Python 客户端的 gRPC 也可以实现 Python/C 跨语言调用，它包括进程间通信以及外部调用的编解码。

不同的多语言互操作机制和实现在性能、安全性、可用性、表达能力等方面可能存在很大差异。以方便量化度量的性能作为示例，图 2.2 是表 2.1 中不同外

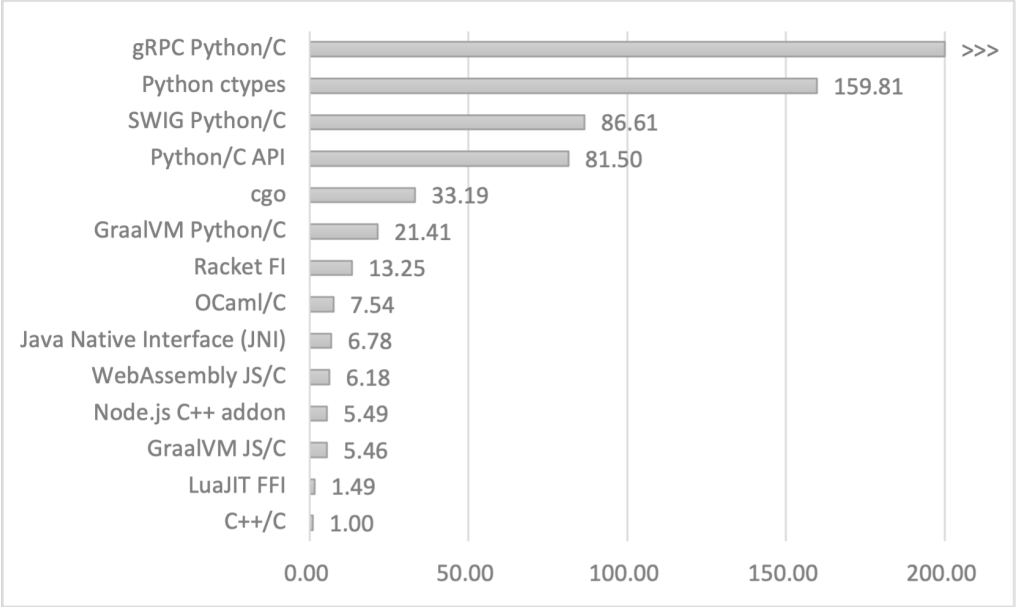


图 2.2　外部接口的性能度量

部接口执行 1 亿次空 C 外部函数调用的时间开销。图 2.2 以平凡的 C++ 调用 C 的情形作为基线，其他值是与基线耗时的比值。其中 gRPC Python/C 无法在有效时间内完成 1 亿次外部调用，因此使用了 >>> 标记代替。通过一百万次调用进行估计，gRPC Python/C 完成 1 亿次外部调用的时间约是基线的 175,000 倍。C++ 调用 C 几乎是无缝的，额外开销很小，可以视作通用运行时的一个特例，其中一种语言（大部分）是另一语言的子集。

由图 2.2 的实验可以看到，不同多语言互操作的外部接口间的差异较大，已有的针对某一对互操作语言的研究可能无法在其他新兴多语言软件架构中有效应用，尤其是在考虑多语言软件的某些复杂语言特性时，同时跨语言程序分析的泛化能力也将是一个挑战。

2.2　程序分析

程序分析指对计算机程序进行自动化的处理，以确认或发现某些程序性质[43]。程序分析的分析对象可以是源代码、二进制可执行代码，或者编译过程中的某种中间表示，如抽象语法树、控制流图（Control Flow Graph，CFG）等。对于不同的源语言，可能需要不同的程序分析技术，例如相较于 C 程序，Java 程序的分析需要处理面向对象的特性。对于同一种源语言，不同应用领域的软件也可能有不同的程序特性，如是否更多地使用指针、并发控制结构、数值计算等。程序的规模也可能从几百行到百万行不等。程序分析的分析性质也是多样的，可能关心性能、正确性、安全性等不同方面，分析结果用于编译优化、漏洞报警等不同任务。例如，大量出现指针和内存动态管理的程序的分析可能更关注空指

针、内存泄漏等问题，嵌入式程序可能需要处理中断，数值计算程序可能更关心溢出，并发程序可能更关心数据竞争，等等。分析对象的多样性和分析性质的复杂性使得程序分析具有较大的难度，Rice[48]已经证明了对于程序行为的任意非平凡属性，都不存在可以检查该属性的通用算法，即多数程序分析问题是不可判定的。在评价程序分析技术和工具时，需要综合考虑分析对象的特性与规模、分析结果的精度、分析过程的时空效率、分析性质可能造成的缺陷的危害程度等等。常见的精度指标包括可靠性、完备性等。程序分析系统是可靠的（sound），是指所有的分析结果都满足约束，例如所有能够通过漏洞检查的程序都不可能实际存在漏洞，即没有漏报（false negative）；程序分析系统是完备的（complete），是指所有满足约束的对象都可以分析得到，例如所有没有通过漏洞检查的程序都确实存在漏洞，即没有误报（false positive）。程序分析算法往往无法兼具可靠性与完备性，同时优化精度往往意味着可扩展性（scalability）、时空效率的损失，因此在设计实现中需要根据分析的对象和性质进行权衡。

一般地，程序分析可以分为静态程序分析与动态程序分析两类。静态分析对程序进行自动化的扫描和分析，而不必运行程序。动态分析则是利用程序运行过程中的动态信息，跟踪程序行为、分析程序性质。动态分析与测试类似，一般需要人为提供输入数据，对于大规模程序难以覆盖所有可能的路径，并且难以在理论上提供可靠性保证。作为和静态分析密切相关的方法，程序验证基于形式化方法严格证明程序具有某种性质，但是目前其可扩展性和分析效率离大规模实际应用还有差距。本文主要聚焦静态程序分析技术。静态分析既有相对严格的数学基础[130]，如基于格（lattices）的数学理论[131-132]，能够给出基于形式化方法描述的分析规范，可以基于抽象解释等数学框架提供可靠性保证[133-134]；又具有较好的实用性，能够结合实证研究等软件工程方法，并在漏洞检查等安全领域实际应用。经典的静态分析技术包括类型推断与检查[135]、各种数据流分析[136]（如常量传播分析、活跃变量分析、可达定义分析等）、指针分析[137-138]、过程间分析[139]、抽象解释[133]等等。这些程序分析技术已经广泛应用于社会生产实践中的软件安全分析。例如，抽象解释工具 ASTRÉE[44]成功应用于空客 A340、A380 等系列飞机飞行控制软件的自动分析。Reps 等人的 IFDS 分析框架[139]基于图可达性问题设计了过程间的数据流分析，已经被实现于 Soot[140]、Wala[141]等分析系统，并在 Android 污点分析[45]等方向广泛应用。

2.3　本章小结

本章分别介绍了编程语言互操作和程序分析，对基本概念进行了说明。在后续的章节中，本文将分析编程语言互操作可能带来的软件安全漏洞，对经典程序

分析技术进行跨语言的扩展，并基于程序分析的评价指标对本文提出的方法进行评估。

第 3 章　Python/C 语言接口的提取分析与漏洞模式

分析外部接口的设计和使用、总结多语言软件的漏洞模式是理解和分析多语言软件的基础。尤其对于新兴高级宿主语言而言，它们和常见的外部语言的语言特性差异更大，跨语言编程更加易错[25]，并且缺乏系统的研究。Python 是近年来最流行的编程语言[142]，丰富的库和扩展模块是其吸引开发者的一大因素。在数据科学和深度学习的浪潮下，Python 前端结合 C/C++ 底层实现的 Python/C 多语言架构兼具开发效率和执行性能，几乎已经成为了许多主流软件系统的标准架构。然而，互操作语言 Python 和 C/C++ 之间语言特性的差异，如异常处理、内存管理、类型系统等方面的不同，反应在 Python/C 互操作的语言接口 Python/C API 的使用中，使得基于 Python/C API 编写的跨语言接口程序潜在多种安全隐患。本章对 Python/C API 的迭代和漏洞模式进行实证分析。迭代分析包括 Python/C API 在 Python 编译器中的设计迭代，及其在主流软件中的使用行为。通过设计实现静态分析工具集 PyCEAC，本章揭示了 Python/C API 的设计迭代和使用行为，并给出了包含 9 种常见漏洞的漏洞模式总结。PyCEAC 在大量使用的图像处理库 Pillow 中发现了 48 个漏洞，其中 19 个是首次发现。PyCEAC 可以扩展接入基于语法的漏洞检查工具，同时本章对漏洞模式的系统分类总结可以指导构建更高精度、更自动化的漏洞检查工具。

3.1　引言

许多软件系统是多语言的，不同的组件使用不同的编程语言开发。多语言软件可以复用已有的代码并组合不同语言的优势。大多数编程语言都提供语言接口以和另一种语言编写的底层方法进行交互。互操作性与语言接口的设计给开发者带来了便利，并在工程实践中被广泛使用。然而，由于语言特性的差异，如内存管理[33-34]、异常处理[35-36]、类型系统[40,90]等的不同，编写安全可靠的多语言软件并不容易。

Python 是一种热门的编程语言，被广泛使用于许多领域，包括数据分析、网络开发、系统管理、机器学习等等。作为编程语言流行度的度量指标，Python 的 TIOBE 指数[142]在 2018 年超越 C++ 升至第三，在 2021 年超越 Java 和 C 升至第一并保持至今。Python 有着强大的生态和社区，提供丰富的库和扩展模块。结合其灵活的高层语法，Python 给开发者提供了极大的编程便利性和快速原型化的能力。另一方面，Python 的性能虽然饱受诟病[143-145]，但这一缺陷也可以通过将性能敏感的关键部件实现为 C/C++ 扩展模块来弥补。在数据科学和深度学习繁

荣发展的背景下，许多主流框架选择使用 Python 作为默认的前端编程语言，并使用语言接口 Python/C API 和底层的 C/C++ 外部函数进行交互。这种多语言的设计已经被越来越多的软件系统所采用。然而，由于缺乏对 Python/C 互操作的深入理解，以及语言接口 Python/C API 本身的复杂性和设计问题，跨语言接口编程可能面临许多潜在的安全隐患。

本章对 Python/C API 进行了一系列实证分析，从分析自 Python 编译器、主流 Python/C 多语言软件、开源项目漏洞实例等角度的证据出发，揭示 Python/C API 的设计迭代、使用行为、漏洞模式。首先，基于词法和语法分析，本章分析了 Python/C API 的规模、迭代、使用行为。然后，本章系统地从不同来源总结了 9 类漏洞模式，结合错误实例分析了这些漏洞在 Python/C 多语言软件中的表现。

本章设计实现了静态分析工具集 PyCEAC 来提取和分析 Python 编译器和 Python/C 多语言软件中的 Python/C API。静态分析能够覆盖所有可能的执行路径，使得 PyCEAC 能够有效地理解 Python/C API 的设计和使用。通过集成集合操作、语法模式匹配、第三方工具如 Clang 静态分析器（Clang Static Analyzer, CSA），PyCEAC 能够检查本章总结的 9 类漏洞模式中的 6 类。这些实证观测包括漏洞实例能够启发和指导高精度、高自动化的程序分析工具的设计。通过对 PyCEAC 给出的漏洞警告的人工检查，本章在广泛使用的 Python 图像处理库 Pillow（5.4.1 版本[146]）中发现了 48 个漏洞实例。其中 19 个是首次发现，另外 29 个曾有开发者报告过，但是这些漏洞报告往往都是应用执行过程中观察到的异常行为，缺乏系统的分析总结。32 个漏洞实例已被 Pillow 社区确认并修复，剩下的 16 个只在内部实现中使用，不暴露给外部开发者所以难以被触发，但是其对应的漏洞模式也被其他工作研究和确认[147]。同时，由于 Python 的特性，私有函数实际上是可以从外部访问到的。

从 2021 年至今 Python 一直是最热门的编程语言，流行的包中也包含大量 Python 和 C/C++ 互操作的多语言软件。但是关注 Python/C 互操作的外部接口的接口安全的工作却不多。本章希望能够为后续对 Python/C 多语言软件的理解和分析提供基础。本章的主要贡献如下：

- 本章设计实现了 PyCEAC 静态分析工具集，通过应用其 Python/C API 的提取和分析部分，本章分析了 Python/C API 在（1）Python 编译器大版本 2.7.0 至 3.8.0 中的迭代，（2）7 个不同领域的流行 Python/C 多语言软件系统中的使用行为，包括频率、热点、相似性等度量。
- 本章给出了已知的第一个相对系统的 Python/C API 相关的漏洞模式总结，包含 9 类漏洞模式。内存管理错误、缓冲区溢出、整数溢出、API 迭代兼容性是经典的漏洞模式，但是在多语言软件中有着不同的表现形式；异常处理错误、不完整的错误检查、类型误用是多语言软件所特有的；引用计数

错误、全局解释器锁（Global Interpreter Lock，GIL）错误是 Python/C 多语言软件所特有的。

- PyCEAC 的漏洞检查部分基于语法模式匹配等方法，检查 9 类漏洞中的 6 类。在 Pillow 上的实验发现了 48 个漏洞，其中 19 个为首次发现，部分漏洞已被社区确认并修复。进一步地，本章敏捷地将异常处理错误的漏洞检查扩展到代码规模更大的 NumPy 并发现了漏洞实例，一定程度上说明了漏洞模式总结的一般性和有效性，以及漏洞检查器的可扩展性。

3.2　研究动机

本节介绍基于语言接口 Python/C API 实现互操作的 Python/C 多语言软件的架构，讨论 Python 和 C/C++ 语言特性的差异，提出本章后续节希望回答的三个研究问题。

3.2.1　Python/C 多语言软件

PyTorch[14] 等多语言软件系统主要使用 Python 和 C/C++ 语言编写。宿主语言 Python 能够提高开发效率，外部语言 C/C++ 则支持高性能的库实现。为了兼容协调宿主语言和外部语言之间语言特性的差异，使用语言接口 Python/C API 编写跨语言接口代码是实现 Python/C 互操作最常见的方式。

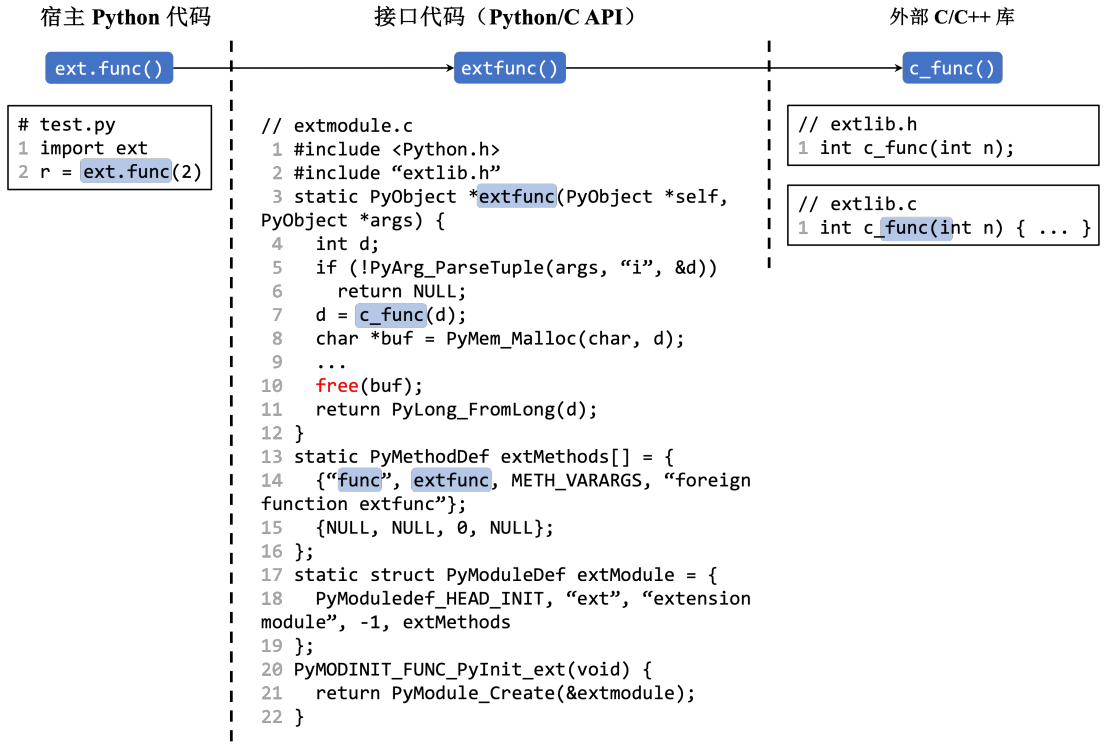


图 3.1　Python/C 扩展模块示例

图 3.1 是一个说明如何使用 C/C++ 模块扩展 Python 的例子。扩展模块 `ext`

在宿主侧提供外部函数 `ext.func` 供 Python 宿主程序调用。在跨语言接口层，扩展模块 `ext` 通过 `PyModuleDef` 类型的结构体定义（第 17 行），通过 `PyModule_Create` 创建（第 21 行）。扩展模块对象 `extModule` 包含一个 `PyMethodDef` 类型的数组 `extMethods`，其中存储着宿主 Python 程序可以调用的外部函数的元数据。例如，第 14 行中的元素声明了一个 Python 侧调用名为 `func` 的外部函数，它被映射到跨语言接口层的包裹函数 `extfun`（在第 3–12 行定义），并通过包裹函数调用 C 库函数 `c_func`。包裹函数除了调用外部库完成计算，还处理跨语言的值传递和类型转换。首先，来自 Python 侧的外部函数参数通过 `PyArg_ParseTuple` 被解析并存储到 C 类型的变量中。接着，转换后的参数被传给纯 C 函数完成计算并返回。最后，C 类型的返回又被转换为 Python 类型并返回给宿主侧，比如通过这里的 `PyLong_FromLong`。所有语言接口 Python/C API 都以 `Py` 或 `_Py` 开头，其中 `_Py` 开头的被用于 Python 编译器内部的语言实现，不应该在扩展模块中使用。

3.2.2　Python 和 C/C++ 的语言特性差异

Python 在流行编程语言中有着一定的的特殊性。本节将简要介绍 Python 相较于 C/C++ 在语言特性和内部实现上的一些不同之处。这些差异从根本上影响着 Python/C API 的设计，并且也是许多 Python/C 多语言软件漏洞模式的根源。

1. 解释执行

不同于 C/C++ 一般被编译为可执行的本地代码，Python 使用解释执行并且存在很多动态行为[148]。Python 的标准解释器 CPython 使用 C 语言实现，因此 Python/C API 也被用于语言的底层实现。

Python 在性能上有所妥协，没有像其他解释型语言如 JavaScript 一样采用即时编译等技术[149]，而许多提升 Python 执行性能的工作是以引入其他问题为代价的，例如可能破坏已有的扩展模块的兼容性[150]，或是只能支持 Python 语言一个子集[151-152]。

2. 动态定型

C/C++ 等具有显式类型的语言在编译期静态地检查程序的类型正确性，而 Python 使用动态类型检查，在运行时施加类型检查，并且允许变量在其生命期中改变类型。和静态定型相比，动态定型虽然提高了程序开发的灵活性和敏捷性，但是也增加了程序潜在类型错误的风险。同时，变量不需要先声明再使用，也增加了拼写错误、作用域混淆的风险。

鸭子定型[153]（duck typing）是讨论 Python 类型系统时会涉及的另一个术语。鸭子定型是和动态定型相关联的一个概念，相较于对象的类型，鸭子定型更关心该对象是否支持其所有的方法和属性调用。该名字来源于俗语：“如果它走路像

鸭子，叫声也像鸭子，那么它一定是一只鸭子”。Python 通过结构化子定型支持鸭子定型[154]。

类型误用是向外部函数调用传递参数时常见的错误[40,90]。动态定型增加了外部函数静态类型推断的困难，无法推理外部函数导致许多 Python 静态类型推断工具[155-157]难以提升其精度。因此，基于类型标注[158]或机器学习[159-161]的方法被提出以解决这一在经典程序分析技术中挑战性的问题，但是这些技术需要部分修改源码或是只能得到概率的推断结果。

3. 内存管理

C/C++ 使用显式内存管理，一个进程下的线程共享本地堆空间。而 Python 把对象分配在一个私有堆上，其在 Python/C 跨语言接口代码中支持如表 3.1 所示共 4 类内存分配器。

表 3.1　Python/C 跨语言接口代码可以使用的 4 类内存分配器

内存分配器	引入版本	默认版本	线程安全性	二进制兼容性	小对象优化
`malloc`	-	< 3.4	是	是	否
`PyMem_RawMalloc`	3.4	3.4, 3.5	是	是	否
`PyMem_Malloc`	3.4	⩾ 3.6	否	是	是
`PyMem_MALLOC`	3.4	-	否	否	是

Python/C 跨语言接口代码在动态内存管理上复杂且多样。因为 Python/C 跨语言接口代码本质上也是 C 程序代码, 所以可以使用系统内存分配器如 `malloc` 族函数。此外，Python 编译器从 3.4 版本开始引入另外三族可以用于动态内存管理的外部接口。`PyMem_RawMalloc` 族 Python/C API 是系统内存分配器的包裹函数，这些函数是线程安全的，在使用前不需要首先获取 Python 的全局解释器锁。`PyMem_Malloc` 族 Python/C API 模拟 C 标准，但是在请求分配零字节内存时做了特殊处理。该族函数用来在 Python 堆上分配和回收内存，并针对小对象进行了优化，但是在使用前必须先获取全局解释器锁。`PyMem_MALLOC` 族 Python/C API 是对应版本的默认 Python 内存分配器的宏，使用这些函数的扩展模块不具有对旧版本 Python 编译器的二进制兼容性。

Python 使用基于引用计数的垃圾收集机制[83]管理已分配的对象。每个 Python 对象都有一个 `ob_refcnt` 域，当它的值变成 0 时，垃圾收集器从私有堆上回收该对象的内存空间。相比其他形式的垃圾收集如标记-清除算法[92]，引用计数垃圾收集简单但是相对低效。

4. 全局解释器锁

Python 编译器引入全局解释器锁从而只允许一个线程拥有 Python 解释器，即在任何时候都只能有一个线程处于执行状态。引入全局解释器锁是为了避免同时操作一个对象的引用计数，而使用在线程间共享的数据结构锁又可能带来

死锁的问题和锁操作的额外开销，因此 Python 使用全局解释器锁作为单一的全局的锁。移除全局解释器锁可能破坏已有的 C/C++ 扩展模块，或是导致大量单线程和多线程 I/O 密集型程序的性能下降。Python/C 多语言架构被使用于许多异构并行系统，其中锁可能成为性能瓶颈[162]。作为弥补，Python 提供了多进程模块，每个进程拥有独立的解释器和内存空间。

3.2.3　Python/C API 的设计使用与潜在漏洞

前文图 3.1 构造了一个动态内存管理的漏洞，通过 `PyMem_Malloc` 分配的对象 `buf` 应当使用 `PyMem_Free` 回收，不匹配的内存管理可能导致程序发生运行时的崩溃。各类内存分配器不匹配都可能造成程序错误，尤其是当 Python/C 跨语言接口程序可以使用 4 族内存分配器。在跨语言接口程序中，不仅内存管理的差异可能导致潜在的安全问题，其他语言特性的差异都可能是危险的。本章对 Python/C API 的漏洞模式进行实证分析来总结相关的安全隐患（研究问题 3）。

由于 Python 编译器的版本迭代和大量不同领域的 Python/C 多语言软件的存在，本章通过实证分析揭示 Python/C API 的设计迭代（研究问题 1）和使用行为（研究问题 2）。首先，Python/C API 的迭代本身可能带来风险，导致扩展模块在其他版本的 Python 编译器中无法成功编译。其次，建模上千个 Python/C API 的行为并检查每一处使用的正确性是困难的，分析 Python/C API 在多语言软件中的使用能够揭示一个核心子集，并指导后续的漏洞检查设计。最后，为 Python/C API 的提取和分析设计的工具可以在漏洞检查工具中复用。

研究问题 1：Python/C API 在不同 Python 版本中的迭代。应用编程接口（Application Programming Interface，API）迭代可能是危险的，尤其是删除和修改 API。不具有后向兼容性的迭代可能引起程序崩溃等安全问题，导致重大的经济和技术负担[163]。Python/C API 随着 Python 编译器的发行不断迭代，主要以增加 API 为主。然而，Python/C API 变化的规模、版本之间主要发生了哪些变化、是否是安全的，这些情况仍然是未知的，同时也没有一个工具可以检查潜在的版本兼容性问题。

研究问题 2：Python/C API 在主流 Python/C 多语言开源项目中的使用行为。Python 和 C/C++ 语言特性的差异为 Python/C 多语言软件带来了安全隐患，但是在分析多语言软件性质或寻找漏洞时，想要检查所有的 Python/C API 是困难、开销巨大且不必要的。因此了解不同领域的 Python/C 多语言软件使用的 Python/C API 热点和核心子集十分重要，同时使用行为与漏洞模式及其检查也是紧密关联的。

研究问题 3：Python/C API 相关的常见漏洞模式。处理跨语言接口代码中 Python/C API 的使用的每个细节是困难的，有些使用方式可能和单语言编程的

经验差距巨大。Python/C API 的漏洞模式总结能够为开发者提供一个减少潜在错误的编程指南，同时也为设计特定漏洞的检查和修复工具提供指导。已有的对 Python/C 互操作的研究主要关注某类语言特性如垃圾收集[33-34]，而已有的多语言软件的漏洞模式分类主要针对 Java 和 C/C++ 互操作[38,80]，其在语言特性上和 Python/C 多语言软件有较大的不同。

对于漏洞的识别和修复，在本章的实证研究中更多的是希望以相对简单的方法构建漏洞检查工具并结合人工检查，主要目的是辅助发现本章总结的漏洞模式在实际流行工程项目中的存在和特征。

3.3　Python/C API 的提取和分析

回答研究问题 1 和研究问题 2 需要分别从不同版本的 Python 编译器源码和主流的 Python/C 多语言软件中提取和分析 Python/C API。本节首先描述静态分析工具集 PyCEAC 在 Python/C API 提取分析部分的设计实现，然后利用 PyCEAC 分析 Python/C API 的设计迭代和使用行为。

3.3.1　PyCEAC 的提取分析模块

扩展模块可以使用的所有 Python/C API 都以 `Py` 为前缀命名，并定义在 Python.h 及其引用或间接引用的头文件中。这些 API 中有一些是宏，有一些是函数，PyCEAC 需要识别这些 API 的定义、声明和调用。为了了解一个特定版本的 Python 编译器定义的 Python/C API 以及它们在 Python/C 多语言软件中的使用，PyCEAC 需要分别分析特定版本的 Python 编译器源码和给定的 Python/C 多语言软件。该分析基于编译前端技术，本节将描述 PyCEAC 的提取分析模块的设计和实现。

PyCEAC 的 Python/C API 提取和分析部分包含如图 3.2 所示左右两个部分。其中左边部分被设计用来从给定版本 i 的 Python 源码中提取 Python/C API 定义，右边部分则从给定的 Python/C 多语言项目 j 中提取并分析其对 Python/C API 的使用。

1. 伪定义头文件

在函数原型提取中，为了处理非 C 标准的类型和平台或版本特定的定义，PyCEAC 借鉴了 pycparser[164] 中伪定义头文件（faker headers）的想法。伪定义头文件中只包含编译解析对应文件所必须的原始定义，其他部分可以通过简化定义替代，这样一方面简化了静态分析的环境配置，一方面可以减少解析后中间表示的体积，从而减少分析的时空开销。

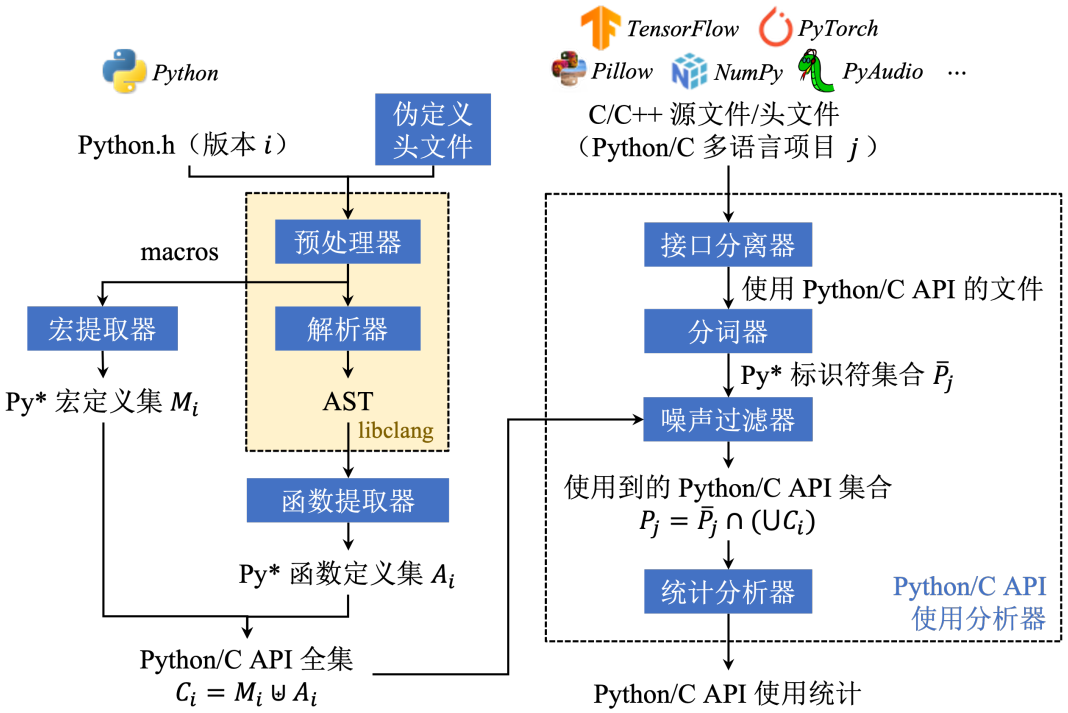


图 3.2　PyCEAC 工具集的 Python/C API 提取分析部分架构图

2. 宏提取器

宏提取器的目的是从 Python.h 及其引用和间接引用的头文件中提取命名以 `Py` 开头的宏定义。PyCEAC 使用 Clang 的预处理器解析输入的头文件，得到所有的宏定义，然后筛选出其中名字是 `Py` 开头的宏。Python 版本 i 中所有的 `Py*` 宏定义的分析结果被记作集合 M_i。Clang 是一个 C 族编程语言的编译器前端，支持包括 C、C++、Objective-C、OpenMP、CUDA 等编程语言。Clang 提供了一个编程接口 libclang[165]，使用 libclang 可以更简便地开发独立的分析工具。在 PyCEAC 中，该前端也可以使用其他编译器如 GCC，并通过编写 GCC 插件替代。

3. 函数提取器

函数提取器的目的是提取出 Python.h 及其引用和间接引用的头文件中所有的 Python/C API 定义。PyCEAC 使用 libclang 解析输入的头文件得到对应的抽象语法树（Abstract Syntax Tree，AST），然后通过一个扫描器遍历 AST 得到所有的 `Py*` 函数定义，记作集合 A_i。Python/C API 全集是 A_i 和 M_i 的并集，记作 C_i。

这里的函数定义包括一般的 C/C++ 函数声明、C++ 类方法、C++ 函数模板、C++ 构造器。在函数提取的分析过程中，PyCEAC 也会解析参数和返回的类型。伪定义头文件的使用可能会导致一些类型信息的损失，但是对 PyCEAC 更关心的函数名没有影响。

4. Python/C API 使用分析器

Python/C API 使用分析器包含多个模块，目标是提取给定的 Python/C 多语言项目 j 中的 Python/C API 并分析其使用行为。第一步，接口分离器把使用 Python/C

API 的文件从 Python/C 多语言软件 j 的 C/C++ 文件中分离出来，判定方法是是否直接或间接引用头文件 Python.h，这些文件组成的集合一定是跨语言接口代码的超集。第二步，基于 libclang 实现的分词器从使用 Python/C API 的文件中提取 `Py` 开头的标识符，将这些标识符组成的集合记作 $\bar{P}_j$。第三步，由于被分析的 Python/C 多语言软件可能会自定义一些名称以 `Py` 为前缀的噪声函数，这些噪声函数也包含在 $\bar{P}_j$ 中，噪声过滤器通过和 Python/C API 全集比较移除这些自定义函数，得到多语言软件 j 使用的 Python/C API 集合 P_j。最后，统计分析器对该 Python/C 多语言软件使用 Python/C API 的行为进行统计分析，计算包括频数、热点、相似度等信息。

3.3.2　Python/C API 在 Python 编译器中的设计迭代

通过使用 PyCEAC 的 Python/C API 提取分析部分，本节提取了 Python 编译器从 2.7.0 到 3.9.0 的大版本中定义和声明的 Python/C API。表 3.2 记录了提取和分析的结果，反映了 Python/C API 随 Python 版本的设计迭代。

表 3.2　Python/C API 在不同版本 Python 编译器中的迭代

版本	发行时间	使用率	API 函数	增加	删除	修改	API 宏
2.7.0	2010.07	5%	563	-	-	-	253
3.2.0	2011.02	2%	663	179	79	30	323
3.3.0	2012.09		702	115	76	2	275
3.4.0	2014.03		732	30	0	16	275
3.5.0	2015.09		751	19	0	2	287
3.6.0	2016.12	7%	764	13	0	9	291
3.7.0	2018.06	13%	804	43	3	6	297
3.8.0	2019.10	27%	844	55	15	16	300
3.9.0	2020.10	35%	862	30	12	16	300

表 3.2 的第 1、2 列分别是每个被分析的 Python 版本的版本号和发行时间。第 3 列则记录了特定 Python 版本的开发者使用份额，数据来自 JetBrains 公司 2021 年度的 Python 开发者调查结果[166]。PyCEAC 通过函数提取器提取了每个 Python 版本实现中扩展模块开发者可见的 Python/C API，并进一步地统计了 Python/C API 函数的总数，以及相邻版本间具体的增加、删除、修改 Python/C API 函数的数目，分别记录在表 3.2 的第 4–7 列。最后一列记录的是 PyCEAC 的宏提取器得到的 Python/C API 宏的数目。

具体地，Python 3.7.0 增加了新的 `contextvars` 扩展模块和一系列 `PyContext*` Python/C API 以支持上下文变量[167]。Python 3.8.0 增加了一系列 `PyConfig_*` Python/C API 用于 Python 初始化配置，以提供对环境配置和

错误报告更好的控制[168]。Python/C API 的修改包括参数和返回的类型声明变化和参数个数的变化。例如，Python 3.7.0 新引入的 `PyContext*` Python/C API 使用 `PyObject *` 作为其参数和返回类型①，它们在 Python 3.8.0 中被替换为精化的 `PyContext *` 类型。新增和删除 Python/C API 包括 API 的引入和废弃。例如，`PyInit_imp` 在 Python 3.3.0 被引入，又在 Python 3.7.0 被废弃。一个特例是 Python/C API 的重命名，一个 API 的重命名会被记作一个新增和一个删除。

研究问题 1：Python/C API 在不同 Python 版本中的迭代。

回答：从表 3.2 可以看到，每个 Python 版本大约有 1000 个 Python/C API，并且总数一直在随着版本更新而增加，这说明 Python/C API 在不断丰富以更好地支持 C/C++ 扩展模块。Python/C API 的删除和修改可能影响扩展模块的二进制兼容性，其数量在 Python 3 版本之初较多，但是从 3.4.0 版本开始就逐渐减少了，后续的更改主要以类型精化为主，不会破坏二进制兼容性，说明 Python/C API 的设计在逐渐趋于稳定。

Python/C API 的迭代可能是危险的。由于 Python/C API 的变化，NumPy 1.13 的某个开发版本出现了在新发行的 Python 3.7 上无法编译的问题[169]。当一个 Python/C API 在新编译器版本中被移除，使用旧 Python/C API 的扩展模块就会在新编译器中编译失败。类型声明的变化是两个不同版本的 Python/C API 的常见修改。例如，一些旧 Python/C API 的 `char` 类型的参数在新版本中变为 `wchar_t` 类型以支持宽字符。另一类出现在少数早期版本的 Python/C API 中的变化是参数数目的变化。当扩展模块根据旧的文档实现，其中某个 Python/C API 可能接收三个参数，该模块在新的版本中将编译失败，因为同样的 Python/C API 可能只接收两个参数。

大多数 Python/C API 的变化都来自某个 Python 增强提案（Python Enhancement Proposal，PEP），PEP 不断增加新的语言特性，提高语言的表达能力。虽然部分变化可能破坏后向兼容性，但是通过后续研究问题 2 和研究问题 3 中的 API 迭代兼容性漏洞检查将会看到，这些发生变化的 Python/C API 大多没有被主流的 Python/C 多语言软件所使用。

3.3.3　Python/C API 在多语言软件中的使用行为

为了理解 Python/C API 在实际 Python/C 多语言软件系统中的使用行为，本节选择了不同领域、不同规模的主流开源项目，其中外部 C/C++ 代码量不低于项目总代码行数的 20%。此外，为了了解 Python/C API 的使用在特定项目中的迭代变化，本节比较了频繁更新的项目在相隔超过一年的两个版本间的相似性。

表 3.3 记录了 17 个实验项目的基础信息及其 Python/C API 使用统计，其中第

①在本文中，* 表示指针类型时与其指向类型间有空格分割，* 表示通配符时和串的其他部分直接相连。

表 3.3　Python/C API 在 Python/C 多语言软件中的使用统计

软件	版本	发行时间	描述	KLOC	C/C++ 占比	Python/C API 种类	Python/C API 相似度	Python/C API 频数
Pillow	5.4.1	2019.01	图像处理	65.4	40.8%	124	77.8%	1047
	7.2.0	2020.06		74.6	40.7%	116		1107
NumPy	1.16.2	2019.02	科学计算	322.0	49.5%	274	82.1%	6166
	1.19.0	2020.06		353.4	46.8%	254		5907
TensorFlow	1.13.1	2019.02	机器学习	1989.2	50.2%	144	89.4%	1137
	2.1.1	2020.05		2391.2	59.4%	161		1322
PyTorch	1.0.1	2019.02	机器学习	687.2	55.1%	142	85.9%	1166
	1.5.1	2020.06		980.5	59.9%	161		1224
python-ldap	3.2.0	2019.03	目录存取	13.5	20.9%	63	96.9%	300
	3.3.1	2020.06		13.9	21.2%	65		330
scipy	1.7.0	2021.06	科学计算	2791.5	83.1%	104	98.1%	1472
	1.8.1	2022.05		2901.4	82.0%	104		1431
presto	2.2	2019.11	天体物理	143.9	67.4%	12	85.7%	219
	3.0.1	2020.02		145.9	66.1%	14		224
PyAudio	0.2.11	2017.03	音频 I/O	3.3	56.2%	40	-	339
python-krbV	1.0.90	2012.03	网络授权	4.9	86.4%	53	-	736
trace-cruncher	0.2.0	2022.03	内核跟踪	5.9	68.0%	40	-	233
hoep	1.0.2	2014.07	Markdown	9.5	59.5%	30	-	205
eigencu	-	2021.09	人脸识别	1.3	64.6%	17	-	155
gmpy	2.2.0.4	2014.10	多精度运算	33.0	63.9%	16	-	66
ecos-python	2.0.8	2021.01	圆锥求解	1.0	64.1%	17	-	195
haproxy	2.0.29	2022.05	TCP/HTTP	169.4	95.4%	31	-	202
amfast	0.5.3	2011.12	Flash	20.7	22.9%	74	-	852
isr_segawayrmp2	-	2018.06	拓扑导航	149.9	30.1%	50	-	303

7 列记录了对应的 Python/C 多语言项目使用到的 Python/C API 种类数，第 9 列则是这些 Python/C API 的总调用频数，数据使用 Python/C API 使用分析器（图 3.2 右半部分）收集。对于表中前 7 行记录的频繁更新的项目，第 8 列记录了这些项目前后两个版本使用到的 Python/C API 种类的 Jaccard 相似性[170]。对于一个项目 j，将其两个版本使用到的 Python/C API 集合分别记作 K_{j_1} 和 K_{j_2}，则 Jaccard 相似性指数 J_j 被定义为两个集合的交集与其并集的大小的比值：$J_j = \frac{|K_{j_1} \cap K_{j_2}|}{|K_{j_1} \cup K_{j_2}|}$。

进一步的分析可知，表 3.3 中 17 个项目的新版本共计使用了 370 个 Python/C API，远小于 Python/C API 总数（见研究问题 1）。表 3.4 列出了被表 3.3 中所有项目使用到的 6 个 Python/C API。`Py_DECREF`、`Py_XDECREF` 和 `Py_INCREF` 被用

表 3.4　所有项目共用的 Python/C API

Python/C API	描述
`Py_DECREF`	减少一个对象的引用计数，该对象必须非空。
`Py_XDECREF`	减少一个对象的引用计数，该对象可能为空。
`Py_INCREF`	增加一个对象的引用计数。
`PyArg_ParseTuple`	把外部函数的参数（Python 对象）解析为外部变量（C 对象）。
`Py_BuildValue`	把 C 对象解析为 Python 对象。
`PyTuple_New`	创建一个指定长度的 Python 元组对象。

来管理外部对象的引用计数，以避免引用计数错误（将在后续研究问题 3 中介绍，见第 3.4.7 小节第 1 小小节）。`PyArg_ParseTuple` 处理外部函数的参数类型转换，把 Python 类型的参数转换为 C/C++ 类型的对象。相反地，`Py_BuildValue` 把 C/C++ 类型的对象转换为 Python 类型的对象，常用于构建外部函数的返回值，并基于元组类型在 C/C++ 中支持 Python 的多返回特性。`PyTuple_New` 返回一个 Python/C 跨语言接口代码中常用的元组对象。

表 3.3 中每个项目使用到的 Python/C API 种类数都远小于 Python/C API 总数。多数开发者是一种或多种单语言程序的专家，但是往往对复杂的跨语言接口编程并不熟悉，在使用外部语言模块扩展宿主语言程序时，开发者一般倾向于根据外部接口参考手册中的基本用法实现跨语言接口的必需功能。图 3.3 可视化了 Python/C 多语言项目的代码量和使用到的 Python/C API 的种类数的关系，以及同一项目两个版本使用的 Python/C API 种类数的距离。菱形块标记的蓝色实线代表旧版本，方形块标记的橙色实线代表新版本。为了平滑单个项目的波动、突出长期趋势，绿色虚线使用滑动平均[171]计算预测曲线。对于序列 $\{x_0, x_1, \cdots, x_{n-1}\}$，其三周期滑动平均序列为 $\{-, -, \frac{x_0+x_1+x_2}{3}, \cdots, \frac{x_{n-3}+x_{n-2}+x_{n-1}}{3}\}$。如绿色虚线所示的趋势，随着代码量从上千行增加到几百万行，使用的 Python/C API 种类数经历了一个大约从 30 到 150 的上升阶段，最终达到一个不到 200 的峰值。这一趋势说明使用到的 Python/C API 种类数不会随着代码规模的增加而不断增加。

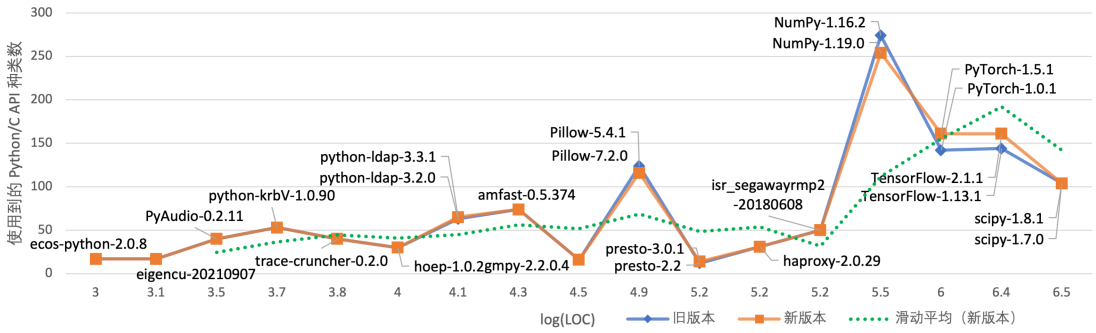


图 3.3　Python/C 多语言软件的规模和使用 Python/C API 种类数关系图

对于 7 个频繁更新的 Python/C 多语言项目，表 3.3 第 8 列计算了相隔约 1

年的两个版本之间使用 Python/C API 种类的相似度。为了减少干扰、放大迭代过程，图 3.4 进一步地把 4 个大量使用、密集更新的项目的版本间隔分成 5 个时间点，每个点相隔大约 4 个月。图 3.4 中的每条曲线代表一个项目，每个点上的标签分别是项目版本、使用到的 Python/C API 种类数、相比上一版本的相似性。由图 3.4 可以看到，除了 NumPy 的最后一个版本，其他所有点的相似度都高于 90%。NumPy 的最后一个版本 v1.19.0 简化了 Python/C 跨语言接口代码，使用到的 Python/C API 数有较大的减少。这个特殊点的变化也符合图 3.3 的趋势曲线，即考虑 NumPy 的代码规模比 TensorFlow 和 PyTorch 小大约一个数量级，它应该有一个相对更紧凑的跨语言接口代码。此外，从曲线反映出的每个项目的迭代趋势的平缓变化也可以看到，一个成熟的 Python/C 多语言项目使用到的 Python/C API 种类在其开发过程中倾向于保持稳定。

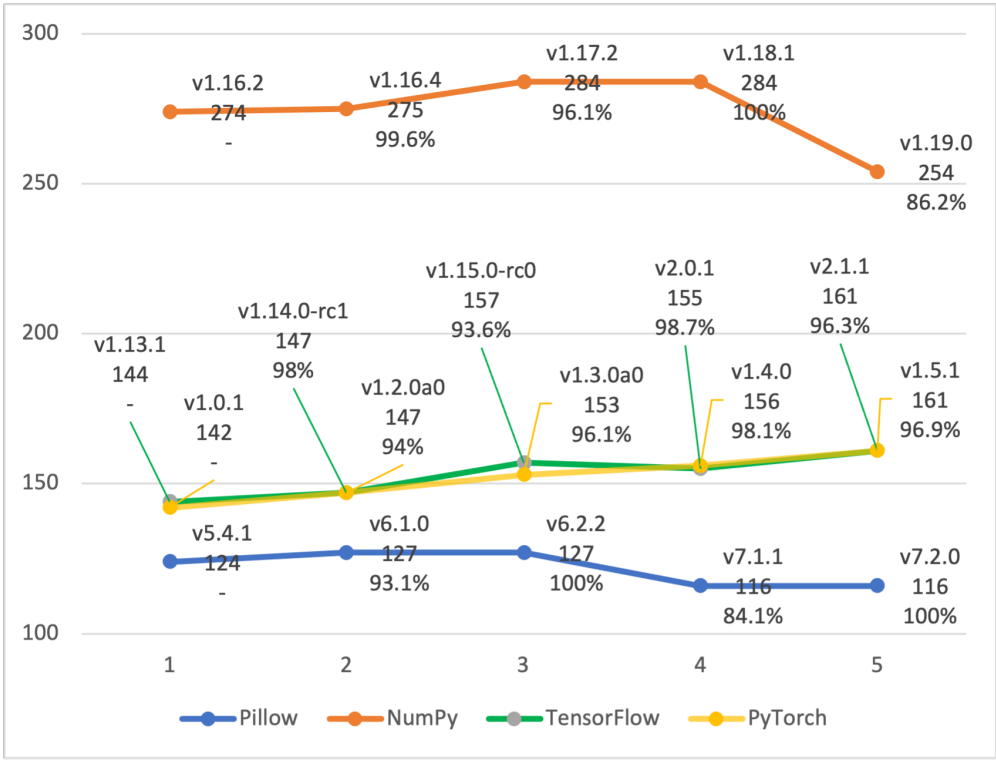


图 3.4　Python/C 多语言软件使用的 Python/C API 种类数随版本迭代

研究问题 2：Python/C API 在主流 Python/C 多语言开源项目中的使用行为。

回答：从表 3.3 和图 3.3 可以看出，一个 Python/C 多语言项目使用到的 Python/C API 种类数和其代码规模总体呈现正相关的趋势。但是当代码量达到一定规模，使用的 Python/C API 种类数不会继续增加，而是稳定在 200 左右，远小于 Python/C API 总数。

图 3.3 中两条实线间的距离和图 3.4 的曲线的变化趋势反映出一个项目所使用的 Python/C API 种类是相对稳定的，不会随着项目的开发迭代而大量变化。以大量使用的深度学习框架 TensorFlow 为例，即使项目规模达到 200 万行的量级，

在超过一年时间的密集更新中，Python/C API 使用种类的累积变化也没有超过 11%。

研究问题 2 的分析结果说明跨语言接口代码是实现特定功能的胶水代码，其复杂性连同漏洞模式不会随着代码规模的增加而持续增长。同时漏洞模式和漏洞检查器在部分项目上的有效性应当可以在更多项目及其迭代中得以保持。了解实际使用的 Python/C API 子集可以极大地减少漏洞检查器的检查范围，聚焦常见且关键的 Python/C API 的用法和正确性。

3.4　漏洞模式分类

了解了 Python/C API 的设计迭代和使用行为，接下来本节将关注 Python/C 跨语言接口代码的漏洞模式以回答研究问题 3。本节希望相对全面地覆盖常见的漏洞模式，其分析和总结的来源包括：

- Python 官方的 Python/C API 参考手册[172]
- 表 3.3 中的 Python/C 多语言项目的问题列表
- 针对多语言软件的已有研究
- 本章对语言特性差异的分析

一些漏洞模式归纳自针对多语言软件的已有研究，如异常处理错误与 Tan 和 Croft[38]针对 Java/C 互操作的总结相近，引用计数错误已由 Li 和 Tan[33]、Mao 等人[34] 分析过。Python/C API 参考手册分散地提及了一些在使用时需要注意的错误，如不完整的错误检查、整数溢出、缓冲区溢出、Python/C API 迭代兼容性。一些漏洞模式分析总结自 Python/C 语言特性的差异，如内存管理错误、GIL 错误、类型误用。本节搜索并分析了表 3.3 中 Python/C 多语言项目的问题列表，并使用漏洞检查得到的漏洞实例和对应的代码修复来说明每个漏洞模式。

为了说明漏洞模式的普遍性，本节应用漏洞模式总结和漏洞检查对 Pillow 5.4.1 进行实证安全研究。表 3.5 罗列了 9 类漏洞模式，以及基于本节所述漏洞模式实现的检查器在 Pillow 上发现的 48 个不同种类的错误，其中 19 个是首次发现①，剩余的 29 个已经有开发者发现，但是也符合本节的漏洞模式并可以被 PyCEAC 的漏洞检查器发现。

以下小节将结合实际工程的源码和漏洞检查的过程讨论每类漏洞模式的细节。

①此前在开源项目的问题列表（GitHub issues）中没有汇报过相应的漏洞或其对应的运行时异常行为

表 3.5　9 类漏洞模式及其静态检查

漏洞模式	静态检查器	漏洞数/首次发现/修复合并
异常处理错误	基于 grep 的脚本（A）	1 / Y / Y
不完整的错误检查	基于 libclang 的扫描器 + 基于 grep 的脚本（B）	16 / Y / N
内存管理错误	基于 CSA 的检查器（C）	2 / Y / Y
整型溢出	基于 grep 的脚本（D）	1 / N / Y
API 迭代兼容性	基于 grep 的脚本（E1）	28 / N / Y
	基于 libclang 的扫描器 + 集合运算（E2）	0
缓冲区溢出	基于 grep 的脚本（F）	0
引用计数错误	-	-
类型误用	-	-
GIL 错误	-	-

3.4.1　异常处理错误

Python 代码抛出异常时，Python 编译器会把控制传递给最近的 `try` 表达式并匹配异常类型进行异常处理。但是，外部 C/C++ 没有内置的异常处理机制，在 Python/C 跨语言接口代码中通过 Python/C API 抛出的异常不能立即中断外部方法的执行。只有被调外部方法执行结束，控制返回 Python 侧，Python 编译器的异常机制才能捕获由 Python/C API 抛出的异常。因此，在跨语言接口代码中需要显式返回以中断异常之后的控制流。同时在抛出异常和返回中间还需要正确处理已分配对象的引用计数，避免内存泄漏等问题。

处理异常的 Python/C API 包括两族，分别被称作异常和警告。异常族的 Python/C API 设置当前线程的错误指示器以抛出异常，而警告族的 Python/C API 仅向 `sys.stderr` 输出警告信息，并可以通过整型的返回值来表明一个警告是否为异常，返回 `-1` 抛出异常，返回 `0` 不抛出异常。PyCEAC 的漏洞检查器使用基于 grep 的脚本匹配上述两族共 27 个 Python/C API，通过基于模式的后处理进行漏洞检查。27 个 Python/C API 中部分是版本或平台相关的，使用一个较小的超集对于漏洞检查的副作用可以忽略不计。该静态检查器的后处理模式包括：

- 引用计数。如果一个异常处理表达式后紧跟引用计数操作，那么认为该异常处理是正确的，即假定抛出异常后立即操作引用计数一定是在做显式返回前的准备。
- `goto` 表达式。另一个后处理是关于“臭名昭著”的 `goto` 表达式[173]。分析异常处理相关 Python/C API 的使用可以发现，在 Python/C 多语言项目中普遍存在使用 `goto` 进行异常处理的情况。这也一定程度上反映出开发者对跨语言异常处理的无奈。

```
1 // Pillow/src/encode.c#L1065
2 if (...) {
3   PyErr_SetString(PyExc_ValueError, "...");
4   Py_DECREF(encoder);
5   return NULL;
6 }
```

图 3.5　异常处理错误漏洞实例

图 3.5 所示是 PyCEAC 的异常处理错误漏洞检查器发现的一个错误实例，应该加上第 5 行红色标记的代码以在抛出异常后把控制显式地返回 Python 侧，第 4 行在返回前将已分配对象的引用计数减 1。本文作者将该漏洞上报后，Pillow 官方确认了该漏洞并在 6.2.0 版本合并了修复[174]。

3.4.2　不完整的错误检查

除了显式抛出异常的 Python/C API，其他 Python/C API 和 C/C++ 常见的错误处理方式相似地通过返回值来标识内部是否发生错误，在对关联对象进行后续操作前应当检查对应的返回值。根据返回类型的不同，Python/C API 一般使用 `NULL` 或 `-1` 来标识错误。然而，一些 Python/C API 始终执行成功，如 `PyObject_HasAttr`；一些 Python/C API 使用 `0` 标识错误，如常用的进行外部函数参数类型转换的 `PyArg_Parse*` 族 Python/C API。

PyCEAC 使用基于 grep 的脚本检查这类漏洞，其实现相对复杂。该检查器首先使用了一个规则字典预置了上述三类错误标记（`NULL`、`-1`、`0`）和对应的 Python/C API。Python/C API 总数巨大，官方或第三方也没有已知的对 Python/C API 及其标识内部错误的返回值的总结。因此，基于前文研究问题 2 中对热点 Python/C API 的实证分析，该检查器仅对使用到的 Python/C API 子集进行检查。选择的 Python/C API、错误标识返回值、在 Pillow 中的使用频数信息如表 3.6 所示。

表 3.6　Pillow 使用的 Python/C API 热点及其错误标识返回值

错误标识返回值	Python/C API	Pillow 使用频数
0	PyArg_ParseTuple	182
NULL	Py_BuildValue	50
NULL	PyLong_FromLong	26
NULL	PyList_New	8
-1	PyDict_SetItemString	16
-1	PyDict_SetItem	1
-1	PyList_SetItem	1
-1	PyList_SET_ITEM	1
-1	PyTuple_SET_ITEM	1

不完整的错误检查的漏洞检查也包含两个启发式的假设：

- `if` 判断。如果要检查的 Python/C API 处于 `if` 表达式的条件测试中，那么假定该条件判断的目的是进行额外的错误检查。
- `return` 表达式。如果要检查的 Python/C API 处于外部函数的 `return` 表达式中，即使没有对该 Python/C API 进行完整的错误检查，SystemError 机制也将捕获可能的错误。然而，需要注意的是，SystemError 一般用于标识编译器内部错误，其错误信息可能不是用户友好的，难以用于定位和调试。

该漏洞检查中另一个问题是用户定义的宏。例如，Pillow 把 `PyLong_FromLong` 定义为 `PyInt_FromLong`，类似的处理在其他 Python/C 多语言项目中也是一种常见的行为。PyCEAC 基于 Python/C API 的提取分析部分的预处理器，在第一个分析趟（pass）进行漏洞检查的同时收集项目中的宏定义，并形成一个额外的规则字典，然后在第二个分析趟增量地分析对于这些宏的不完整的错误检查漏洞。此外，这一漏洞模式也适用于内存分配函数等库或系统调用，如 `malloc`。通过简单地增加目标规则，PyCEAC 就可以同时分析这些 API。事实上，该漏洞检查器确实发现了一个未检查的 `malloc`。作为一个特例，其本质是一个单语言问题，因此该漏洞发现没有记录在表 3.5 中。

人工确认该检查器的报错比较麻烦，仅考虑热点 Python/C API 及其错误标记并不足够，还需要考虑内部具体实现。例如，在

```
PyList_Append(..., Py_BuildValue(...))
```

中，`Py_BuildValue` 使用 `NULL` 标识错误，但是 `PyList_Append` 在其实现内部会检查传入的值是否为空，当传入值为空时则抛出异常，因此这里不需要对 `Py_BuildValue` 做额外的检查。

图 3.6 所示是一个在该类漏洞检查的报警中确认的漏洞，其中应当添加如第 3 行所示的错误检查。不同于 `PyList_Append`，`PyList_SetItem` 在内部不会对参数进行检查，这种行为容易使开发者感到迷惑，进而导致漏洞或不确定的行为。

```
1 // Pillow/src/_imaging.c#L2004
2 PyObject* item = Py_BuildValue("iN", v->count, getpixel(
    self->image, self->access, v->x, v->y));
3 if (item == NULL) {...}
4 PyList_SetItem(out, i, item);
```

图 3.6　不完整的错误检查漏洞实例

另一类特殊的 Python/C API 如 `PyDict_SetItem` 在内部使用断言而非异常来保证输入值非空。考虑到断言的设计目的和运行时行为，这类 Python/C API 仍应进行额外的错误检查，根据这一模式 PyCEAC 还发现了 15 个漏洞。

3.4.3　内存管理错误

如前文（第 3.2.2 小节第 3 小小节）对 Python 和 C/C++ 内存管理系统的差异的介绍，Python/C 跨语言接口代码可以使用 4 族内存分配器。在每一族内存分配器内部都存在三类典型的内存管理错误:(1)内存泄漏,(2)多次回收,(3)悬空指针解引用。同时，在每一族内存分配器内部和不同族内存分配器之间都可能存在不匹配的错误。例如，通过 `PyMem_Malloc` 分配的内存应当使用 `PyMem_Free` 回收，使用 `PyMem_Del` 或 `free` 都会导致多语言软件崩溃。

在 Python 编译器内部实现的源码和 Python/C API 的参考手册中甚至对内存管理存在不一致的叙述。图 3.7 是 Python/C API 参考手册中的一个内存管理错误示例，它说明通过 `PyMem_New` 分配的 `buf1` 应当使用 `PyMem_Del` 回收。然而，根据如图 3.8 所示 Python 源码中注释的说明，图 3.7 中的 `buf1` 应当使用 `PyMem_Free` 回收。虽然不像不同族内存分配器的不匹配可能导致严重的错误，但是这种行为仍然让开发者感到迷惑，也不利于编码的一致性。

```
1 char *buf1 = PyMem_New(char, BUFSIZ);
2 char *buf2 = (char *) malloc(BUFSIZ);
3 char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
4 ...
5 PyMem_Del(buf3); /* Wrong -- should be PyMem_Free() */
6 free(buf2);      /* Right -- allocated via malloc() */
7 free(buf1);      /* Fatal -- should be PyMem_Del()  */
```

图 3.7　Python/C API 参考手册中的内存管理错误示例

```
1 /* Include/pymem.h
2  * PyMem{Del,DEL} are left over from ancient days,
3  * and shouldn't be used anymore.
4  * They're just confusing aliases for PyMem_{Free,FREE}
     now.
5  */
6 #define PyMem_Del   PyMem_Free
7 #define PyMem_DEL   PyMem_FREE
```

图 3.8　Python 源码对内存管理 Python/C API 的说明与定义

Clang 静态检查器（Clang Static Analyzer，CSA）是分析 C、C++、Objective-C 程序漏洞的静态程序分析工具集。其 unix 系列检查器检查 POSIX/Unix API 的使用，包括 unix.Malloc 检查内存泄漏、多次回收、回收后使用（use-after-free）和 `malloc` 的偏移量等漏洞，unix.MismatchedDeallocator 检查不匹配的内存回收。PyCEAC 通过一个简单的脚本调用 CSA 的检查器，并处理头文件依赖和目标文件遍历等辅助操作。

使用该漏洞检查器，PyCEAC 发现了如图 3.9 所示两个漏洞实例。进入第二

```
1  // Pillow/src/libImaging/Resample.c#L622
2  ksize_horiz = precompute_coeffs(
3    imIn->xsize, box[0], box[2], xsize,
4    filterp, &bounds_horiz, &kk_horiz);
5  if ( ! ksize_horiz) {
6    return NULL;
7  }
8
9  ksize_vert = precompute_coeffs(
10   imIn->ysize, box[1], box[3], ysize,
11   filterp, &bounds_vert, &kk_vert);
12 if ( ! ksize_vert) {
13   free(bounds_horiz);
14   free(kk_horiz);
15   free(bounds_vert);
16   free(kk_vert);
17   return NULL;
18 }
```

图 3.9　内存管理错误漏洞实例

个 `if` 条件判断意味着 `*_horiz` 对象已经创建成功，而 `*_vert` 对象则没有，第 15、16 行回收为空的对象会导致程序崩溃。本文作者对该漏洞的反馈已被确认，其在 Pillow 7.0.0 中被修复[175]。

3.4.4　整数溢出

Python 2 在内部实现时区分 `int` 和 `long` 类型的整数，但是 Python 3 则不再使用 `int` 类型。抛开版本系列和编译器实现细节，Python 程序员早已习惯了长整型的支持。在使用 `PyLong_AsLong` 等 Python/C API 时，开发者可能不会想起 C/C++ 中令人困扰的整型精度损失和溢出，导致参数跨语言传递时发生整数溢出错误。事实上，从版本 3.2 开始，Python 编译器提供 `PyLong_AsLongAndOverflow` 等 Python/C API 以进行额外的检查。

另一方面，进行外部函数参数解析的 Python/C API 如 `PyArg_Parse*` 使用格式化串来指明跨语言的类型转换，这些 Python/C API 也可能引起整数溢出的漏洞。存在安全隐患的格式化字符及其含义如表 3.7 所示。

表 3.7　需要检查整数溢出的格式化字符

格式化字符	Python 类型	C 类型
B	integer	unsigned char
H	integer	unsigned short int
I	integer	unsigned int
K	integer	unsigned long long
k	integer	unsigned long

除 b 以外的所有格式化字符在转换到无符号整型时都不进行溢出检查，而 b 也要求传入外部函数的 Python 整型非负。PyCEAC 的漏洞检查器会匹配格式化串中包含 b 或表 3.7 中所有格式化字符的 PyArg_Parse* Python/C API。通过对这些潜在风险的调用点的人工检查，本文作者发现了如图 3.10 所示的漏洞实例，该漏洞已被确认并在 Pillow 6.1.0 中被修复[176]。

```
1 // Pillow/src/encode.c#L997
2 if (!PyArg_ParseTuple(args, "ss|OOOsOIOOOssi", &mode, &
     format, ...))
3   return NULL;
```

图 3.10　整数溢出漏洞实例

3.4.5　API 迭代兼容性

Python 有着活跃的开发社区，语言本身在不断地更新，并且存在 Python 2 和 Python 3 两个并不完全兼容的版本系列[177]。根据 JetBrains 公司 2019 年度的 Python 开发者调查[178]，不同 Python 版本都有较多开发者正在使用（如表 3.2 第 3 列所示）。由于数据结构定义的变化等原因，Python/C API 并不是二进制兼容的，增加新的域或修改域的类型都可能破坏二进制兼容性，导致开发者需要针对不同 Python 版本重新编译外部模块。从 Python 3.2 开始，一个 Python/C API 子集被声明为稳定的应用二进制接口（Application Binary Interface，ABI）。在使用该子集时（在引入 Python.h 后定义 Py_LIMITED_API 宏），一些编译器的运行时细节对外部模块不再可见，但是也不需要针对不同的编译器版本重新编译外部模块。

新增 Python/C API 一般不会引发兼容性问题，但是如果跨语言接口代码使用到的 Python/C API 在后续版本中被删除或修改，那么外部模块可能会编译失败。前文第 3.3 节研究问题 1 和研究问题 2 分别分析了 Python/C API 在 Python 编译器大版本中的设计迭代，以及在主流 Python/C 多语言项目中的使用行为。需要注意的是表 3.2 中的修改主要来自类型声明的变化，例如将类型声明由 char 改为 wchar_t 以支持宽字符，分别使用 Py_UCS4、Py_UCS2、Py_UCS1 替代 32、16、8 位无符号整型，等等。在编译扩展模块时，这类 Python/C API 修改导致的类型转换只会以警告的形式出现，不会导致编译失败。

PyCEAC 基于集合操作实现了一个 Python/C API 迭代兼容性的检查工具，给定 Python/C 项目源码和 Python 编译器版本，检查该项目是否使用了对应版本不支持的 Python/C API。主流项目经过来自不同平台和版本的大量用户的编译构建，人工检查也确认了该检查器发现的警告都是误报，如图 3.11 所示是一个误报实例，主流项目通常会使用宏进行不同版本的分支控制。但是对于个人开发

者，其可能缺少跨语言接口编程的经验和完整的多版本测试，导致这样的处理常常被忽略，并且可能在代码开源后收到大量的错误反馈，该检查器可以为这类小型项目提供有效的辅助。

```
// Pillow/src/_webp.c#L851
#if PY_VERSION_HEX >= 0x03000000
PyMODINIT_FUNC PyInit__webp(void) {
  ...
  m = PyModule_Create(&module_def);
  ...
}
#else
PyMODINIT_FUNC init_webp(void) {
  PyObject* m = Py_InitModule("_webp", webpMethods);
  ...
}
#endif
```

图 3.11　API 迭代兼容性漏洞误报实例

此外，还有一类值得注意的类型声明变化。官方文档指出，在跨语言接口代码中用整数索引 Python 序列对象时，应该使用 `Py_ssize_t` 替代 C 的 `int` 类型。前者和编译器定义的 `size_t` 类型长度相同但是是有符号的。使用 C `int` 时，被索引的序列不能使用完整的地址空间，在 64 位机器上被限制不超过 2^{31} 个元素[179]。在未来将会禁止使用 C `int` 索引 Python 序列。事实上，使用 Python 3.8 编译 Pillow 已经会产生过期（deprecated）编译警告。这一漏洞模式也和整数溢出中介绍的格式化字符相关联。格式化字符 `i` 将 Python 整型转换为 C `int` 类型，而格式化字符 `n` 则转换到 `Py_ssize_t` 类型。PyCEAC 使用和整数溢出检查器相似的方法检查这类漏洞。图 3.12 是发现的一个漏洞，相似的漏洞在 Pillow 中共发现了 28 个，这些漏洞在 6.0.0 版本被修复[180]。

```
// Pillow/src/encode.c#L128
int bufsize = 16384;
if (!PyArg_ParseTuple(args, "|i", &bufsize))
  return NULL;
```

图 3.12　API 迭代兼容性漏洞实例

3.4.6　缓冲区溢出

Python 对于数组会进行自动的边界检查，但是通过 Python/C API 传递的值仍然可能在外部函数中触发缓冲区溢出的错误。因此，额外的边界检查对于 Python/C 跨语言程序仍然是需要的。此外，在 Python 中索引可以是负数，例如使用 `-1` 索引最后一个元素。这种行为外部语言模块也无法支持。外部侧危险的操作包括 `strcpy`、`strcat`、`memcpy`、`fscanf`、`malloc`、`calloc`。

涉及索引的 Python/C API 如 `PyList_GetItem` 在内部实现中会进行边界检查，但是其宏形式如 `PyList_GET_ITEM` 则不会。相似的 Python/C API 宏包括 `PyList_SET_ITEM`、`PyTuple_GET_ITEM`、`PyTuple_SET_ITEM`。该漏洞的静态检查器没有在 Pillow 中发现此类漏洞。

3.4.7　其他漏洞模式

下面接着描述其他三类涉及复杂语言特性的漏洞模式，本章仅介绍其漏洞特征，对应的漏洞检查已有相关的工作或需要精细的设计。本章以覆盖更多漏洞模式为研究目标，因此不对复杂语言特性的程序分析方法加以展开。本文将在下一章中给出其中类型误用漏洞的程序分析方法。

1. 引用计数错误

前文在 Python 和 C/C++ 的语言特性差异中介绍了 Python 基于引用计数算法的垃圾收集机制（第 3.2.2 小节第 3 小小节）。不幸的是，外部对象不在垃圾收集的管理范围内。在通过 Python/C API 操作 Python 对象时，其引用计数不会自动进行调整，而是需要开发者在跨语言接口层手动地处理，即通过 `Py_INCREF`、`Py_DECREF` 等 Python/C API 显式地增减引用计数。考虑到借引用、偷引用等特殊情形[84]，这种管理是易错的，可能导致严重的内存问题。Li 和 Tan[33]、Mao 等人[34] 设计不同的程序分析方法实现了引用计数错误的静态检查器。

事实上，前述异常处理错误的示例图 3.5 中第 4 行的 `Py_DECREF` 操作常常被忽略。PyCEAC 对异常处理错误的一个假设是，抛出异常后的引用计数操作后面总是跟着返回语句。但是另一方面，在返回语句前，正确地处理所有对象的引用计数并不容易。

2. 类型误用

如前文 Python 和 C/C++ 的类型系统差异所述（第 3.2.2 小节第 2 小小节），动态定型的宿主语言增加了类型误用的风险。Python/C API 包含一系列类型转换的语言接口，比如 `PyLong_AsLong` 把传入的 Python 整型对象转换为 C 中 `long` 类型的整数对象。由于 Python 不是静态定型的，传入的值的类型检查本身就是一个问题，传入一个不匹配的类型的值可能导致错误或非预期的行为。这类漏洞的检查需要精细的设计，包含大量独立的工作，本文将在下一章介绍一种基于类型推断的技术以分析外部函数的类型约束。

3. GIL 错误

如前文语言特性差异中对 Python 特殊的全局解释器锁的介绍（第 3.2.2 小节第 4 小小节），扩展模块可能是非线程安全的，当前线程在操作 Python 对象时需要获取全局解释器锁。在 Python 中使用 threading 等模块创建线程时，线程状态会自动和线程对象关联。但是当外部模块创建线程时，这些本地线程不拥有全

局解释器锁和线程状态信息。出于这一原因，部分 Python/C API 必须在获取全局解释器锁之后才能安全地使用。考虑多语言互操作的并发语义缺少理论基础[77]，Python/C API 的并发安全性也没有完整的总结，本章没有深入研究该漏洞模式并设计其静态检查器。

3.5　讨论

本节首先概览 PyCEAC 静态分析工具集的漏洞检查器部分，这些静态检查器的设计服务于上一节中漏洞模式的介绍。本节将讨论这些轻量级的漏洞检查器的局限性，并介绍一些未来的方向，如构建自动化程序更高、精度更好的检查工具。

3.5.1　PyCEAC 的漏洞检查器

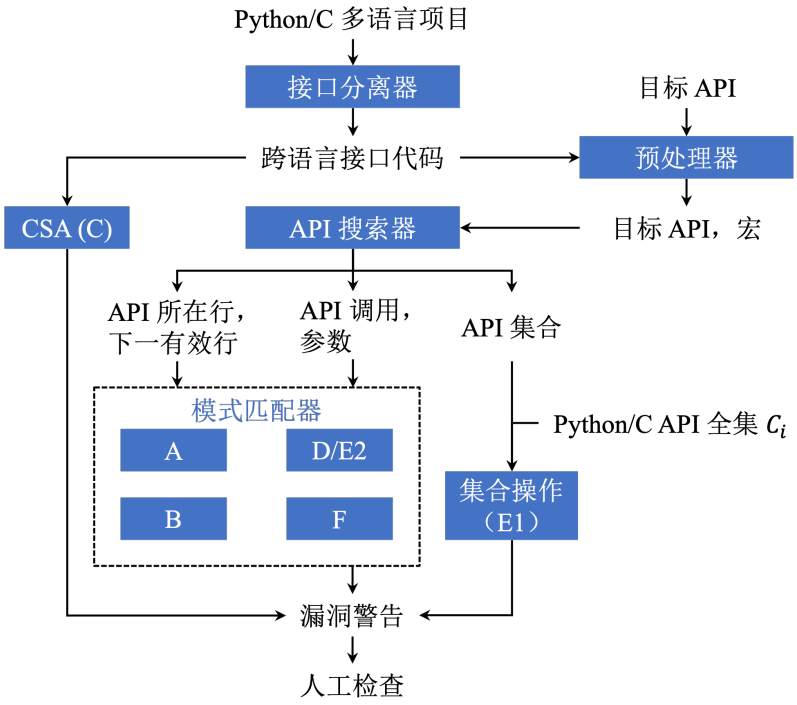


图 3.13　PyCEAC 工具集的漏洞检查部分架构图

第 3.4 节提到的漏洞检查器主要包括基于 grep 的脚本、基于集合操作的检查器、基于 CSA 的检查器。图 3.13 总结了这些检查器的设计实现，字母按照第 3.4 节的先后顺序标记了各个漏洞模式（见表 3.5 第 2 列）。由多语言项目分离出的跨语言接口代码是这些分析的输入。对于不同的漏洞模式，检查器预置了目标 Python/C API 检查集合和相关的信息。例如，异常处理错误的漏洞检查器搜索 27 个抛出异常或警告的 Python/C API；不完整的错误检查的漏洞检查器预置了需要额外检查的热点 Python/C API 及其用以标识内部错误的返回值，同

时通过预处理器寻找目标 Python/C API 的宏。漏洞检查器 A（异常处理错误）、漏洞检查器 B（不完整的错误检查）检查目标 Python/C API 所在行及其下一有效行是否与规则相匹配，然后用基于模式的假设来提高精度。下一有效行会忽略注释、空行、折行。漏洞检查器 D（整数溢出）、漏洞检查器 F（缓冲区溢出）分析目标 Python/C API 调用的实参。规则和目标 Python/C API 子集的总结来自 PyCEAC 的提取和分析模块得到的设计迭代和使用行为（见第 3.3 节）。漏洞检查器 C（内存管理错误）基于 CSA 的 unix.Malloc 和 unix.MismatchedDeallocator 分析器。漏洞检查器 E（API 迭代兼容性）包含两种不同的方法。E1 基于集合操作检查被分析项目中是否使用到了某个 Python 编译器版本之外的 Python/C API。不同版本的 Python/C API 全集使用 PyCEAC 的提取工具分析得到。E2 使用和 D、F 类似的参数解析分析。所有漏洞检查器的输出警告通过人工检查验证并剔除误报。考虑到本章覆盖更多漏洞模式的研究目标，及其带来的人工警告校验的工作量，虽然漏洞模式的总结和漏洞检查器的实现考虑了不同 Python/C 多语言软件系统的行为，但是实证安全分析仅在 Pillow 上检查了这 6 类漏洞。

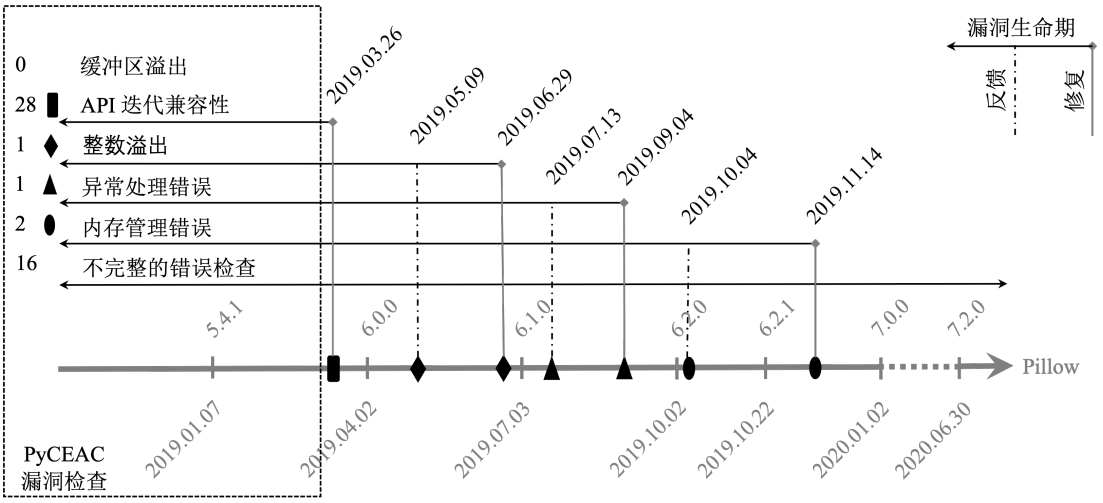


图 3.14　Pillow 中漏洞发现的时间线

图 3.14 所示是 Pillow 中漏洞发现的时间线，水平实线标记了发现版本和对应的漏洞存在的时间区间，竖直点划线是对应漏洞通过问题列表被反馈的时间，竖直实线则是漏洞修复被合并的时间。在大约一年的时间里，PyCEAC 的漏洞检查器发现的 48 个漏洞中的 32 个被逐渐修复，16 个仍然存在。16 个没有被修复的漏洞来自不完整的错误检查，该漏洞模式依赖特定 Python/C API 的内部实现。当表 3.6 中的 Python/C API 的返回表示错误发生却没有被检查时，多语言软件会崩溃或产生非预期的结果。然而，当错误返回仅作为某个 Python/C API 的参数且该 Python/C API 在内部检查参数的值并抛出异常时，不需要再对返回进行额外的检查。错误情形不容易触发、不同 Python/C API 在错误检查上的不一致导致难以说服开发者为何部分 Python/C API 需要额外的检查，但是该漏洞模式已在其

他单语言的研究中被确认[147]。

3.5.2　漏洞模式的有效性

PyCEAC 使用轻量级的漏洞检查器以覆盖更多的漏洞模式，其结果说明这些漏洞模式几乎全部都在主流多语言软件的开发中实际存在。仅使用 Pillow 进行漏洞实证分析并不会影响漏洞模式的一般性，因为漏洞总结的 4 类来源都不局限于某一特定的项目。由于分析和总结漏洞模式包含人工检查以及和开发者的交流，如图 3.14 所示，Pillow 上所有漏洞的问题讨论、修复、合并花费了一年的时间，因此难以在更多的项目上进行如此全面的实验。

将异常处理错误的漏洞检查扩展到 NumPy 验证了漏洞模式的有效性。NumPy 是使用 Python 进行科学计算的基础包，它支持大型的多维数组和矩阵，并提供大量的高层数学函数以操作这些数据结构。NumPy 1.19.0 包含超过 35 万行代码，其中外部 C/C++ 代码超过 16 万行，其在规模上比 Pillow 5.4.1 高一个数量级。在 NumPy 上（numpy 子文件夹，超过 27 万行代码）的异常处理错误漏洞检查在 7s 内扫描了 680 个文件并分析了其中的 70 个跨语言接口文件，给出了 63 个警告。通过人工检查，本文在这些警告中确认了 1 个异常处理错误漏洞，该漏洞已被官方确认并在版本 1.19.1 中被修复[181]。

3.5.3　误报率

本章以全面总结漏洞模式为研究目标，轻量级的检查器结合人工检查通过实际项目中的漏洞实例辅助漏洞模式的说明，并没有针对单一漏洞进行精细的设计，误报率能够反映这一点。

以异常处理错误为例，其误报存在三个主要原因：（1）抛出异常的 Python/C API 和对应的后处理之间存在条件编译宏；（2）使用 `if`、`else`、`switch` 等条件表达式发送不同的错误信息改变了异常抛出和后处理之间的控制流；（3）漏洞检查器未知的后处理，例如 NumPy 中的自定义函数 `npy_PyErr_ChainExceptionCause`。

降低误报率对于学界和业界的静态程序分析研究和工具而言都是一种挑战[182-183]。PyCEAC 的漏洞检查器增加了许多启发式的判断，例如前面描述的多种使用模式假设，并取得了不错的效果。但是整体的人工检查工作量仍然较大。一方面，目标多语言软件 Pillow 作为 Python 中大量使用的图像处理库，经过了大量测试和使用，以及长期版本开发迭代，相对容易触发的漏洞大多已经被暴露并修复了。另一方面，本章出于全面分析总结漏洞模式的需求，静态检查工具更关心以相对简单的方法覆盖多种漏洞模式。

对于 PyCEAC 的漏洞检查器，其误报率（False Positive Rate，FPR）如表 3.8

表 3.8　PyCEAC 漏洞检查的误报率

漏洞模式	静态检查器	漏洞数	警告	FPR
异常处理错误	基于 grep 的脚本（A）	1	10	90%
不完整的错误检查	基于 libclang 的扫描器 + 基于 grep 的脚本（B）	16	59	72.9%
内存管理错误	基于 CSA 的检查器（C）	2	2	0%
整型溢出	基于 grep 的脚本（D）	1	8	87.5%
API 迭代兼容性	基于 grep 的脚本（E1）	10 (28)	95	89.5%
缓冲区溢出	基于 grep 的脚本（F）	0	9	100%

所示。使用基于 grep 的脚本检查 API 迭代兼容性漏洞时，由于一个参数解析表达式能够处理多个变量的类型转换，即一个格式化串可能包含多个格式化字符，因此在 10 个确认的错误使用位置共发现了 28 个过期的类型使用。

3.5.4　静态分析框架

PyCEAC 的静态检查是粗粒度的，基于简单的语法模式匹配。构建高自动化、高精度的静态分析框架需要更高层级的抽象和更精细的架构和算法设计。本章的漏洞模式分类对其是重要的基础。除了单一漏洞模式的检查，不同漏洞模式的分析之间还可以相互辅助。例如对于异常处理错误，如果跨语言接口层的一个自定义函数中存在抛出异常的 Python/C API，那么该函数内部和其调用点都应该显式返回以确保不会进入非预期的控制流。PyCEAC 的漏洞检查器只支持函数内的检查，而函数间的异常处理错误本质上是自定义函数的不完整的错误检查漏洞。PyCEAC 只检查目标 Python/C API 及其对应的用户定义宏，不检查自定义函数。支持更高层的抽象和更精确的分析需要过程间的程序分析框架。同时对于涉及复杂语言特性的漏洞模式，抽象解释[133,184-185]、可满足性模理论[156,186]（Satisfiability Modulo Theories，SMT）、类型推断[187-188]等静态程序分析技术可能有所帮助。

3.5.5　有效性威胁

本小节讨论本章的实证安全分析的内部和外部有效性。

1. 内部有效性

内部有效性威胁主要关乎实验偏差和错误。

在回答 Python/C API 在多语言软件中的使用迭代时，版本分析的时间间隔大约是 15 个月，该时间选择和本文作者希望使用当时最新的版本进行分析，以及漏洞检查实验的周期有关。但是作者认为该时间间隔的选择不会带来结论性的偏差，因为图 3.4 中的多语言项目都是密集更新的主流项目，其发行版本的时

间间隔大多不超过一个月。尽管如此，先计算 Python/C 多语言项目开发的一般周期可能是更好的选择[189]。一个阻碍是 GitHub API 不直接提供多语言的项目搜索，而嵌套查询很快会超过查询频率限制，因此本章在多语言项目选择时直接使用了 Pillow、NumPy、TensorFlow、PyTorch 等流行的项目。此外，本章没有选择一些外部代码占比太少的多语言项目，筛选阈值 20% 或许存在更优化的选择策略。较高的误报率是另一个威胁。漏洞检查器不是本章的重心，其主要被用来说明现实工程中漏洞的存在和模式。语法模式匹配对于精细设计的漏洞检查工具而言不是一个好的技术选择，相关细节和启示已在前文中讨论。

2. 外部有效性

外部有效性威胁主要关乎实证发现的一般性。

本章主要关注 Python/C API 的设计迭代、使用行为、漏洞模式。尽管后续设计的漏洞检查器的精度表现较差，但是其不影响对设计迭代、使用行为、漏洞模式的分析。其中漏洞模式的分析总结考察了 4 个来源：（1）Python/C API 参考手册，（2）多语言项目的问题列表，（3）已有的多语言研究，（4）语言特性差异的分析，漏洞模式的分析总结和漏洞检查器的实现是独立的。另一个威胁是漏洞模式的完整性。考虑以上 4 个来源，总结的 9 类漏洞模式比较常见，但是仍然可能遗漏相对不常见的漏洞模式。

3.6　本章小结

对于 Python 及其 C/C++ 扩展模块组成的多语言软件系统，Python/C API 编程的安全隐患就像是跨语言接口代码中的定时炸弹。本章揭示了 Python/C API 在 Python 编译器中的设计迭代和在主流 Python/C 多语言软件中的使用行为，并分析总结了 Python/C 跨语言接口代码中常见的漏洞模式。本章设计实现的静态分析工具集 PyCEAC 包含 Python/C API 的提取分析模块和多类漏洞的轻量检查器，并在主流多语言软件中发现了多种漏洞，部分已被确认并修复。考虑到 Python/C 多语言软件架构的广泛使用，其设计迭代、使用行为、漏洞模式分类为构建高自动化、高精度的多语言漏洞检查工具提供了有益的指引。工具实现以及对其局限性和未来方向的讨论对于多语言软件的跨语言接口编程、漏洞分析、语言接口设计具有实际的价值。

第 4 章　动态类型语言的外部函数的静态类型推断

跨语言程序的类型误用是第 3 章总结的 9 类多语言软件漏洞之一，并且难以通过语法模式匹配等相对简单的方法对其进行有效的检查。作为多语言软件一个复杂且难以分析的性质，跨语言的类型约束是互操作的关键因素之一。静态类型推断是维护动态类型语言程序安全性的有效手段。然而，使用另一种语言编写的外部函数往往不在推理范围内。作为一个流行的动态类型语言，Python 有许多广泛使用的包采用包含 C/C++ 扩展模块的多语言架构。已有的确定性的、不基于类型标注的 Python 静态类型推断工具无法处理这些扩展模块中的外部函数。本章提出了一种通过分析跨语言接口层中外部接口使用的隐式信息来推断外部函数类型签名的方法。基于对 Python/C 跨语言程序的静态语义的形式建模，描述抽象语法、类型和类型推断规则，本章设计了静态类型推断系统 PyCType。在 CPython、NumPy、Pillow 上的评估说明 PyCType 能够可靠地推理大部分外部函数的参数类型和数量。类型推断的结果可以进一步作为对 Python 静态类型推断工具的补充，使得其在包含外部函数调用的程序上的精度得到提高。同时，该方法还发现了 48 个外部函数声明及其实现不一致的漏洞，该漏洞会导致无参外部函数可以接收任意类型的参数，其中 NumPy 和 Pillow 上的 8 个漏洞已被开发者确认并修复。

4.1　引言

如第 3 章所分析的，语言特性差异可能导致使用 Python/C API 的 Python/C 跨语言接口代码存在许多安全隐患，类型误用就是常见的漏洞之一。同时类型信息可以服务于多种程序分析任务，使得类型推断成为了提高动态类型语言程序的安全性和可维护性的重要技术。

学界和业界已经为设计和实现 Python 的静态类型推断方法和工具做了很多努力。Python 社区的 mypy[190]、微软公司的 Pyright[191]、Meta 公司①的 Pyre[192] 使用基于类型标注的方法，需要修改源码，一定程度地破坏了 Python 灵活敏捷的开发实践。Xu 等人[159]、Allamanis 等人[160] 使用基于机器学习的方法，得到的是非确定性、概率性的类型推断结果。其他确定性的、不依赖类型标注的代表工具则都不支持外部函数的类型推断，包括 Monat 等人的工作[157]、Hassan 等人的工作[156]、Fritz 和 Hage 的工作[155]、谷歌公司的 Pytype[193]、SourceGraph 等公司使用的 PySonar2[194] 等等。由于没有自动推断外部函数的类型签名的能力，

①旧称为脸书（Facebook）公司

这些工具选择忽略外部函数或预置类型存根（type stub），前者可能带来严重的精度损失，后者需要繁重的人工或外部库开发者的配合。

宿主语言通过外部接口调用外部语言。一个全功能的外部接口一般可以分成两层：一个机器相关的下层，比如常见的基于 libffi[195] 或相似的实现；一个上层指明跨语言的值传递和类型转换等调用接口描述。对于 Java、OCaml 等静态类型的宿主语言，外部函数在声明时带有显式类型，使得外部函数的静态类型推断成为可能[40,90]。然而，动态类型语言的外部函数的静态类型推断并非无解。本章的关键想法在于静态分析调用接口描述中的隐式信息，推理参数的类型和个数等作用于外部函数的约束。

本章提出 Python 的 C 外部函数的静态类型推断系统 PyCType。PyCType 基于一系列推断外部函数类型签名的规则进行构建。这些规则可以分成三组：（1）外部函数声明，其指明 Python 侧调用的外部函数的构造方式，包括外部 C 实现的绑定和调用惯例的声明；（2）参数类型转换，其指明 Python 侧传入外部函数的参数是如何转换为外部侧 C 类型的对象的，包含基于调用惯例标记的无参分析、未使用形参分析、参数解析 Python/C API 分析；（3）返回类型转换，其指明外部函数是如何使用 C 变量构建其 Python 侧的返回值的，例如对多返回特性的支持，这部分规则包括值构建 Python/C API 分析、显式转换 Python/C API 分析、类型转换分析、可达定义分析。

本章在 CPython、NumPy、Pillow 上对 PyCType 进行了评估，它们是三个大量使用的 Python/C 多语言软件系统，且都包含大量的外部函数。对于总计 1751 个外部函数，PyCType 推断了其中 1338 个的类型签名。由于本章提出的静态类型系统是保守且可靠的，其推断结果是正确的，但可能不够精确。考虑外部函数的返回可能是其他外部函数或本地函数的参数，这使得一定程度的不完备和不精确是可以接受的。Pytype 是谷歌公司开源的先进的 Python 静态类型推断工具，使用 PyCType 对 Pillow 的类型推断结果作为对 Pytype 的补充，在不同领域的基于 Pillow 的流行项目上的实验表明，PyCType 可以将 Pytype 的召回精度提高 27.5%，类型增强实验验证了 PyCType 有效性。

据本文作者所知，本章的研究是对以下两点内容的首次探索：（1）本章提出了第一个 Python 外部函数的静态类型推断系统，该系统是确定性的，不使用机器学习方法，同时也不依赖于类型标注；（2）本章第一次研究了外部函数实现及其声明不一致的漏洞，其违反了外部函数的参数类型约束。

本章的主要贡献如下：

- 本章基于一系列可组合的规则定义了一个 Python 外部函数的静态类型推断系统。该系统包含对 Python/C 跨语言接口代码的多个静态分析。本章基于外部接口语义的方法可以推广到 Python 外的其他动态类型宿主语言。

- 本章对提出的方法进行了原型实现，类型推断系统 PyCType 在三个代表性 Python/C 多语言项目上的评估证明了其能够推断大多数外部函数，基于形式化方法的严格规范保证了推断结果的可靠性，而其完备性可以通过描述更多 Python/C API 的类型语义加以提高。通过使用推断结果增强 Pytype 说明了 PyCType 的有效性，PyCType 提升了 Pytype 在包含外部函数调用的 Python 程序上的精度。
- 作为类型推断系统的一部分，外部函数声明与实现的一致性检查规则被表示为门限语义谓词[93]及其合取范式。PyCType 在类型推断的同时发现了 48 个外部函数声明与实现不一致的漏洞，这些漏洞会导致无参外部函数可以接收任意类型的参数，其中 8 个漏洞已被确认并修复。

4.2　研究动机

本章介绍 Python/C 多语言软件架构中的跨语言接口实现，以及单语言视角下 Python 静态类型推断的不足之处。

4.2.1　Python/C 跨语言接口

NumPy、Pillow 等多语言软件系统通过多种语言编写，其中宿主语言 Python 能够提高开发效率，而外部语言 C 则支持高性能的库实现。两个互操作语言间的跨语言接口代码是基于 Python/C API 的 C 代码。第 3 章第 3.2 节从互操作性角度介绍了 Python/C 多语言软件，本章更关注跨语言接口代码及其中的类型隐式信息，而非多语言软件的整体架构。如图 4.1 所示是一个简化的 Python 的 C 扩展模块，其中忽略了跨语言接口代码对 C 库的调用。

图 4.1(b) 第 1–3 行声明外部模块 `ext` 的方法表，其中每个 `PyMethodDef` 类型的元素对应一个外部函数的描述信息，`PyMethodDef` 结构体依次包含 4 个域：`ml_name` 是外部函数在 Python 侧的调用名，`ml_meth` 是外部函数的 C 实现的函数指针，`ml_flags` 是一个位段（bit field）标记，其指明外部函数构建的方式，可选的 `ml_doc` 是外部函数的文档字符串。如图 4.1(b) 第 2 行，`"foo"` 是 Python 侧的外部函数调用名，对应图 4.1(a) 中的 `ext.foo`，`_foo` 指向图 4.1(b) 第 7–12 行定义的 C 函数，位段标记 `METH_VARARGS` 对应一种典型的调用惯例，表示外部函数的 C 实现为 `PyCFunction` 类型，其接收两个 `PyObject *` 类型的值作为参数。第一个 `PyObject *` 类型的参数一般被称作 `self`。当方法表通过 `PyModuleDef` 初始化时，如该例图 4.1(b) 第 4–6 行，`self` 代表外部模块对象；而当方法表是通过 `PyTypeObject` 声明的时候，`self` 则对应于方法所属的类实例对象。第二个 `PyObject *` 参数（一般称作 `args`）是一个元

```
import ext
size = ext.foo(width, height)
```

(a) host.py

```
static PyMethodDef extMethods[] = {
  {"foo", (PyCFunction)_foo, METH_VARARGS, "docstring"}
};
static struct PyModuleDef extModule = {
  PyModuleDef_HEAD_INIT, "ext", "docstring", -1,
   extMethods
};
static PyObject* _foo(PyObject* self, PyObject* args) {
  int x, y;
  if (!PyArg_ParseTuple(args, "II", &x, &y))
    return NULL;
  return PyLong_FromLong(x * y));
}
```

(b) interface.c

图 4.1　Python/C 跨语言接口代码示例

组对象，其中打包了 Python 侧传入外部函数的所有实参。

4.2.2　单语言视角下的静态类型推断

静态类型推断是维护动态类型语言安全性的有效手段，已有许多工作关注 Python 的静态类型推断，这些工作可以分成三类。

（1）基于类型标注。通过使用类型提示[158,196]重写 Python 程序中类型敏感的部分，进而推理并检查 Python 程序中更多对象的类型。mypy[190]、Pyright[191]、Pyre[192] 等工具能够基于类型标注检查类型误用。基于类型标注的方法可以对任意 Python 对象或函数进行标注，而不区分被标注函数是定义在 Python 中的本地函数，还是定义在外部 C 代码中的 Python 外部函数。但是由于标注源码需要有经验的开发者许多额外的工作量，同时破坏了 Python 敏捷开发的实践，导致该方法可扩展性较差，难以大规模地应用。一个使用基于标注的类型推断方法的合适场景是在向其他开发者暴露某个 Python 库的接口时给出接口的类型标注。

（2）基于机器学习。机器学习的方法[159-160]通过给 Python 变量打上类型标签，学习变量的命名规则和类型特征，然后基于学得的知识进行类型推断。该推断方法基于变量名称文本，如函数形参的名称，因此也可以支持外部函数。基于机器学习的类型推断简单实用，在测试中表现出不错的精度和时间效率。然而，机器学习方法的精度往往是一个 top-N 的结果，每个推断类型是在一定概率下正确，难以在理论上提供严格的可靠性保证，导致类型推断结果需要人工校验，这对于一些确定性的、安全敏感的程序分析任务是无法接受的。

（3）基于经典静态程序分析技术。由于一些固有的不足与类型推断的安全保

证间的矛盾，基于类型标注和机器学习的类型推断方法在工业开发实践中使用较少。更多研究仍然聚焦于设计实现基于经典静态程序分析技术的类型推断方法，包括基于数据流分析[155]、抽象解释[157,193-194]、可满足性模理论[156] 等技术。这些方法能够产生确定性的结果且不需要修改源码。此外，基于经典静态程序分析技术的方法可以分别为基于类型标注和机器学习的方法提供自动类型标注和训练标签。然而，这些方法需要从程序的语法、语义出发，因此都没有考虑外部函数的推断，使得对于 Python 生态中大量使用的多语言软件，其类型召回率始终难以继续提高。

已有的确定性的、不基于类型标注的 Python 静态类型推断工具都是从单语言视角出发分析程序的[155-157,193]。对于如图 4.1(a) 中的外部函数调用 `ext.foo`，其参数类型和返回类型都无法推断。因此由宿主侧传递到外部侧的参数（`width` 和 `height`）的类型无法在编译时进行检查，导致难以保持程序的安全性和可维护性。

已有的基于经典静态程序分析技术的 Python 类型推断的不足来自一个固有的认识，即动态类型语言无法在编译时得到类型信息。将图 4.1 中 Python 的外部函数声明和其他静态类型宿主语言相比，这一假设似乎是无法改变的。如下，在 Java 中，关键字 `native` 声明了一个名为 `bcopy` 的外部函数，它接收一个 `byte[]` 类型的参数，并返回 `void`。

```
private native void bcopy(byte[] arr);
```

相似地，在 OCaml 中，一个外部函数接收两个参数：通过 `foreign` 声明的要绑定的外部 C 实现的函数名，以及一个值用来描述被绑定函数的类型。`@->` 操作符将一个参数类型加入参数列表，`returning` 终结参数列表，并且后跟返回类型。

```
let newwin =
  foreign "newwin"
    (int @-> int @-> int @-> int @-> returning window)
```

由于 Python 标准库和内置函数的很大一部分也是 C 外部函数，谷歌公司的 Python 静态类型推断工具 Pytype 通过内置 Python 社区提供的 Typeshed[197] 作为外部类型标注。Typeshed 以及 mypy[190] 中相似功能的模块 stubgen 通过人工编码、文档提取、运行时内省（introspection）等方法提供类型存根。这些方法都无法大规模地应用，导致 Typeshed 仅支持一些小型的第三方扩展模块。同时在现实开发场景中，C 扩展模块可能是仅供内部使用的优化后的私有库。

4.3　关键想法与基础定义

本节描述建模 Python 外部函数的静态类型推断这一问题的关键想法，并给出 Python/C 跨语言程序的抽象语法、类型等基础的形式定义。

4.3.1　关键想法概述

没有显式的类型声明，Python/C 跨语言接口代码需要给出调用接口描述来桥接语言间不同的语言特性，例如内存管理[33-34]和异常处理[35-36]。本章关注类型系统，建模并分析跨语言接口中类型转换的隐式信息。

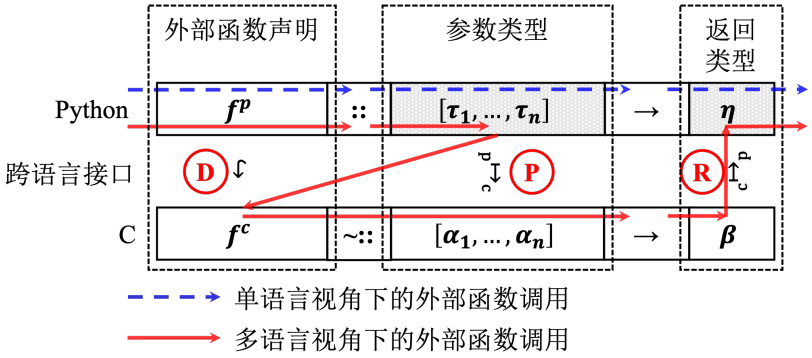


图 4.2　多语言和类型系统视角下的外部函数调用

图 4.2 是单语言和多语言类型系统视角下的外部函数调用的对比。在蓝色虚线所示的单语言视角下，外部函数的参数类型和返回类型无法被静态推断（灰色框）。因为不同于函数实现和函数调用都在 Python 侧的本地函数，Python 外部函数在 Python 侧调用，但是其基于 Python/C API 的声明和实现都在 C 侧。在红色实线所示的多语言视角下，不同于静态类型宿主语言的显式类型声明，动态类型宿主语言在跨语言接口代码中通过 Python/C API 和相关程序性质给出的隐式信息来指明：外部函数的声明方式（Ⓓ）、宿主侧到外部侧的参数类型转换（Ⓟ）、外部侧到宿主侧的返回类型转换（Ⓡ）。

为了建模并分析这些隐式信息，类型推断规则包含图 4.2 所示的三个部分作为前提：外部函数声明 Ⓓ、参数类型转换 Ⓟ、返回类型转换 Ⓡ。以图 4.1 中的跨语言接口代码为例对这三类推理前提加以解释：

- 外部函数声明前提 Ⓓ 通过 `PyMethodDef` 将外部函数在宿主侧的调用名 f^{p} 映射到其外部 C 实现 f^{c}，即将图 4.1 中的 `"foo"` 映射到 `_foo`。f^{c} 是 `PyMethodDef` 结构体的 `ml_meth` 域指向的 C 函数，当通过 `ml_flags` 域指明对应的外部函数接收位置参数（positional arguments）时，f^{c} 有 `PyCFunction` 类型并接收两个 `PyObject *` 类型的参数 `self` 和 `args`。因此事实上，f^{c} 的类型约束并不是其 C 类型签名，而应该是 Python 外部函数的类型签名对应的转换类型。

- 参数类型转换前提 Ⓟ 根据如图 4.1(b) 第 9 行中 Python/C API 函数 `PyArg_ParseTuple` 的格式化串 `"II"` 指明的类型约束，将 Python 侧的外部函数的实参解析并转换为 C 变量。格式化串 `"II"` 指明对应的外部函数接收两个 Python 整数作为参数，它们在跨语言接口层被转换为 `unsigned int` 类型的 C 变量。
- 返回类型转换前提 Ⓡ 根据外部函数 C 实现的返回将 C 变量转换为 Python 变量，并作为外部函数的返回值。在图 4.1 中，Python/C API 函数 `PyLong_FromLong` 施加一个从 C 整型到 Python 整型的类型转换。

根据以上三类前提组合给出的约束，如果一个非整型的 Python 对象被传入外部函数 `ext.foo`，由于 Python 使用动态类型检查，因此会在运行时得到一个类型错误；如果一个具有负值或零值的 Python 整型变量被传入，则会导致对应的 C 变量的溢出，外部函数返回后得到一个非预期的错误值。图 4.1 的例子也说明，Python 外部函数的静态类型推断对于保持多语言软件系统的安全性和可维护性是很重要的。

关键想法对应的完整严格的形式化规范，以及对相关的 Python/C API 和程序性质的静态程序分析将在第 4.4 节中详细描述。

4.3.2　抽象语法

图 4.3 所示是 Python/C 跨语言程序的抽象语法。上标 p 和 c 标记项所属的语言侧，p 代表宿主 Python 侧，c 代表外部 C 侧。该简化的语法只包含必要的语言构造，聚焦于外部函数以及类型推断，而忽略其他不相关的语言特性。本章的方法实现 PyCType 支持 CPython 3.3+ 和 C99 语言。

在每个语言侧，程序 p 是一个表达式列表。表达式遵循其所属语言侧的语法。在 Python 中，import m 导入模块 m，其可能是纯 Python 模块或包含 C/C++ 外部程序的扩展模块。Python 本地函数通过 def 表达式定义，其定义和调用都在 Python 侧。Python 外部函数在宿主 Python 侧通过 $e.f$ 调用，其中 e 是扩展模块或外部变量。当 e 是扩展模块 m 时，$m.f$ 调用外部模块函数；当 e 是外部变量 x 时，$x.f$ 调用外部类方法。Python 外部函数仅有其调用在宿主 Python 侧，其声明和定义都在外部 C 侧。如前所述，外部函数的声明基于 Python/C API 结构体 `PyMethodDef`，定义则是一个 `PyCFunction` 类型的函数，函数体内通过参数解析、值构建、显式转换等 Python/C API 进行跨语言的类型转换。

4.3.3　类型

在 Python/C 跨语言程序中，Python 侧和 C 侧的类型集合分别表示为 $\mathbb{T}_{\mathrm{p}}$ 和 $\mathbb{T}_{\mathrm{c}}$。图 4.4 定义了 Python 侧的类型 $\tau, \eta \in \mathbb{T}_{\mathrm{p}}$，图 4.6 定义了 C 侧的类型 $\alpha, \beta \in$

p^{p} ::= $[s_1^{\mathrm{p}}, \cdots, s_n^{\mathrm{p}}]$ — Python 程序

s^{p} ::= import m — 模块导入
| def $f(x_1^{\mathrm{p}}, \cdots, x_n^{\mathrm{p}})$: p^{p} — 本地函数定义
| $x^{\mathrm{p}} = e^{\mathrm{p}}$ — 赋值

e^{p} ::= x^{p} — 变量
| m — 模块
| $(e_1^{\mathrm{p}}, \cdots, e_n^{\mathrm{p}})$ — 元组
| $[e_1^{\mathrm{p}}, \cdots, e_n^{\mathrm{p}}]$ — 列表
| $f(e_1^{\mathrm{p}}, \cdots, e_n^{\mathrm{p}})$ — 本地函数调用
| $m.f^{\mathrm{p}}(e_1^{\mathrm{p}}, \cdots, e_n^{\mathrm{p}})$ — **外部函数调用**
| $x^{\mathrm{p}}.f^{\mathrm{p}}(e_1^{\mathrm{p}}, \cdots, e_n^{\mathrm{p}})$ — 外部类方法调用

p^{c} ::= $[s_1^{\mathrm{c}}, ..., s_n^{\mathrm{c}}]$ — C 程序

s^{c} ::= *cType* x^{c} — 变量类型声明
| $x^{\mathrm{c}} = e^{\mathrm{c}}$ — 赋值
| PyMethodDef $\{f^{\mathrm{p}}, f^{\mathrm{c}}, \mathit{flag}, \cdots\}$ — **外部函数声明**
| PyObject $*f^{\mathrm{c}}$(PyObject $*\mathit{self}$, PyObject $*\mathit{args}$) $\{p^{\mathrm{c}}\}$ — **外部函数定义**
| return e^{c} — 函数返回

e^{c} ::= x^{c} — 变量
| $\iota_{\mathrm{ap}}(\mathit{args}, u_1 \cdots u_n, x_1^{\mathrm{c}}, \cdots, x_n^{\mathrm{c}})$ — 参数解析
| $\iota_{\mathrm{vb}}(u_1 \cdots u_m, x_1^{\mathrm{c}}, \cdots, x_m^{\mathrm{c}})$ — 值构建
| $\iota_{\mathrm{ec}}(x^{\mathrm{c}})$ — 显式转换

其中 $\mathit{flag} \in \mathbb{F}$，$\iota_{\mathrm{ap}} \in \mathbb{I}_{\mathrm{ap}}$，$\iota_{\mathrm{vb}} \in \mathbb{I}_{\mathrm{vb}}$，$\iota_{\mathrm{ec}} \in \mathbb{I}_{\mathrm{ec}}$。$\mathbb{F}$ 是位段标记集合，$\mathbb{I}_{\mathrm{ap}}$、$\mathbb{I}_{\mathrm{vb}}$、$\mathbb{I}_{\mathrm{ec}}$ 分别表示进行参数解析、值构建、显式转换的三族 Python/C API。

图 4.3　Python/C 跨语言程序的抽象语法

$\mathbb{T}_{\mathrm{c}}$。

Python 是动态类型语言，Python 侧的类型是程序运行时绑定到 Python 变量的类型值，如 `str`（Python 3 中的 `unicode`）、`int`、`object` 等。除了这些内置类型，积类型（*pProduct*）被用来表示 `list`、`tuple`、`dict` 等类型，函数类型（*pFunc*）被用来表示（本地和外部）函数的类型。*pUnion* 通过一个类型集合初始化，其不属于 Python 运行时，但其对应的和类型是 C 中常见的语言特性，并且可能参与外部函数的返回类型转换。这里省略了 `module`、`iterator` 等 Python 类型，它们不能使用 C 的语言特性表示，该类型的值作为外部函数参数时会被视作 `object` 类型。

τ, η ::= `pUnicode|pBytearray|pBytes|pArray|pMemoryview`
| `pInt|pFloat|pComplex|pInt_nonnegative`
| `pObject|pBool|pNone`
| $\mathit{pProduct}(\tau_1, \cdots, \tau_n) \mid \mathit{pUnion}(\tau_1, \cdots, \tau_n) \mid \mathit{pFunc}([\tau_1, \cdots, \tau_n], \tau_r)$

图 4.4　Python 侧的类型（*pType*: $\tau, \eta \in \mathbb{T}_{\mathrm{p}}$）

调用接口描述不仅可以指明跨语言的类型转换，还可以指明比类型更严格的值约束。例如，格式化字符 `b`、`B`、`H`、`I`、`k`、`K` 要求传入外部函数调用的实参为非负整数。为此引入了 `pInt` 类型的子类型 `pInt_nonnegative`。给定子

定型关系 <:，τ' 是 τ 的子类型 $\tau' <: \tau$，如果在任何期待 τ 类型的项的上下文中都可以使用 τ' 类型的项替代。图 4.5 所示是 Python 侧类型的子定型规则。

$$\frac{}{\tau <: \tau}$$

$$\frac{}{\tau <: \texttt{pObject}}$$

$$\frac{}{\texttt{pInt\_nonnegative} <: \texttt{pInt}}$$

$$\frac{}{\texttt{pInt} <: \texttt{pFloat}}$$

$$\frac{\tau_i <: \eta_i \quad 1 \leqslant i \leqslant n}{pProduct(\tau_1, \cdots, \tau_n) <: pProduct(\eta_1, \cdots, \eta_n)}$$

$$\frac{set(\tau_1, \cdots, \tau_n) \subseteq set(\eta_1, \cdots, \eta_n)}{pUnion(\tau_1, \cdots, \tau_n) <: pUnion(\eta_1, \cdots, \eta_n)}$$

$$\frac{\tau_i <: \eta_i \quad 1 \leqslant i \leqslant n \quad \eta_r <: \tau_r}{pFunc([\eta_1, \cdots, \eta_n], \eta_r) <: pFunc([\tau_1, \cdots, \tau_n], \tau_r)}$$

图 4.5　Python 侧类型的子定型规则

函数对于参数类型是逆变的（contravariant），对于返回类型是协变的（covariant）。`pBool` 对应任意一个能够测试真假的 Python 值。在 Python 的鸭子类型系统中（参考前文第 3.2.2 小节第 2 小小节），布尔（bool）对象可以是任意类型的值，只要该对象拥有 `__bool__` 或 `__len__` 属性。为了避免原始类型对跨语言类型转换规则的影响，`pBool` 被设置为 `pObject` 类型的子类型。而在 Python 单语言类型系统中，`pBool` 是 `pInt` 的子类型。`pInt` 和 `pFloat` 之间的子类型关系也不在 Python 运行时中，但是考虑与 C 类型系统的兼容，在外部函数调用中整型可能被隐式地转换为浮点型。

α, β ::= α * | `const` α | `unsigned` α | `void`
| `char` | `short int` | `int` | `long` | `long long` | `float` | `double`
| `Py_UNICODE` | `PyByteArrayObject` | `PyBytesObject` | `Py_buffer`
| `Py_ssize_t` | `Py_complex` | `PyObject`
| $cProduct(\alpha_1, \cdots, \alpha_n)$ | $cUnion(\alpha_1, \cdots, \alpha_n)$ | $cFunc([\alpha_1, \cdots, \alpha_n], \alpha_r)$

图 4.6　C 侧的类型（*cType*: $\alpha, \beta \in \mathbb{T}_c$）

C 侧的类型包括：（1）常见的 C 类型，例如 `int`、`long` 等不同位长的整型、指针类型等，（2）实现 Python 内置类型的底层结构体，如 `Py_UNICODE` 等，（3）为支持 Python 侧语言特性而引入的类型，如 *cProduct* 等。Python 侧的类型都使用“p”作为前缀，在不引起混淆的情况下，这里一般的 C 类型在表示时不再附加前缀。

4.4　类型系统

多语言类型系统包含宿主语言和外部语言的类型集合，以及用以确定给定上下文中不同语言侧的程序构造的类型的一系列规则。前文已经定义了 Python/C 跨语言程序中的程序构造和类型，本节关注类型推断规则。

4.4.1　假设判断定义

首先形式定义类型系统中使用到的假设判断。

$$\Gamma \vdash e :: \tau \qquad \text{(类型赋值)}$$

陈述表达式 e 在定型上下文 Γ 中有类型 τ，其中定型上下文 Γ 包含若干已知的类型判断 $x :: \tau$。根据变量所属的语言侧可以把 Γ 分为宿主侧上下文和外部侧上下文，与前文同样使用上标 p 和 c 标记。由于在 Python 语法中含有单冒号，因此本文中的类型赋值使用双冒号作为区分。

在宿主语言程序中，本地函数是定义和调用都在宿主侧的函数，而外部函数其调用在宿主侧，声明和定义在外部侧。形如

$$\Gamma^{\mathrm{c}} \vdash f^{\mathrm{p}} \xrightarrow{flag} f^{\mathrm{c}} \qquad \text{（外部函数声明）} \qquad \text{(D)}$$

的外部函数声明把宿主 Python 侧的外部函数调用名 f^{p} 映射到实现在 C 侧的外部函数定义 f^{c}。Python 是动态类型的语言，在声明外部函数时没有显式类型标注，只使用调用惯例标记位 *flag* 指明外部函数构建的方式，包括是否接收参数，以及位置参数或关键字参数的个数。

$$\Gamma^{\mathrm{c}} \vdash pType \; {}_{\mathrm{p}}\overset{\mathbb{P}}{\mapsto}_{\mathrm{c}} \; cType \qquad \text{（参数类型转换）} \qquad \text{(P)}$$

$$\Gamma^{\mathrm{c}} \vdash cType \; {}_{\mathrm{c}}\overset{\mathbb{P}}{\mapsto}_{\mathrm{p}} \; pType \qquad \text{（返回类型转换）} \qquad \text{(R)}$$

通过引入表示不同形式的类型转换约定的属性 $\mathbb{P}$，进而推理作用于不同语言侧的变量的类型约束，以及它们之间跨语言的转换规则。本章推理的程序属性包括参数解析族 Python/C API 的格式化单元、特定 Python/C API 的语义，以及跨语言接口代码的程序分析得到一些程序性质。这些推理和分析作用于跨语言接口代码的不同中间表示如抽象语法树和控制流图，由于跨语言接口代码本质是 C 代码，因此此处的定型上下文为 Γ^{c}。

一些类型转换的判断仅在关于程序属性 $\mathbb{P}$ 的某个谓词为真时成立。这样的关系可以表示为如下的判断：

$$\{\pi(\mathbb{P})\}?\{J\}$$

其中 $\pi \in \Pi$ 是门限语义谓词[93]（gated semantic predicate）。在本章中，J 是参数类型转换 (P) 形式的假设判断。

4.4.2　类型推断

外部函数声明 (D)、参数类型转换 (P)、返回类型转换 (R) 共同构成外部函数类型推断规则的前提，决定外部函数的类型签名。

$$\frac{D \quad P \quad R}{\Gamma^{\mathrm{p}} \vdash f^{\mathrm{p}} :: pFunc(\circ, \diamond)} \quad \text{（类型推断）} \qquad \text{(TInfer)}$$

其中外部函数的参数类型 $\circ$ 和返回类型 $\diamond$ 由三部分前提的具体组合确定。

以下小节将分别给出三类前提的不同判断形式。

4.4.3　外部函数声明

这部分的推理前提建立外部函数调用名及其 C 实现之间的映射关系，表示为判断 (D) 的形式。由 *flag* 给出的调用惯例标记包括位段值 `METH_VARARGS`、`METH_NOARGS`、`METH_O` 等。常见的调用惯例标记 `METH_VARARGS` 指明对应的外部函数在宿主 Python 侧接收一个或多个实参作为输入，并将这些参数打包到 C 侧外部函数定义的 `PyObject * ` 类型的第二个形参（一般叫做 `args`）中。`METH_NOARGS` 应用于无参外部函数，向这些外部函数传递参数会导致运行时的类型错误。根据 Python/C API 参考手册[172]和 Python 编译器实现，其他调用惯例标记也嵌入在类型推断系统中。

4.4.4　参数类型转换

参数类型转换刻画传入宿主侧外部函数的 Python 类型的实参被外部函数 C 实现解析并保存到 C 类型变量的过程，其包含两种形式。

1. 调用惯例分析

如前所述，调用惯例标记会影响外部函数的参数类型。`METH_NOARGS` 指明外部函数是无参的，这一约束可以表示为：

$$\{\mathcal{P}_{\mathrm{PFA}}(flag)\}?\{\Gamma^{\mathrm{c}} \vdash \texttt{pNone}_{\,\mathrm{p}} \mapsto_{\mathrm{c}} \texttt{void}\} \qquad \text{(PFA)}$$

无参分析（Parameter Free Analysis，PFA）被表示为门限语义谓词 $\mathcal{P}_{\mathrm{PFA}}$，它分析 `PyMethodDef` 结构体的调用惯例标记位。只有当该谓词为真时，由 `pNone` 到 `void` 的参数类型转换判断才成立。

然而，兼容互操作双方的语言特性导致跨语言接口代码中的隐式信息可能是冗余的，使得调用惯例标记并不是外部函数无参的唯一决定因素。外部函数定义 f^{c} 的第二个形参 `PyObject * ` 类型的 `args` 被用来打包宿主侧所有传递给外部函数调用的实参。如果 `args` 在外部函数实现 f^{c} 的函数体中没有被使用，那么该外部函数实际上也是无参的。该约束对应于表示为门限语义谓词 $\mathcal{P}_{\mathrm{UPA}}$ 的

未使用形参分析（Unused Parameter Analysis，UPA）：

$$\{\mathcal{P}_{\text{UPA}}(f^{\text{c}})\}?\{\Gamma^{\text{c}} \vdash \texttt{pNone}\ {}_{\text{p}}{\mapsto}_{\text{c}}\ \texttt{void}\} \tag{UPA}$$

外部函数实现指明的约束 (UPA) 应当和调用惯例标记施加的约束 (PFA) 相匹配。然而，当调用惯例标记不是 `METH_NOARGS` 但外部函数定义不使用其参数时，该设计目的为不接收参数的外部函数将可以接受任意类型的参数，并且该扩展模块在编译时不会得到任何警告。在不增加编译属性的情况下，该漏洞可能被利用形成对多语言软件的安全攻击。从类型系统的视角来看，这种不匹配混淆了参数类型 `pNone` 和 $\mathit{pUnion}(\texttt{pNone}, \cdots)$。后者实际可以是任意 *pType* 类型，等价于 Python 官方类型提示语法[158]中的 `Optional[Any]`。

当谓词 $\mathcal{P}_{\text{PFA}}$ 和 $\mathcal{P}_{\text{UPA}}$ 同时为真时，对应的外部函数才是实际上的无参函数。利用门限语义谓词的合取范式可以表达这一约束：

$$\{\mathcal{P}_{\text{PFA}}(flag) \wedge \mathcal{P}_{\text{UPA}}(f^{\text{c}})\}?\{\Gamma^{\text{c}} \vdash \texttt{pNone}\ {}_{\text{p}}{\mapsto}_{\text{c}}\ \texttt{void}\} \quad \text{（调用惯例）} \tag{Pcc}$$

2. 参数解析分析

当谓词 $\mathcal{P}_{\text{PFA}}$ 和 $\mathcal{P}_{\text{UPA}}$ 同时为假时，对应的外部函数接收至少一个参数。本节将进一步描述参数类型转换基于参数解析族 Python/C API $\mathbb{I}_{\text{ap}}$ 的判断形式。

进行参数解析的 Python/C API 函数使用格式化串来指明外部函数期待的参数情况，包括个数、类型，以及值约束。一些值约束并没有在运行时绑定到 Python 内置类型，而是作为非形式化的规范写在参考手册中，开发者容易忽略这些跨语言编程特有的约束，进而导致多语言软件出现运行时崩溃或非预期的行为。本章通过精化的子类型描述了一些常见的值约束，如非负整数。

一个格式化串由零或多个格式化单元组成。一个格式化单元描述作为外部函数实参的一个 Python 对象的类型，以及对该对象的跨语言类型转换得到的 C 对象的类型。一个格式化单元包括一个或多个字符，或是括号包裹的一个格式化单元序列。例如，图 4.1(b) 中的 `PyArg_ParseTuple` 是 $\mathbb{I}_{\text{ap}}$ 中典型的 Python/C API 函数，其格式化串包含两个格式化单元 `I`，指明对应的外部函数 `foo` 接收两个 Python 整型对象作为参数，这两个参数将在外部函数实现中被转换为 `unsigned int` 类型的 C 变量。

对于一个参数解析 Python/C API 函数 $\iota_{\text{ap}} \in \mathbb{I}_{\text{ap}}$，$\iota_{\text{ap}}$ 的格式化串表示为格式化单元的序列 "$u_1, \cdots, u_n$"，其中每个格式化单元 u_i 指明外部函数对应位置 i 上的参数的跨语言类型转换规则：

$$\Gamma^{\text{c}} \vdash \tau_i\ {}_{\text{p}}{\overset{\iota_{\text{ap}}.u_i}{\longmapsto}}_{\text{c}}\ \alpha_i \quad 1 \leqslant i \leqslant n \quad \text{（参数解析）} \tag{Pap}$$

表 4.1 和续表 4.2 所示是格式化单元施加的参数类型转换规则的完整列表。为了节约空间，其中和类型 $\mathit{union}(\tau_1, \tau_2)$ 记作 $\tau_1 \mid \tau_2$，积类型 $\mathit{product}(\tau_1, \tau_2)$ 记作 (τ_1, τ_2)，函数类型 $\mathit{func}(\tau_{in}, \tau_{out})$ 记作 $\tau_{in} \to \tau_{out}$。Python/C 跨语言接口代码是基于

Python/C API 的 C 代码。在 Python 编译器中，Python/C API 声明在 Python.h 及其包含或间接包含的头文件中。对 Python/C API 的分析的定型上下文始终是 Γ_{c}，因此在表中也忽略了相同的定型上下文。

表 4.1　格式化单元施加的参数类型转换规则

字符串和缓冲区
$\mathrm{pUnicode}\ \overset{\mathtt{s}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{const\ char\ *}$
$\mathrm{pUnicode} \mid \mathrm{pByteslike}\ \overset{\mathtt{s*}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{Py\_buffer}$
$\mathrm{pUnicode} \mid \mathrm{pImmbyteslike}\ \overset{\mathtt{s\#}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ (\mathrm{const\ char\ *}, \mathrm{int} \mid \mathrm{Py\_ssize\_t})$
$\mathrm{pUnicode} \mid \mathrm{pNone}\ \overset{\mathtt{z}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{const\ char\ *}$
$\mathrm{pUnicode} \mid \mathrm{pByteslike} \mid \mathrm{pNone}\ \overset{\mathtt{z*}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{Py\_buffer}$
$\mathrm{pUnicode} \mid \mathrm{pImmbyteslike} \mid \mathrm{pNone}\ \overset{\mathtt{z\#}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ (\mathrm{const\ char\ *}, \mathrm{int} \mid \mathrm{Py\_ssize\_t})$
$\mathrm{pImmbyteslike}\ \overset{\mathtt{y}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{const\ char\ *}$
$\mathrm{pByteslike}\ \overset{\mathtt{y*}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{Py\_buffer}$
$\mathrm{pImmbyteslike}\ \overset{\mathtt{y\#}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ (\mathrm{const\ char\ *}, \mathrm{int} \mid \mathrm{Py\_ssize\_t})$
$\mathrm{pBytearray}\ \overset{\mathtt{w*}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{Py\_buffer}$
$\mathrm{pUnicode}\ \overset{\mathtt{u}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{const\ Py\_UNICODE\ *}$
$\mathrm{pUnicode}\ \overset{\mathtt{u\#}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ (\mathrm{const\ Py\_UNICODE\ *}, \mathrm{int} \mid \mathrm{Py\_ssize\_t})$
$\mathrm{pUnicode} \mid \mathrm{pNone}\ \overset{\mathtt{Z}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{const\ Py\_UNICODE\ *}$
$\mathrm{pUnicode} \mid \mathrm{pNone}\ \overset{\mathtt{Z\#}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ (\mathrm{const\ Py\_UNICODE\ *}, \mathrm{int} \mid \mathrm{Py\_ssize\_t})$
$\mathrm{pBytes}\ \overset{\mathtt{S}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{PyBytesObject\ *}$
$\mathrm{pBytearray}\ \overset{\mathtt{Y}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{PyByteArrayObject\ *}$
$\mathrm{pUnicode}\ \overset{\mathtt{U}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ \mathrm{Py\_UNICODE\ *}$
$\mathrm{pUnicode}\ \overset{\mathtt{es}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ (\mathrm{const\ char\ *}, \mathrm{char\ **})$
$\mathrm{pUnicode} \mid \mathrm{pBytes} \mid \mathrm{pBytearray}\ \overset{\mathtt{et}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ (\mathrm{const\ char\ *}, \mathrm{char\ **})$
$\mathrm{pUnicode}\ \overset{\mathtt{es\#}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ (\mathrm{const\ char\ *}, \mathrm{char\ **}, \mathrm{int} \mid \mathrm{Py\_ssize\_t})$
$\mathrm{pUnicode} \mid \mathrm{pBytes} \mid \mathrm{pBytearray}\ \overset{\mathtt{et\#}}{\underset{\mathrm{p}}{\longmapsto}}_{\mathrm{c}}\ (\mathrm{const\ char\ *}, \mathrm{char\ **}, \mathrm{int} \mid \mathrm{Py\_ssize\_t})$

字符串和缓冲区类型的格式化单元允许把对象作为一块连续的内存进行操作。常见的格式化单元 `s` 接收一个 Python Unicode 类型的对象作为参数，并将其转换为指向一个字符串的 C 指针类型变量。该格式不接受字节形式的对象，如文件系统的路径。`pImmbyteslike` 是不可变（immutable）字节形式对象 `pBytes`、`pArray`、`pMemoryview` 的和类型。`pByteslike` 包括 `pImmbyteslike` 和可变的字节数组 `pBytearray`。

一些 Python 参数的转换过程（`#` 后缀的格式化单元）会额外创建另一个 `int` 或 `Py_ssize_t` 类型的 C 变量，其保存参数对象的长度。当一个格式化单元设置了一个指向缓冲区的指针，该缓冲区将由对应的 Python 对象管理，并共享 Python 对象的生命周期，这使得开发者不需要关心内存释放的问题。只有 `es`、`es#`、`et`、`et#` 四个格式化单元例外，它们使用另一个 `char **` 类型的 C 变

表 4.2　格式化单元施加的参数类型转换规则（续）

数值
pInt_nonnegative $\overset{\texttt{b}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ unsigned char
pInt_nonnegative $\overset{\texttt{B}}{\mathrel{\mathrm{p}\mapsto_{\mathrm{c}}}}$ unsigned char
pInt $\overset{\texttt{h}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ short int
pInt_nonnegative $\overset{\texttt{H}}{\mathrel{\mathrm{p}\mapsto_{\mathrm{c}}}}$ unsigned short int
pInt $\overset{\texttt{i}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ int
pInt_nonnegative $\overset{\texttt{I}}{\mathrel{\mathrm{p}\mapsto_{\mathrm{c}}}}$ unsigned int
pInt $\overset{\texttt{l}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ long int
pInt_nonnegative $\overset{\texttt{k}}{\mathrel{\mathrm{p}\mapsto_{\mathrm{c}}}}$ unsigned long
pInt $\overset{\texttt{L}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ long long
pInt_nonnegative $\overset{\texttt{K}}{\mathrel{\mathrm{p}\mapsto_{\mathrm{c}}}}$ unsigned long long
pInt $\overset{\texttt{n}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ Py_ssize_t
pBytes[1] | pBytearray[1] $\overset{\texttt{c}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ char
pUnicode[1] $\overset{\texttt{C}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ int
pFloat $\overset{\texttt{f}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ float
pFloat $\overset{\texttt{d}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ double
pComplex $\overset{\texttt{D}}{\mathrel{\mathrm{p}\Mapsto_{\mathrm{c}}}}$ Py_complex
对象
pObject $\overset{\texttt{O}}{\mathrel{\mathrm{p}\mapsto_{\mathrm{c}}}}$ PyObject *
pObject $\overset{\texttt{O!}}{\mathrel{\mathrm{p}\mapsto_{\mathrm{c}}}}$ (α *, PyObject *)
pObject $\overset{\texttt{O\&}}{\mathrel{\mathrm{p}\mapsto_{\mathrm{c}}}}$ ((PyObject *, void *) → status, any *)
pBool $\overset{\texttt{p}}{\mathrel{\mathrm{p}\mapsto_{\mathrm{c}}}}$ status
$(\tau_1, \ldots) \overset{\texttt{(format unit 1, ···)}}{\mathrel{\mathrm{p}\longmapsto_{\mathrm{c}}}} (\alpha_1, \ldots)$
特殊字符： `| $ : ;`

量来指明参数转换过程使用的编码，如 utf-8，开发者也需要负责回收转换后的对象。当转换后的参数填充到 `Py_buffer` 结构体时（带有 `*` 后缀的格式化单元），其底层的缓冲区是带锁的，因此调用者在后续使用（可能在多线程块中）该缓冲区时不用担心可变对象被修改或回收。

在表 4.2 中，数值类型的格式化单元使用双线转换箭头 $\overset{\texttt{format unit}}{\mathrel{\mathrm{p}\Longmapsto_{\mathrm{c}}}}$ 表示在参数类型转换时施加额外的运行时溢出检查。相对于 C 中不同位长的整数类型，Python 支持长整型的语言特性。本章的类型推断系统使用基于类型精化[198]的子类型实现编译时的溢出检查。精化类型是具有谓词的类型，假定该谓词对于精化类型的任意元素成立。以 `pInt_nonnegative` 为例，Python 的长整型可以被分成更多区间以和 C 中不同位长的整型相对应。`Py_ssize_t` 类型

和编译器的 size_t 类型长度相同，但是有符号。因此使用 Py_ssize_t 替代 C int 能够避免在 64 位机器上只能索引 2^{31} 个元素的序列的问题[179]。*pType*[n] 表示对 *pType* 的额外的长度约束，而非表示数组。例如，格式化单元 C 把一个长度为 1 的 Unicode 对象表示的 Python 字符转换为 C int 类型的变量。

对象类型的格式化单元把 Python 对象存入 C 对象指针。与 S、Y、U 等字符串与缓冲区类型的格式化单元不同，格式化单元 O 只进行跨语言的传递而不施加类型转换，外部函数的 C 实现直接接收 Python 侧传入的参数对象。O! 通过另一个与 Python 类型对应的 C 结构体的指针来指明传入外部函数的 Python 对象应有的类型，从而在类型错误时抛出运行时异常。O& 通过一个自定义的转换函数把 Python 对象转换为 C 对象。根据转换函数的不同实现，转换后的 C 变量可以是任意类型，转换结果存储在 void * 类型的地址中。转换函数通过返回一个整型值 status，值为 1 表示转换成功，值为 0 表示转换失败。格式化单元 p 通过布尔谓词测试传入值的真假，并把结果转换为 C 中等价的 1、0 值。如果一个积类型的 Python 序列被作为一个参数传入外部函数，其转换规则由括号包裹的格式化单元序列指明。括号包裹的格式化单元序列可以嵌套，对应于嵌套的积类型。

格式化串还允许一些不指明类型转换、而是具有其他特殊含义的字符。| 指明后续的参数是可选的。$ 指明后续的参数是关键字参数。: 和 ; 表示格式化单元的终止，其后续的字符串会被用作错误处理。

区别于其他基于文本的类型推断方法[159,199]，本节基于严格的参数类型转换规则的格式化串解析是可靠的、确定性的。

4.4.5　返回类型转换

返回类型转换刻画在外部函数的 C 函数实现中构建宿主 Python 侧返回值的过程中涉及的跨语言类型转换，其包含四种形式。

1. 值构建分析

Python 支持多返回的语言特性，为了使 C 外部函数实现兼容这一特性，一族用于值构建的 Python/C API 函数 $\mathbb{I}_{\text{vb}}$ 使用和参数解析族 Python/C API 函数 $\mathbb{I}_{\text{ap}}$ 相似的基于格式化串的类型转换约束。用于值构建的 Python/C API 函数 $\iota_{\text{vb}} \in \mathbb{I}_{\text{vb}}$ 根据格式化串创建一个新值，该过程遵循与表 4.1 和续表 4.2 反向的规则。例如，图 4.1(b) 第 11 行的 PyLong_FromLong(x*y) 等价于使用值构建 Python/C API 函数 Py_BuildValue("I", x*y)，此处在值构建 Python/C API Py_BuildValue 中，其格式化单元 I 将 C unsigned int 转换为一个

Python 整型对象。值构建的规则可以表示为：

$$\Gamma^{\mathrm{c}} \vdash \beta_j \ {}_{\mathrm{c}}\overset{\iota_{\mathrm{vb}}.u_j}{\longmapsto}_{\mathrm{p}} \eta_j \quad 1 \leqslant j \leqslant m, \eta = (\eta_1, \cdots, \eta_m) \qquad \text{（值构建）} \qquad \text{(Rvb)}$$

若干个类型为 $\beta_1, \cdots, \beta_m$ 的 C 变量基于值构建 Python/C API ι_{vb} 的格式化串 "$u_1, \cdots, u_n$" 被分别转换为 $\eta_1, \cdots, \eta_m$ 类型的 Python 对象，元组 $\eta = (\eta_1, \cdots, \eta_m)$ 为支持多返回特性的外部函数的返回类型。

2. 显式转换分析

当外部函数仅返回一个非元组类型的 Python 对象，即外部函数是一个单返回函数时，显式转换 Python/C API 能够直接指明单个 C 变量到 Python 对象的返回类型转换规则，例如图 4.1 中的 `PyLong_FromLong`。显式转换规则可以表示为：

$$\Gamma^{\mathrm{c}} \vdash \beta \ {}_{\mathrm{c}}\overset{\iota_{\mathrm{ec}}}{\longmapsto}_{\mathrm{p}} \eta \qquad \text{（显式转换）} \qquad \text{(Rec)}$$

显式返回类型转换 Python/C API 集合 $\mathbb{I}_{\mathrm{ec}}$ 包含以下几种形式的 Python/C API：（1）`PyPT_FromCT`(如 `PyLong_FromLong`)将 C 类型 `CT` 的 C 变量转换为 Python 类型 `PT` 的 Python 对象，（2）`PyPT_New`（如 `PyList_New`）返回一个 Python 类型 `PT` 的新 Python 对象，（3）`Py_PT`（如 `Py_None`）其自身是一个 Python 类型 `PT` 的对象。

3. 类型转换分析

C 程序支持作为右结合操作符的类型转换（type cast），对于形如 `return` $(\beta_1)\cdots(\beta_n)e$ 的返回表达式，其类型转换规则可以表示为：

$$\Gamma^{\mathrm{c}} \vdash (\beta_1)\cdots(\beta_n)e :: \beta_1 \quad \beta_1 \ {}_{\mathrm{c}}\mapsto_{\mathrm{p}} \eta_1 \qquad \text{（类型转换）} \qquad \text{(Rtc)}$$

作为 Python 外部函数的 C 实现的返回，C 类型 β_1 必须是用来描述 Python 内置类型的 C 结构体。例如，在没有其他跨语言类型转换规则作用时，`Py_UNICODE` ${}_{\mathrm{c}}\mapsto_{\mathrm{p}}$ `pUnicode` 是一个固有的一一映射。

4. 可达定义分析

```
1 T1 result, tmp;
2 result = Py_BuildValue(...); // T2
3 if (...) {
4   tmp = PyT3_New(...);
5   result = tmp;
6 }
7 return result;
```

图 4.7　返回类型转换的一个复杂例子

考虑如图 4.7 所示的一个更加复杂的例子。变量 `result` 被声明为类型 `T1`，在跨语言接口代码中声明类型一般为 `PyObject *`，通过分析如前所述不同类型的 Python/C API 调用或赋值语句可以得到其精化类型。可达定义分析 $\mathcal{T}_{\mathrm{RDA}}$

（Reaching Definition Analysis）分析过程内的类型传递信息。$\mathcal{T}_{\text{RDA}}$ 可以调用前三小节所述的返回类型转换分析规则。例如在图 4.7 中，外部函数实现的返回变量 `result` 的类型可能是 `T1`、`T2` 或变量 `tmp` 实际的类型。类型 `T1` 可以通过分析变量声明语句得到，类型 `T2` 可以通过对 Python/C API `Py_BuildValue` 的值构建分析得到，而 `tmp` 的类型可以通过递归调用可达定义分析计算，并最终通过对 Python/C API `PyT3_New` 的显式转换分析得到 `tmp` 的精化类型 `T3 <: T1`。当存在子定型规则成立时，类型列表 `[T1, T2, T3]` 可以合一[200]。使用可达定义分析 $\mathcal{T}_{\text{RDA}}$ 的类型推理规则表示为：

$$\Gamma^{\text{c}} \vdash e :: \mathcal{T}_{\text{RDA}}(e) \quad \text{（可达定义）} \tag{Rrd}$$

$\mathcal{T}_{\text{RDA}}$ 以返回表达式中的返回变量名 e 作为输入，对外部函数实现做上述的函数内的分析得到返回变量 e 的类型。

4.4.6　总结

对于外部函数类型推断规则 (TInfer) 的三部分前提，外部函数声明 (D) 只有一种形式，参数类型转换 (P) 有两种形式：调用惯例分析 (Pcc) 、参数解析分析 (Pap)，返回类型转换 (R) 有四种形式：值构建分析 (Rvb)、显式转换分析 (Rec)、类型转换分析 (Rtc)、可达定义分析 (Rrd)。

(D)、(P)、(R) 的具体形式及其组合将确定推断结果中外部函数的参数类型和返回类型。举例来说，以下规则是对一种常见模式的外部函数的类型推断，这类外部函数基于参数解析 Python/C API 指明参数类型转换，基于值构建 Python/C API 指明返回类型转换。

$$\Gamma^{\text{c}} \vdash f^{\text{p}} \xleftarrow{flag} f^{\text{c}} \tag{TInfer: D-Pap-Rvb}$$

$$\frac{\Gamma^{\text{c}} \vdash \tau_i \,{}_{\text{p}}\overset{\iota_{\text{ap}}.u_i}{\longmapsto}_{\text{c}}\, \alpha_i \quad 1 \leqslant i \leqslant n \qquad \Gamma^{\text{c}} \vdash \beta_j \,{}_{\text{c}}\overset{\iota_{\text{vb}}.u_j}{\longmapsto}_{\text{p}}\, \eta_j \quad 1 \leqslant j \leqslant m \quad \eta = (\eta_1, \cdots, \eta_m)}{\Gamma^{\text{p}} \vdash f^{\text{p}} :: pFunc([\tau_1, \cdots, \tau_n], (\eta_1, \cdots, \eta_m))}$$

另一种常见的外部函数模式是通过显式转换 Python/C API 返回的无参外部函数。

$$\Gamma^{\text{c}} \vdash f^{\text{p}} \xleftarrow{flag} f^{\text{c}} \tag{TInfer: D-Pcc-Rec}$$

$$\frac{\{\mathcal{P}_{\text{PFA}}(flag) \wedge \mathcal{P}_{\text{UPA}}(f^{\text{c}})\}?\{\Gamma^{\text{c}} \vdash \texttt{pNone} \,{}_{\text{p}}\mapsto_{\text{c}}\, \texttt{void}\} \qquad \Gamma^{\text{c}} \vdash \beta \,{}_{\text{c}}\overset{\iota_{\text{ec}}}{\longmapsto}_{\text{p}}\, \eta}{\Gamma^{\text{p}} \vdash f^{\text{p}} :: pFunc(\texttt{pNone}, \eta)}$$

所有的 8（$1 \times 2 \times 4$）种变体组成了 Python 的 C 外部函数的类型推断系统。

对于图 4.1 所示的例子，外部函数 `foo` 通过 `PyMethodDef` 声明，其对应的外部函数实现 `_foo` 使用参数解析 Python/C API `PyArg_ParseTuple` 进行

参数类型转换，使用显式转换 Python/C API `PyLong_FromLong` 进行返回类型转换。因此对于该例，由跨语言接口代码中的隐式信息可以确定如下的类型推断规则：

$$\Gamma^{\mathrm{c}} \vdash f^{\mathrm{p}} \xleftarrow{flag} f^{\mathrm{c}} \qquad \text{(TInfer: D-Pap-Rec)}$$

$$\Gamma^{\mathrm{c}} \vdash \tau_i \overset{t_{\mathrm{ap}}.u_i}{\underset{\mathrm{p}}{\longmapsto}} \underset{\mathrm{c}}{\alpha_i} \quad 1 \leqslant i \leqslant n$$

$$\frac{\Gamma^{\mathrm{c}} \vdash \beta \overset{t_{\mathrm{ec}}}{\underset{\mathrm{c}}{\longmapsto}} \underset{\mathrm{p}}{\eta}}{\Gamma^{\mathrm{p}} \vdash f^{\mathrm{p}} :: pFunc([\tau_1, \cdots, \tau_n], \eta)}$$

进而结合具体规则分析对应的 Python/C API 可以推出：

$$\Gamma^{\mathrm{p}} \vdash \texttt{foo} :: pFun([\texttt{pInt\_nonnegative}, \texttt{pInt\_nonnegative}], \texttt{pInt})$$

即由 C 侧的外部函数实现可以推出，Python 外部函数 `foo` 在宿主侧的类型签名为接收两个整型参数并返回一个整型变量，且两个入参变量的值应当非负。

4.5　原型实现

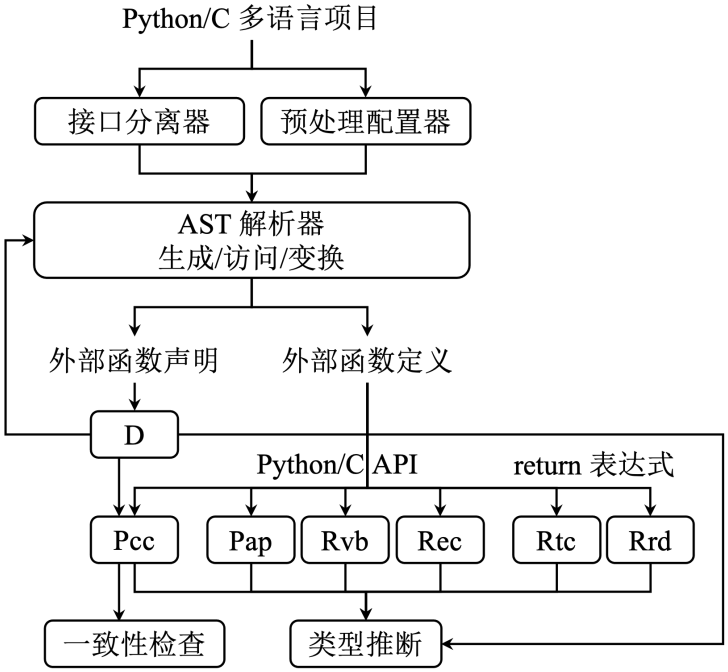


图 4.8　PyCType 架构概览

PyCType 是本章提出的 Python 的 C 外部函数的静态类型推断的原型系统实现，图 4.8 所示是其整体架构。接口分离器从 Python/C 多语言项目中分离出跨语言接口代码。预处理配置器配置解析跨语言接口代码所需的头文件，导入 Python/C 多语言软件的项目内依赖、Python/C API 定义、系统与第三方库依赖。AST 解析器基于 Python 实现的完整的 C99 解析器 pycparser[201] 实现。在生成跨语言接口代码的 AST 后，部分分析模块基于 AST 访问实现。当某个分析需要其

他中间形式如控制流图（Control Flow Graph，CFG）的代码中间表示时，AST 变换模块以部分子节点作为输入进行变换，从而提高时间效率，减小中间表示体积。主要分析的 AST 子节点包括外部函数声明和外部函数定义（对应图 4.3 的抽象语法）。外部函数声明是特定类型的结构体。分析器 D 由外部函数声明提取类型推断的前提 (D)。AST 解析器根据外部函数声明中外部函数定义的函数指针搜索外部函数定义。外部函数定义的函数体和外部函数声明的调用惯例标记一起用于调用惯例分析器 Pcc。其他的分析模块都作用于外部函数定义，其中参数解析分析 Pap、值构建分析 Rvb、显式转换分析 Rec 基于特定的 Python/C API，类型转换分析 Rtc 和可达定义分析 Rrd 作用于返回表达式。(D)、(P)、(R) 符合某一形式的分析结果组合用于外部函数的类型推断，(Pcc) 的门限语义谓词则用于外部函数声明与实现的一致性检查。

4.6　评估

本节将评估类型推断系统的原型实现 PyCType 的有效性，介绍外部函数声明与实现不一致的漏洞的检查，并讨论本文的实验方法的有效性威胁。

4.6.1　实验评估

PyCType 基于形式化的类型推断规范和保守的必须（must）静态分析，其结果在设计上是可靠的。本节通过两组实验分析其完备性和有效性。

1. 完备性

可靠和完备一般无法同时实现，评估 PyCType 完备性的实验基于 CPython、NumPy、Pillow 这三个代表性的、广泛使用的 Python/C 多语言项目。CPython 是 Python 语言的默认实现，其标准库包含大量 C 外部函数，提供加解密、解压缩、操作系统接口等功能。NumPy 是 Python 中科学计算的基础包，它是 Python 在科学计算应用中存储多维数据的事实标准。Pillow 是 Python 事实上的图像处理标准库，它提供了丰富的文件格式支持、高效的内部表示、强大的图像处理能力。C 扩展模块赋予了这些性能敏感的库良好的计算效率。

表 4.3　参数类型推断的完备性

项目	KLOC	时间 (s)	外部函数	Pcc	Pap	一致性漏洞	覆盖率
CPython	279.2	159.7	1529	503	606	32	74.6%
NumPy	507.1	76.4	90	9	51	6	73.3%
Pillow	29.0	40.7	132	2	119	10	99.2%
总计	815.3	276.8	1751	514	776	48	76.4%

表 4.3 所示是外部函数的参数类型推断的完备性实验结果。参数类型相比返回类型（以单返回为主）包含更多变量，并且可以直接用于类型检查。第二列

KLOC 列出了对应项目中 C 代码的千代码行数。整体的分析时间和项目中的外部函数数成正相关。平均地，PyCType 基于参数类型转换 (P) 的两类推断规则 (Pcc) 和 (Pap) 推理一个外部函数的参数类型签名大约花费 158 ms。最后一列覆盖率计算多少外部函数能够通过这两个规则进行推断或报告漏洞。覆盖率和 Python/C 多语言项目的跨语言接口代码的具体实现有关，从 73.3% 到 99.2% 不等。当前不能覆盖的推理情形总计不超过 24%。例如，Python/C API 族 `PyPT_AsCT` 施加与 `PyPT_FromCT` 相反的类型转换约束。该族 Python/C API 可以把单个 Python 对象转换为 C 变量。`PyPT_AsCT` 可以视作仅带有单个简单（非括号包裹的）格式化单元的参数解析族 Python/C API。类型推断系统的完备性可以通过给出 `PyPT_AsCT` 等更多 Python/C API 的类型语义和推断规范来提高。

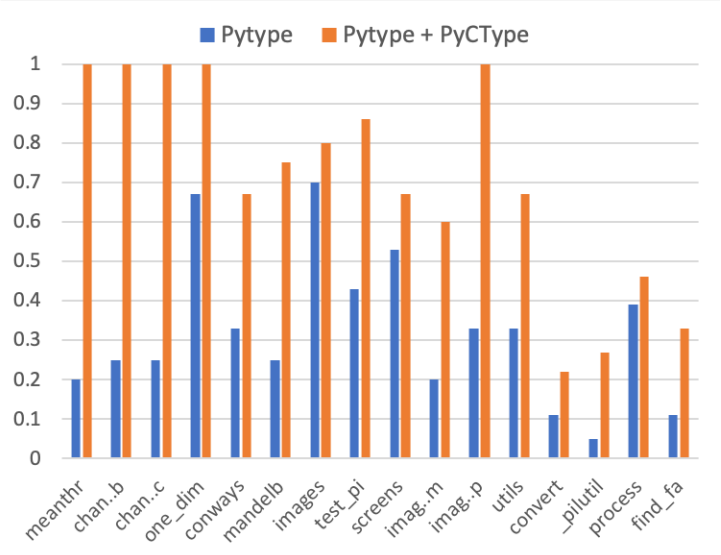
2. 有效性

PyCType 在 Pillow 上推断了高达 99.2% 的外部函数的类型签名，对其类型推断结果的人工检查证明了方法的可靠性，即 PyCType 给出的类型推断都是正确的。但是一些类型推断结果可能并不足够准确，比如只能保守地推断为 `pObject` 类型。然而，一定程度的不完备和不准确是可以接受的。对于某个未推出（精确）类型的外部函数的返回对象，其可能作为其他外部函数或本地函数的参数，而该参数类型是可（精确）推出或已知的。出于这一原因，PyCType 的有效性评估被设计为一个类型推断增强实验。来自 Google 公司的先进的 Python 静态类型推断工具 Pytype[193] 不支持外部函数，它内置了 Typeshed[197] 作为外部类型标注。通过把 PyCType 的类型推断结果编码为 Typeshed 使用的类型提示[158]，将其作为对 Pytype 外部类型标注的补充，可以评估 PyCType 的有效性。

有效性实验在完备性实验表现最好的 Pillow 上进行。Pillow 有 132 个外部函数，PyCType 能够推断其中 121 个的类型签名，其中 63 个是 `ImagingCore` 类的外部类方法，46 个是 `Image` 模块的外部函数，将这 109 个外部函数的推断类型签名编码为类型提示。测试集是使用 GitHub API 筛选出的星标高于 3 万的使用 Pillow 的项目，这些项目来自不同的应用领域，包括图像处理、机器学习、网络框架等。表 4.4 是类型推断增强的实验结果。实验中出于以下目的对部分测试文件进行了简单的重写：(1) 移除已有的类型标注，(2) 移除没有使用 Pillow 的辅助函数，(3) 将嵌套函数调用 `a=f1(f2())` 展开为 `b=f2() a=f1(b)`，(4) 所有函数调用的返回都赋值给一个变量，即无返回外部函数的返回类型应当为 `pNone`。Pytype 列记录了使用未修改的 Pytype 时变量的类型推断率（召回精度）。Pytype+PyCType 列是使用 PyCType 的类型推断结果作为补充后的精度，如表 4.4 最后一行的类型推断增强的提升效果对比图所示，类型召回精度在不同项目上提升了 7% 至 80% 不等，在所有测试项目上平均提升 27.5%。除了 Pillow，一些项目还使用了其他包含外部函数的包，使用 PyCType 静态推断这些包作为补充

表 4.4　类型推断增强实验

文件	项目	星标数	应用领域	Pytype	Pytype + PyCType
meanthreshold.py	TheAlgorithms/Python	105k	图像算法	20%	100%
change_brightness.py				25%	100%
change_contrast.py				25%	100%
one_dimensional.py				67%	100%
conways_game_of_life.py				33%	67%
mandelbrot.py				25%	75%
images.py	django/django	57.2k	网络框架	70%	80%
test_pipeline_images.py	scrapy/scrapy	40.5k	网络爬虫	43%	86%
screenshots.py	apache/superset	38.3k	数据可视化	53%	67%
image_mobject.py	3b1b/manim	33.4k	动画引擎	20%	60%
image_processing.py	home-assistant/core	42.4k	智能家居	33%	100%
utils.py	tensorflow/models	69.8k	机器学习	33%	67%
convert_to_tflite.py				11%	22%
_pilutil.py	scikit-learn/scikit-learn	45.6k		5%	27%
processing_image.py	huggingface/transformers	45.5k		39%	46%
find_faces_in_picture.py	ageitgey/face_recognition	39.8k		11%	33%



可以进一步提升类型推断的增强效果。

4.6.2　一致性漏洞

调用惯例分析 (Pcc) 的谓词条件是分别来自无参分析 (PFA) 和未使用形参分析 (UPA) 的两个门限语义谓词 $\mathcal{P}_{\text{PFA}}$ 和 $\mathcal{P}_{\text{UPA}}$ 的和取范式。只有当 $\mathcal{P}_{\text{PFA}}$ 和 $\mathcal{P}_{\text{UPA}}$ 相互匹配且都为真时，调用惯例分析 (Pcc) 的判断才成立。当它们相互匹配且

都为假时，外部函数接收至少一个参数，PyCType 会根据参数解析分析 (Pap) 等参数类型转换规则继续进行推断。此外，实验过程中确实发现了一些 $\mathcal{P}_{\text{PFA}}$ 和 $\mathcal{P}_{\text{UPA}}$ 不匹配的情形，这种冗余但不一致的隐式类型信息会导致类型推断系统挂起并报告一个漏洞警告。人工检查所有警告发现不存在误报，这也进一步证明了类型推断系统的可靠性[①]。在这些错误情形中，基于调用惯例标记的无参分析 $\mathcal{P}_{\text{PFA}}$ 为假，指明外部函数至少接收一个某种类型的参数；而未使用形参分析 $\mathcal{P}_{\text{UPA}}$ 为真的含义则相反，它指明外部函数不应接收任何参数，因为外部函数实现根本不会解析并使用传入的参数。该漏洞可以对应到单语言情形中的编译器警告 `-Wunused-parameter`[202]。从类型系统的视角描述，外部函数定义给出的外部函数参数类型约束为 `pNone`，而由外部函数声明指明的参数类型为 *pUnion*(`pNone`, ⋯)，使得实际上该外部函数可以接收任意类型的参数。对于 PyCType 在 CPython、NumPy、Pillow 中发现的 48 个一致性漏洞，其中 5 个对于外部函数实现未使用的 `args` 参数标记了编译属性 `__unused__`。增加编译属性虽然不能改变类型系统的错误，但是避免了潜在的利用该漏洞的安全攻击。目前 NumPy 和 Pillow 社区已经确认并修复了其中的 8 个一致性漏洞[203-204]。

4.6.3　评估效果总结

本章提出的外部函数的静态类型推断系统 PyCType 基于形式化的类型推断规范和保守的必须静态分析，能够得到可靠的类型推理结果。与此同时，在 CPython 标准库、NumPy、Pillow 等广泛使用的 Python/C 多语言软件的 1751 个外部函数上的完备性实验表明，PyCType 能够推断出 76.4% 的外部函数的参数类型。进一步地，在基于 Pillow 开发的流行多语言软件上的实验表明，使用 PyCType 增强谷歌公司的单语言类型推断工具 Pytype 能够使其精度提高 27.5%。此外，基于 PyCType 的一致性漏洞检查能够无误报地检查外部函数实现与声明不一致的漏洞。

4.6.4　有效性威胁

本文在类型推断和一致性漏洞检查的实验评估过程中分别存在一些有效性威胁。对于类型推断系统 PyCType，虽然前文给出了形式化的类型系统，但是并没有通过机械验证等方法对类型推断的可靠性加以严格证明，而是对 Pillow 上的类型推断结果进行了人工校验，同时借助一致性漏洞检查的实验结果在一定程度上验证了类型推断的可靠性。此外，在评估类型推断系统的完备性时只考虑了参数类型，因为参数类型可以直接用于外部函数调用的类型误用检查，而返回

[①]这里的可靠性是对类型推断而言的。在漏洞检查的研究中，可靠一般指没有漏报。如果将本节的一致性漏洞检查作为一个独立的系统，其应表示为类型推断系统中调用惯例分析的谓词条件的否定命题。

类型以相对简单的单返回为主，并且可以结合单语言程序的类型推断工具进行传播分析。由于 Python/C 多语言程序上没有其他确定性的外部函数类型推断工作进行对比，因此有效性实验设计为对单语言类型推断工具 Pytype 的增强实验，而增强 Pytype 的实验过程需要一些人工辅助，包括使用特定格式编码类型签名，以及修改部分测试程序。在一致性漏洞检查的实验过程中，部分漏洞警告来自内部函数，如 CPython 编译器内部实现中定义的一些外部函数，这些函数有特定的使用方式，并且不希望暴露给外部开发者。尽管它们在内部遵循特定使用方式时不会触发程序错误，但是它们在定义和实现上确实存在不一致性，并且可以通过增加特定前缀的方式被外部调用。

4.7　本章小结

本章提出了 Python 的 C 外部函数的静态类型推断系统 PyCType。PyCType 的类型推断规则表示为三部分可组合的前提判断，其建模并分析 Python/C 跨语言接口代码中类型转换的隐式信息。在三个代表性的 Python/C 多语言软件上的实验表明，PyCType 能够可靠地推断大多数外部函数的类型签名。作为对 Python 静态类型推断工具 Pytype 的补充，PyCType 能够提升其在包含外部函数调用的 Python 程序上的精度。PyCType 能够检查外部函数的声明与实现不一致的漏洞，该漏洞会导致无参外部函数可以接收任意类型的参数，部分漏洞发现已被社区确认并修复。

第 5 章　支持多种宿主语言的跨语言调用图构建

现代软件系统越来越多地采用多语言的架构，软件系统的不同组件由不同的编程语言开发，并基于外部接口实现宿主语言和外部语言之间的互操作，从而复用已有的库并组合语言特性。以 C/C++ 作为互操作的外部语言的 C 外部接口几乎是所有主流编程语言的语言标准的一部分。第 3 章和第 4 章研究了 Python 和 C/C++ 之间的 C 外部接口 Python/C API 的设计迭代、使用行为、漏洞模式与漏洞检查，以及包括类型系统等复杂语言特性在内的多语言软件的程序分析等问题。但是，多语言软件系统的广泛流行也对跨语言程序分析提出了更多需求，包括提高跨语言的程序分析技术对不同互操作语言和外部接口的泛化能力。作为第 4 章外部函数类型推断的前提之一的外部函数声明 (D) 建立宿主侧函数调用和外部侧函数实现之间的映射关系。虽然其在 PyCType 中只有一种形式，但是相似的映射关系以各种不同的语法和范式出现在其他多语言互操作方式中。由特殊到一般地，本章回答一个跨语言基础程序分析问题，即构建连接宿主语言和外部语言的调用图，提出了一种支持不同宿主语言调用 C/C++ 外部函数的跨语言调用图构建方法。本章建模了外部函数声明的外部接口的语义以描述跨语言的调用关系，包括语义抽象以支持不同的宿主语言，同时定义了图变换和节点融合算法以构建完整的跨语言调用图。实验表明本章的方法能够有效地自动构建 Python 和 JavaScript 调用 C/C++ 的静态调用图。该跨语言的调用图构建能够作为集成开发环境（Integrated Development Environment，IDE）的语言服务高效运行并和其他工具集成。

5.1　引言

多语言的软件架构广泛使用于实际工程实践。通过多语言互操作的方式，开发者能够混合和匹配不同编程语言的优势，并复用已有的优化库。统计发现，GitHub 上星标数最多的 60 个 C++ 项目中，有 24 个存在和其他高级编程语言的互操作。在多语言软件中，宿主语言通过外部接口调用外部语言，最常见的外部语言是低级语言 C/C++，因此 C 外部接口是多数高级语言的默认外部接口。不同高级语言以不同的机制和实现提供其 C 外部接口。根据 Lee 等人[101] 的分类，Python、Rust 等语言提供语言级的外部接口[172,205]，而 Java 和 JavaScript 则提供运行时环境特定的外部接口 Java 本地接口[206]（Java Native Interface）和 Node.js C++ addons[207]。许多主流软件系统通过使用这些 C 外部接口构建多语言软件架构。例如，机器学习框架 PyTorch[14] 和 TensroFlow[13]、图像处理库 Pillow[146]、科

学计算库 NumPy[9] 都使用 Python 作为宿主语言、C/C++ 作为外部语言。Python/C 多语言架构结合了 Python 的开发效率和 C/C++ 的执行性能。

尽管实用且常见，但是编写安全可靠的多语言软件并不容易。考虑 Java/C 多语言软件的漏洞[38]和 Python/C 多语言软件的漏洞（本文第 3 章），开发者需要在跨语言编程时考虑互操作语言在语言特性上的差异，如内存管理、异常处理、类型系统等。举例来说，Python 整型可以有任意的长度，并且负数可以用来反向索引；而在 C 中整数根据类型的不同有固定的长度，并且数组的索引不能为负。这导致虽然不同组件在单语言视角下都实现正确，但其组成的多语言软件仍然可能存在溢出的风险。更糟糕的是，由于异常处理机制的不同，外部代码引发的漏洞往往难以调试。已有的研究表明 JavaScript 包可能依赖不安全的代码[208-209]，同时错误定位对于多语言软件而言是困难的[210]。

本章提出了一种静态程序分析技术，用以构建不同宿主语言和外部 C/C++ 互操作的多语言软件中的跨语言调用图。调用图为系统中的每个函数构建一个节点，而从节点 f 到节点 g 的有向边 $f \to g$ 则表示函数 f 可能调用函数 g[211]。本章的方法首先根据函数声明的位置把函数节点分成三个子集：宿主侧、跨语言接口层、外部侧，并在这三个域中分别构建调用子图。然后通过定义图变换和节点融合算法来构建完整的跨语言调用图。宿主侧和外部侧的调用子图是传统的单语言调用图。跨语言接口层的调用子图则基于外部函数声明的外部接口的语义建模。这一抽象是语言泛化的，可以支持不同的宿主语言与 C 外部接口，甚至跨语言接口生成工具，如 SWIG[121]、pybind11[212] 等。函数节点融合通过合并可能有相同的函数实现的函数节点来连接变换之后的子图。

调用图构建作为一种基础程序分析技术被许多客户程序分析任务所使用，包括错误定位[213-214]、安全扫描[215]、影响分析[216-217]，以及 IDE 功能[218-219] 如代码定位、补全、重构，等等。学界和业界设计实现了许多调用图构建的技术方法。大多数工作只能支持单一语言程序，例如 Nielsen 等人[215]、Yin 等人[220] 针对 JavaScript 的工作，Salis 等人[221]、Horner 等人[222]、Eads 等人[223] 针对 Python 的工作，Fabry 等人[224] 针对 Go 的工作，等等。少数工作基于某种中间表示构建调用图，它们可以支持多种编程语言。但是把不同的语言编译到其中间表示并不容易，需要较大的工作量，且难以支持所有语言特性，典型的就是难以支持外部函数调用。这些工作支持构建不同语言的调用图，具有了泛化性；但语言之间不能互操作，本质仍然是基于中间表示语言的单语言问题。例如 Rogowski 等人[225] 针对多种动态语言的工作，以及 Arzt 等人[226] 在针对 Java 字节码的分析平台 Soot[227] 上兼容 C# 等语言的中间表示 CIL 的工作。支持 Java/C 互操作的多语言软件的调用图构建的工作，如 Lee 等人[101]、Fourtounis 等人[228] 的工作，分别从源码和二进制代码的角度分析外部 C 程序，进而和 Java 侧的外部函数声

明匹配得到跨语言的调用关系。这种跨语言调用分析依赖 Java 基于同名符号的外部函数声明语法，不需要分析外部接口，对于其他多语言软件不具有一般性。Python/C 多语言软件的调用分析工具 ffi-navigator[229] 基于正则模式匹配分析外部函数声明 Python/C API，其依赖熟悉被分析软件的专家定制提取规则，无法适用于任意 Python/C 多语言软件，或是其他宿主语言和互操作方法。

支持多语言互操作的软件架构是本章提出的跨语言调用图分析的核心特性。该分析可以分成两个阶段。其一，离线分析阶段分析多语言软件包得到跨语言接口层调用子图的总结（summary），该总结代表外部函数声明中的映射关系。其二，在使用该多语言软件包时在线地构建完整的调用图。这种模块化的方法能够重用对多语言软件包的分析总结，既符合现实环境中的开发实践，又节约了跨语言调用图构建的整体时间开销。

本章的主要贡献如下：

- 本章提出跨语言程序的调用图构建分析 Frog。Frog 首先分析外部函数声明以提取外部映射关系，然后在变换后的子图上融合函数节点以构建完整的调用图。
- 本章提出了一种语言泛化的模型来表示不同宿主语言和接口生成器的外部函数声明的外部接口的语义。
- 本章的跨语言调用图构建工具实现为一个 VSCode 的语言服务扩展，在 10 个多语言应用上的实验表明，Frog 能够为基于不同宿主语言的外部接口 Python/C API 和 Node.js C++ addons，以及接口生成工具 pybind11 的多语言软件构建跨语言的调用图。同时，完整的调用图构建能够与其他的单语言 IDE 工具一起高效工作。

5.2　研究动机

本节从人工分析 PyTorch 中的一个跨语言调用链实例出发，说明跨语言调用图构建对于多语言软件的程序分析的重要性，以及现有工具的不足。

5.2.1　PyTorch 编程接口

PyTorch[14] 是流行的机器学习框架。数据分析程序通过在 Python 程序中使用 `torch` 包来调用 PyTorch 编程接口，其底层的计算会被分发到用 C++ 实现的内核函数。PyTorch 采用了典型的 Python/C 多语言软件架构，`torch` 包中的 Python 外部函数存在两种声明方式：在跨语言接口代码中直接调用 Python/C API，以及使用 pybind11 编程接口的跨语言接口生成。

在 2021 年 11 月，PyTorch 编程接口 `torch.kthvalue` 被发现存在漏

洞[230]。

```
torch.kthvalue(input, k, dim=None,
                    keepdim=False, *, out=None)
```

应当返回一个命名的元组 (values, indices)，其中 values 和 indices 分别是输入的 dim 维张量的每一行中第 k 小的元素的值和索引。torch.kthvalue 被发现当 k 超出维度的长度范围时，其不会抛出异常而是会返回一个随机值。v1.10.0 及之前版本的 PyTorch 都受到该漏洞的影响。然而，torch.kthvalue 被广泛使用于此前的深度学习项目中[231-233]。在这些项目中，k 的值都是动态计算的，存在触发该漏洞的风险。在该漏洞被发现之前，如果这些项目想要调试该 PyTorch 编程接口出现的非预期的随机性，检查外部函数 torch.kthvalue 的外部实现将会有很大帮助。

5.2.2　Python/C 调用链

考虑 PyTorch 的多语言软件架构，torch.kthvalue 在 Yang 等人提出的深度学习框架[232]中的跨语言调用链如下：

$$\texttt{pgd\_attack\_smooth} \rightarrow \texttt{kthvalue}^{\texttt{py}} \rightarrow \texttt{THPVariable\_kthvalue} \rightarrow \texttt{kthvalue}^{\texttt{c}} \rightarrow \texttt{kthvalue\_out} \tag{py/c-1}$$

$\texttt{kthvalue}^{\texttt{py}}$ 是 torch 包中宿主 Python 侧的外部函数，即 torch.kthvalue。$\texttt{kthvalue}^{\texttt{c}}$ 是外部函数对应的 C++ 底层实现。在 PyTorch 中两者名字相同，但这种同名关系不是 Python/C 互操作所必须的。THPVariable_kthvalue 是跨语言接口层的包裹函数，对应第 4 章跨语言程序的抽象语法（图 4.3）中的外部函数定义。包裹函数是使用外部接口编程的 C/C++ 程序，它进行参数解析、类型转换等跨语言的处理，并把实际的计算分发给 C/C++ 底层实现。C/C++ 底层实现一般是计算密集型的函数或库调用，对于简单的应用，底层实现不是必须的，计算逻辑也可以全部实现在包裹函数中。

跨越语言边界的跳转是调试和优化多语言软件的基础需求。然而，已有的工具缺乏足够的能力进行跨语言的跳转。Pylance[234]是 VSCode 默认的 Python 扩展，它基于微软公司的 Python 静态类型检查工具 Pyright[191]和 VSCode 的 Python 基础语言扩展 MS-Python[235]。对于外部函数调用，Pylance 会跳转到类型存根 .pyi 文件中，类型存根被用来为 Python 编程提供类型提示[158]。PyTorch 会在构建 torch 包的过程中生成类型存根文件。Pylance 视角下的调用链 (py/c-1) 为：

$$\texttt{pgd\_attack\_smooth} \rightarrow \texttt{kthvalue}^{\texttt{py}} \rightarrow \texttt{kthvalue}^{\texttt{pyi}} \tag{py/c-2}$$

其中 $\texttt{kthvalue}^{\texttt{pyi}}$ 是类型提示语法形式的 $\texttt{kthvalue}^{\texttt{py}}$：

```
def kthvalue(input: Tensor, k: _int, ...)
    -> namedtuple_values_indices: ...
```

它标注函数的参数和返回值的类型，并忽略函数体。类型提示函数必须使用和被标记的函数相同的函数名。类型存根只能辅助包含外部函数的宿主语言程序的编程开发，而不能帮助检查跨语言接口层和外部语言程序的实现和行为。已有的研究[38,80]和本文第 3 章已经说明跨语言程序是易错的，同时外部语言程序可能是性能敏感的[47]。完整的跨语言调用图对于错误定位、性能调试等程序分析任务有很大的帮助。

此外，对于大多数并不会生成类型存根的 Python/C 多语言软件，Pylance 将无法对外部函数做任何处理，调用链会终止在纯 Python 侧：

$$\texttt{pgd\_attack\_smooth} \rightarrow \texttt{kthvalue}^{\texttt{py}} \tag{py/c-3}$$

一些其他工具如 ffi-navigator[229] 提供比 Pylance 更好的效果，但是它们使用基于正则的模式匹配，要求熟悉对应 Python/C 多语言包的专家在程序分析前定制匹配规则，这些方法往往可扩展性和有效性较差，本章将在实验评估中对其进行对比分析。

5.2.3　JavaScript/C 调用链

多语言的软件架构不局限于 Python/C 互操作。优化的 C/C++ 库如基础线性代数子程序（Basic Linear Algebra Subroutines，BLAS）作为外部库被不同应用广泛地复用，在不同的宿主语言的程序中都提供了调用这些库的外部编程接口。为了把一个已有的 C/C++ 库迁移到某一宿主语言中作为外部扩展模块，例如实现一个 $\texttt{kthvalue}^{\texttt{js}}$ 并复用底层的 $\texttt{kthvalue}^{\texttt{c}}$，开发者需要编写新的跨语言接口代码，其中使用 JavaScript 的 C 外部接口替代 Python/C API，即实现一个新的包裹函数 `THPVariable_kthvalue`。然而，形如

$$\begin{aligned}&\texttt{pgd\_attack\_smooth} \rightarrow \texttt{kthvalue}^{\texttt{js}} \rightarrow \\ &\texttt{THPVariable\_kthvalue} \rightarrow \texttt{kthvalue}^{\texttt{c}} \rightarrow \texttt{kthvalue\_out}\end{aligned} \tag{js/c-1}$$

的调用链在 JavaScript 分析工具中也无法得到。VSCode 针对 JavaScript 和 TypeScript 的官方扩展[236]提供的调用关系如下：

$$\texttt{pgd\_attack\_smooth} \rightarrow \texttt{kthvalue}^{\texttt{js}} \rightarrow \texttt{kthvalue}^{\texttt{.d.ts}} \tag{js/c-2}$$

与 Python 中的 .pyi 文件类似，.d.ts 文件为 JavaScript 或 C/C++ 实现的编程接口提供基于 TypeScript 语法的类型信息。相似地，这些信息只能用作类型标注，无法辅助错误定位、性能优化等任务。

针对本节分析的跨语言调用图构建这一基础程序分析能力的不足，本章将提供一种不同宿主语言跳转进入外部 C/C++ 侧的解决方案。

5.3　外部映射构造

对比人工构造的理想的调用链 (py/c-1) 和 (js/c-1)，来自主流工具的调用链 (py/c-2)、(py/c-3)、(js/c-2) 都停止在真正进入外部侧之前。构建外部函数映射是构造跨语言调用图的核心。本节介绍跨语言调用图分析的阶段一，构造不同多语言软件的外部函数映射总结。

5.3.1　Python/C 映射

为了标记跨越语言边界的跨语言调用，调用链 (py/c-1) 可以重写为：

$$\texttt{pgd\_attack\_smooth} \rightarrow \texttt{kthvalue}^{\texttt{py}}{}_{\texttt{p}}\!\rightarrow_{\texttt{c}} \qquad \text{(py/c-4)}$$

$$\texttt{THPVariable\_kthvalue} \rightarrow \texttt{kthvalue}^{\texttt{c}} \rightarrow \texttt{kthvalue\_out}$$

其中跨语言调用箭头 $_{\text{p}}\!\rightarrow_{\text{c}}$ 把调用链分成了宿主 Python 侧和外部 C/C++ 侧。在大多数宿主语言中，C 外部接口都实现为 C 头文件，这使得跨语言接口层的包裹函数始终是 $_{\text{p}}\!\rightarrow_{\text{c}}$ 右侧的第一个函数。在 PyTorch 中，跨语言调用 $\texttt{kthvalue}^{\texttt{py}}{}_{\texttt{p}}\!\rightarrow_{\texttt{c}}\texttt{THPVariable\_kthvalue}$ 通过以下 `PyMethodDef` 类型的结构体声明：

```
{"kthvalue", cast(THPVariable_kthvalue),
 METH_VARARGS | METH_KEYWORDS, NULL}
```

Python/C API 定义的结构体类型 `PyMethodDef` 被用来声明一个外部函数，其包含如表 5.1 所示的四个域。`cast` 将外部函数 C/C++ 实现转换为 `PyCFunction` 类型，该类型是多数 Python 可调用对象（callable）对应的 C 实现。每个 `PyCFunction` 函数接收两个 `PyObject *` 类型的参数并返回一个 `PyObject *` 类型的值。第一个参数一般称为 `self`，其代表该外部函数所属的模块或类。第二个参数一般称为 `args`，是一个打包了所有宿主 Python 侧传入外部函数的位置参数的元组对象。常见的标记位 `METH_VARARGS` 是和 `PyCFunction` 类型对应的调用惯例，而 `METH_KEYWORDS` 则指明该外部函数实现可以接收第三个被称作 `kwargs` 的参数，它是一个保存传入外部函数的关键字参数的字典对象。

表 5.1　`PyMethodDef` 结构体的四个域

域	C 类型	含义
ml_name	const char *	外部函数的函数名
ml_meth	PyCFunction	指向外部函数 C/C++ 实现的指针
ml_flags	int	指明调用构造方法的标记位
ml_doc	const char *	文档字符串的内容

如前所述，`torch.kthvalue` 接收两个位置参数（`input`、`k`）和三个关

键字参数（dim、keepdim、out）。在 Python 语法中，参数 * 是一个分隔符，表示后续的参数只能是关键字参数[237]。图 5.1 展示了沿着 torch.kthvalue 调用图的跨语言参数传递和转换。

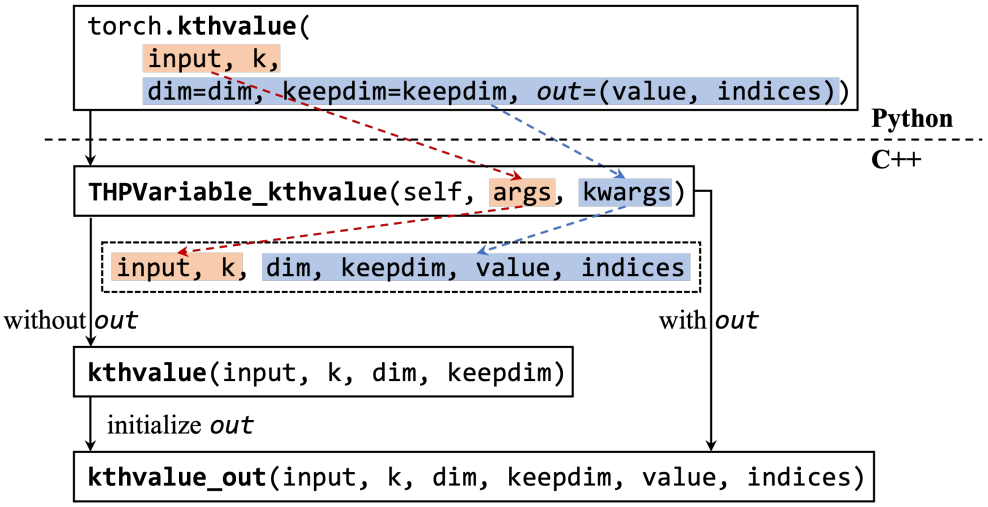


图 5.1　沿着 torch.kthvalue 调用图的跨语言参数传递和转换

Python 外部函数 torch.kthvalue 的位置参数和关键字参数分别被打包进入两个 PyObject * 类型的参数 args 和 kwargs，然后传给跨语言接口层的包裹函数 THPVariable_kthvalue。在包裹函数中，不同族的 Python/C API 被调用并依次完成以下功能：（1）把 Python 侧的传入的参数从 args 和 kwargs 中解包出来，（2）把 Python 参数转换为 C 类型的变量，（3）根据参数调用不同的 C/C++ 底层实现。图 5.1 中核心的调用关系是跨越语言边界的

$$\texttt{kthvalue}^{\texttt{py}}{}_{\texttt{p}\rightarrow\texttt{c}}\texttt{THPVariable\_kthvalue} \qquad \text{(py/c-5)}$$

$_{\texttt{p}\rightarrow\texttt{c}}$ 的左侧是使用 PyMethodDef 的 ml_name 域命名的 Python 外部函数，右侧是 ml_meth 域指向的 C/C++ 包裹函数。

5.3.2　不同外部映射的分析

Python/C API 中的结构体 PyMethodDef 并不是声明外部函数、构建 Python/C 互操作的映射关系的唯一方式。本小节进一步分析不同的跨语言接口声明方式，以及不同的宿主语言中的外部映射特性。

1. Python/C 接口生成

使用 Python/C API 编写安全可靠的 Python/C 跨语言接口代码并不容易（参考第 3 章以及 Li 和 Tan[33]、Mao 等人[34]的工作）。除了直接使用 PyMethodDef 声明一个 Python 的 C 外部函数，一些第三方工具还提供了接口生成的编程接口。这些工具的设计目标是减少人工编写扩展模块过程中的样板代码，减少跨语言程序的编码量和错误风险，并且更好地支持一些 C++ 特性如 lambda 函数。例

如，PyTorch 和 TensorFlow 都使用 pybind11[212]处理一部分的外部函数声明。在 pybind11 中，只需调用 `def` 函数就能简明地声明一个如跨语言调用关系 (py/c-5) 所示的外部映射：

```
def("kthvalue", &THPVariable_kthvalue)
```

其中第二个参数可以是普通 C++ 函数、函数指针或 lambda 函数。一方面，pybind11 省略了 `PyMethodDef` 结构体需要的函数类型转换 `cast` 和标记位 `METH_VARARGS|METH_KEYWORDS`，因为这些信息实际上可以从外部函数实现中推理得到。这种冗余来自 Python/C API 的设计，并且可能导致一些不一致的漏洞（如前文第 4.6.2 小节所述的类型错误）。另一方面，pybind11 可以容易地使用 C++ lambda 函数作为外部实现。这是 PyTorch 等多语言软件中一种常见的使用方式。一些其他接口生成工具如 SWIG[121]的 Python/C 映射关系的声明方式与 pybind11 相类似。

2. JavaScript/C 映射

外部接口是大多数主流编程语言的语言标准的一部分。这些外部接口支持宿主语言与外部 C/C++ 互操作，包括调用操作系统的编程接口，以及复用许多已有的优化库。例如，Node.js C++ addons[207]是 JavaScript 和 TypeScript 运行时环境提供的 C/C++ 外部接口。不同于 Python/C API 使用特定的结构体 `PyMethodDef` 声明外部函数，Node.js C++ addons 通过调用内联函数 `NODE_SET_METHOD` 声明外部函数。如下是来自 JavaScript/C 多语言软件 nBLAS[238]的一个例子，nBLAS 是 C 实现的单双精度 BLAS 例程的 Node.js 绑定。

```
NODE_SET_METHOD(exports, "ddot", ddot);
```

其中 `exports` 是存储当前模块中所有外部函数声明的容器。`"ddot"` 是 JavaScript 侧的外部函数调用名，`ddot` 是其对应的 C 外部包裹函数。包裹函数 `ddot` 处理参数传递和类型转换，然后把计算分发给库函数，即 BLAS 例程 `cblas_ddot`，它是双精度标量乘的 C 实现。

在 nBLAS 中，`ddot` 的底层调用链可以写作：

$$\texttt{ddot}^{\texttt{js}} \rightarrow \texttt{ddot} \rightarrow \texttt{cblas\_ddot} \tag{js/c-3}$$

其中 $\texttt{ddot}^{\texttt{js}}$ 是名为 `"ddot"` 的 JavaScript 外部函数，`ddot` 是外部函数实现（包裹函数）。这里的 JavaScript/C 映射关系可以表示为：

$$\texttt{ddot}^{\texttt{js}}{}_{\texttt{j}} \rightarrow_{\texttt{c}} \texttt{ddot} \tag{js/c-4}$$

在 PyTorch 中，外部函数（如 $\texttt{kthvalue}^{\texttt{py}}$）和库函数（如 $\texttt{kthvalue}^{\texttt{c}}$）同名，而包裹函数（如 `THPVariable_kthvalue`）有特定的前缀（如 `THPVariable_`）。在 nBLAS 中，外部函数和包裹函数同名，库函数则有特定的前缀 `cblas_`。这些命名方式都不是必须的，不同的多语言软件可能有不同

的命名。这解释了基于词法的工具需要熟悉分析目标的专家辅助定制提取规则的原因，也导致这些工具不能有效且自动地扩展到其他多语言软件。

5.3.3　外部映射语义模型

基于对不同多语言互操作中外部映射特性的分析，本节提出一种泛化的 X/C 映射，并构建宿主语言无关的外部映射语义模型。

1. X/C 映射

分别如 (py/c-5) 和 (js/c-4) 所示的 Python/C 和 JavaScript/C 中外部映射的跨语言调用关系可以进一步泛化为：

$$\text{ffname}\,{}_{\text{h}}{\rightarrow}_{\text{c}}\,\text{ffimpl} \qquad \text{(h/c)}$$

其中 ffname 是宿主侧的外部函数调用名，ffimpl 是对应的 C/C++ 包裹函数实现，其可以是普通函数（或方法）、函数指针或 lambda 函数。包裹函数是外部侧的入口，它调用 C 外部接口处理跨语言的参数传递和类型转换，并把计算进一步分发到底层的 C/C++ 库函数。对于外部函数功能相对简单的多语言软件，进一步的计算分发不是必须的，外部函数实现可以仅在包裹函数中完成。

2. 语义模型

外部映射语义模型（Foreign Mapping Semantics Model，FMSM）从不同的多语言软件中提取 X/C 映射，其可以形式化地表示为如下的范型函数：

$$\mathcal{M} : \{(\mathit{Kind}, \mathit{Pattern})\} \rightarrow \text{(h/c)} \qquad \text{(FMSM)}$$

其中 $\mathit{Kind} ::= \text{struct} \mid \text{func}$ 是进行外部函数声明的外部接口的类型。例如，Python/C 多语言程序的 C 外部接口 Python/C API 中的 `PyMethodDef` 是一个结构体，接口生成工具 pybind11 中的 `def*` 编程接口是函数，JavaScript/C 多语言程序的 C 外部接口 Node.js C++ addons 中的 `NODE_SET_METHOD` 是内联函数。$\mathit{Pattern} ::= \mathit{apiname} < \cdots >$ 描述不同外部函数接口的提取模式。基于外部映射语义模型 $\mathcal{M}$，大多数不同宿主语言的 C 外部接口和不同接口生成工具的编程接口都可以归约到外部映射关系 (h/c)。Frog 支持的编程接口如表 5.2 所示，并且可以容易地扩展支持更多编程接口。

3. 模式抽象

考虑不同多语言互操作中不同的外部映射特性，外部函数声明的编程接口可能是结构体 $\{\texttt{field}_i\}$ 或函数调用 $(\texttt{parameter}_i)$。为了统一，表 5.2 使用了 $< \cdots >$ 形式的模式描述来指明如何得到基础的 (h/c) 关系，其中占位符代表一个在外部映射语义模型中不关心的域或参数位置。`*_static` 声明静态外部函数。`def_property*` 把外部 get/set 方法绑定到一个属性变量，外部 get/set 方法对应的 C/C++ 外部实现分别为第二和第三个参数位置指明的函数：

```
def_property("var", getVar, setVar, extra)
```

当属性 `var` 作为赋值语句的右手侧（Right Hand Side，RHS）出现时，其外部 get 方法 `getVar` 被调用。当它作为左手侧（Left Hand Side，LHS）出现时，其外部 set 方法被调用。对于这类外部函数声明，Frog 会同时建立两条外部映射关系。如果属性是只读的，那么只有其外部 get 方法可以被调用。

表 5.2　外部函数声明的 C 外部接口和生成器编程接口及其模式抽象

编程接口	模式抽象
Python/C API	
`PyMethodDef`	<ffname, ffimpl, _, _>
pybind11	
`def`	<ffname, ffimpl, _>
`def_static`	<ffname, ffimpl, _>
`def_property`	<ffname, ffimpl, _, _> (get) <ffname, _, ffimpl, _> (set)
`def_property_static`	<ffname, ffimpl, _, _> (get) <ffname, _, ffimpl, _> (set)
`def_property_readonly`	<ffname, ffimpl, _, _> (get)
`def_property_readonly_static`	<ffname, ffimpl, _, _> (get)
Node.js C++ addons	
`NODE_SET_METHOD`	<_, ffname, ffimpl>

5.3.4　外部映射分析

外部映射构造是跨语言调用图构建系统 Frog 的第一个分析阶段，从多语言软件代码库 $\mathcal{B}$ 中提取外部映射总结 $\mathcal{S}$。每一条外部映射总结形如

(ffname, ffimpl, file, line)

其中 ffname 和 ffimpl 记录外部映射关系 (h/c)，file 和 line 记录外部函数声明的位置。

外部函数总结由图 5.2 所示的系统部件分析得到。多语言代码库 $\mathcal{B}$ 由宿主侧代码 $files_h$ 和外部侧 C/C++ 代码组成。基于静态的可能包含（may-include）分析，接口分离器把外部 C/C++ 代码分成两部分，跨语言接口代码 $files_i$ 和外部库代码 $files_c$。如果一个文件包含或间接包含了定义 C 外部接口（CFI）或生成器编程接口（GI）的头文件，那么则认为该文件是一个跨语言接口文件。例如，CFI Python/C API 定义在 Python.h 及其包含的一些头文件中，CFI Node.js C++ addons 定义在 node.h 及其包含的一些头文件中，GI pybind11 的编程接口定义在 pybind.h 及其包含的一些头文件中。静态的可能包含分析可以找到所有可能的跨语言接口文件，其可靠但不精确。接口分离器可以同时得到该多语言软件使用的 CFI 和/或

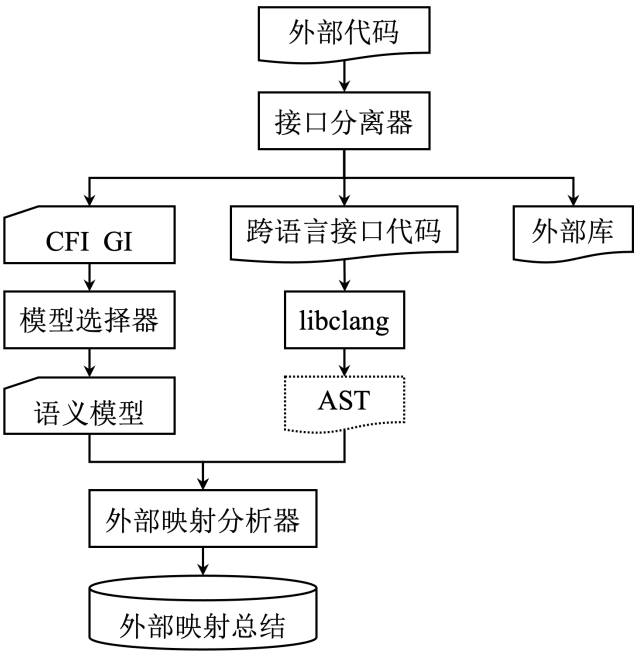


图 5.2　Frog 阶段一：外部映射构造

GI，它假设多语言软件中只有一种宿主语言，但是 CFI 和 GI 可以同时使用并分别声明一部分外部函数。例如，PyTorch 的源码分成宿主 Python 程序和外部 C++ 程序，接口生成器识别 PyTorch 同时使用 Python/C API 和 pybind11 来完成 C++ 外部实现和 Python 外部函数的绑定。

算法 5.1　外部映射构造

输入: 分离后的多语言代码库 $\mathcal{B} = (files_h, files_i, files_c)$
输出: 外部映射总结 $\mathcal{S} = \{(\text{ffname, ffimpl, file, line})\}$
$\mathcal{S} \leftarrow \{\}$
for $\forall f \in files_i$ **do**
　$FI \leftarrow getFI(f)$
　switch *FI* **do**
　　case *Python/C API* **do**
　　　$m \leftarrow \{(\text{struct, } \texttt{PyMethodDef} \text{ <ffname, ffimpl, \_, \_>})\}$
　　end
　　case *pybind11* **do**
　　　$m \leftarrow$
　　　{(func, `def` <ffname, ffimpl, _>),
　　　(func, `def_static` <ffname, ffimpl, _>), ⋯ }
　　end
　　case *Node.js C++ addons* **do**
　　　$m \leftarrow \{(\text{func, } \texttt{NODE\_SET\_METHOD} \text{ <\_, ffname, ffimpl>})\}$
　　end
　end
　// $\bar{\mathcal{M}}_m$ 是由特定的外部映射提取规则 $\mathcal{M}|_m$ 生成的 `AST` 访问者，其中 $\mathcal{M}|_m : m \to (h/c)$，$\mathcal{M}|_m \subseteq \mathcal{M}$。
　$\mathcal{S} \leftarrow \mathcal{S} \cup \bar{\mathcal{M}}_m(AST(f))$
end

以分离后的多语言代码库 $\mathcal{B} = (files_h, files_i, files_c)$ 为输入的外部映射构建过

程如算法 5.1 所示。接口分离的过程基于前述的包含关系，并同时记录每个跨语言接口文件使用的外部接口（*FI*）。基于外部接口对应的 CFI 和 GI 信息，模型选择器根据该多语言项目的外部接口选择特化的语义模型，包括外部函数声明的外部接口的类型（*Kind*）和模式（*Pattern*），其指明具体的外部函数声明语义对应的提取规则。即 CFI 和 GI 的信息被用于确定 AST 遍历过程中的外部映射提取规则 m，完成外部映射语义模型 (FMSM) 即范型函数 $\mathcal{M}$ 的参数化。$\mathcal{M}|_m : m \to (h/c)$，$\mathcal{M}|_m \subseteq \mathcal{M}$，$\mathcal{M}|_m$ 是 $\mathcal{M}$ 在特定跨语言互操作方法下的投影。特定的 AST 访问者 $\bar{\mathcal{M}}_m$（外部映射分析器）根据具体的提取规则 $\mathcal{M}|_m$ 中外部函数声明的外部接口的类型和模式搜索外部映射关系 (h/c)。AST 基于 libclang[165] 生成，其中宏已经展开。算法 5.1 对跨语言接口代码的 AST 节点做一次遍历，寻找特定类型的节点构造外部映射关系，其时间复杂度为 $O(n)$，其中 n 是跨语言接口代码的 AST 节点数。

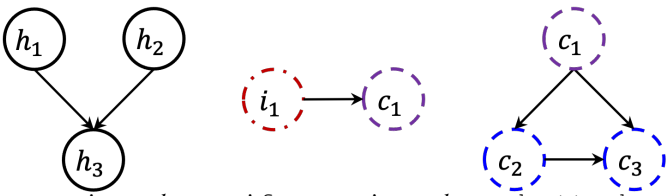
5.4　调用图构造

外部映射 (h/c) 描述了跨越语言边界的核心调用关系，这一关系可以视作完整跨语言调用图的子图。本节介绍使用图变换和函数节点融合构造完整跨语言调用图的分析阶段二。

5.4.1　调用子图

1. 宿主侧和外部侧的单语言调用图

对于不同语言的单语言程序，往往已有相对成熟的调用图构造工具。仍然以 Yang 等人[232]基于 PyTorch 的深度学习网络为例，调用链 (py/c-1) 中 `pgd_attack_smooth` 是一个进行白盒攻击的函数，攻击者能够获取模型的梯度、拥有模型权重的拷贝。该函数同时被两个文件的 `main` 函数 `train.__main__` 和 `verify.__main__` 调用。



h_1: `train.__main__`, h_2: `verify.__main__`, h_3: `pgd_attack_smooth`
i_1: $\texttt{kthvalue}^{\text{py}}$
c_1: `THPVariable_kthvalue`, c_2: $\texttt{kthvalue}^{\text{c}}$, c_3: `kthvalue_out`

图 5.3　宿主侧、跨语言接口层、外部侧的调用子图示例

图形化地，Python 和 C/C++ 侧的调用子图分别如图 5.3 的左右两部分所示，其中标记 h_* 的实线圆圈代表宿主侧的函数节点，标记 c_* 的虚线圆圈代表外部侧的函数节点。

形式化地，宿主侧的单语言调用图可以表示为 $CG^h = (V^h, E^h)$，其中 V^h 是宿主侧函数节点的有限集合，即图 5.3 中所有 h_* 节点组成的集合，$E^h \subseteq V^h \times V^h$ 是宿主侧函数节点间的有向边组成的集合。相似地，在外部 C/C++ 侧，调用子图 $CG^c = (V^c, E^c)$ 包含函数节点的有限集 V^c 和调用边集合 $E^c \subseteq V^c \times V^c$。

2. 跨语言接口层调用图

图 5.3 中间部分所示是跨语言接口层的调用子图，对应于第 5.3 节中介绍的外部映射关系。

不同于单语言程序中的函数在同一语言域中声明、定义、调用，外部函数在宿主语言侧被调用，但是在跨语言接口层和外部侧声明和定义（参见第 4 章中跨语言程序的抽象语法）。

形式化地，作为图的外部映射总结由若干相连的节点对组成，即 $CG^i = \{(v^i, v^c)\}$。$v^i \in V^i$ 代表外部函数节点，对应于图 5.3 中 i_* 标记的点划线圆圈节点。外部函数节点对于宿主语言和外部语言的单语言调用图工具都是不可见的。$v^c \in V^c$ 是外部函数实现的包裹函数，其本质上是一个 C/C++ 函数，可以被单语言调用图工具发现。

5.4.2　跨语言调用图

将完整的跨语言调用图定义为 $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$，$\mathcal{G}$ 是调用子图 CG^h、CG^i、CG^c 不带有额外节点的超图，即 $\mathcal{V} \subseteq V^h \cup V^i \cup V^c$。本节将介绍连接不同语言域的子图的图变换和函数节点融合。

1. 图变换（t）

宿主语言的外部函数虽然是一个宿主侧的函数对象，但是由于其声明和实现在跨语言接口层和外部侧，所以外部函数不能被单语言的分析工具检测到。宿主侧的调用子图 CG^h 不包含外部函数节点。例如，Python 本地函数 h_3（`pgd_attack_smooth`）调用定义在 Python/C 扩展模块 `torch` 包中的外部函数 i_1（`kthvalue`）。在 `torch` 中，(py/c-5) 是数百个外部映射之一。包括 `torch.kthvalue` 在内，`pgd_attack_smooth` 调用的所有 `torch` 编程接口在 CG^h 中都不可见。在使用 PyTorch 的应用程序如 Yang 等人提出的 rs4a[232]中，其调用的 `torch` 外部函数构成子集 $V^i_{\text{rs4a}} \subseteq V^i$，该子集的大小远小于外部映射总结中存储的外部函数总数。

给定宿主侧的调用子图 CG^h，以及保存在外部映射总结对应的跨语言接口层调用子图 $CG^i = \{(v^i, v^c)\}$ 中的外部函数名集合 V^i，图变换 (t) 在本地函数的词法标记 (token) 中搜索外部函数调用，并把调用子图 CG^h 变换到 CG^{hi}（见图 5.4）。算法 5.2 所示是图变换（t）的分析过程，其时间复杂度为 $O(|V^h| \cdot nt_h)$，其中 nt_h 是宿主语言程序中函数的词法标记数。

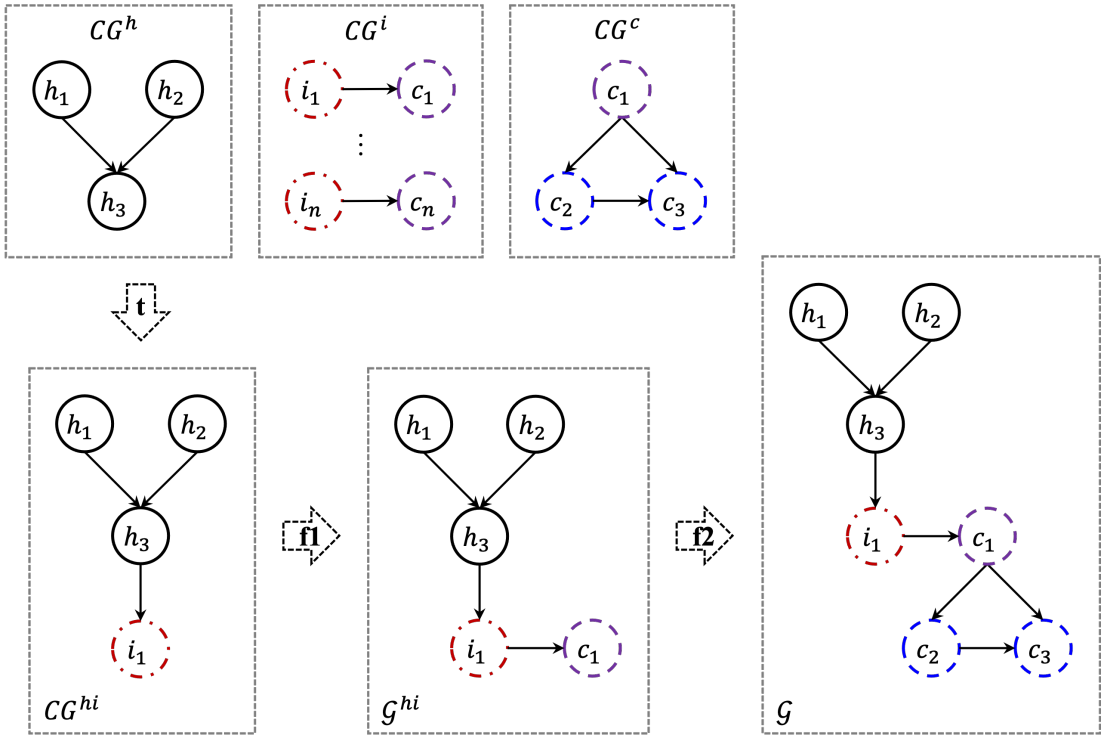


图 5.4　图变换（t）和函数融合（f1，f2）以构造跨语言调用图

算法 5.2　图变换（t）搜索外部函数调用

输入: $CG^h = (V^h, E^h), V^i$
输出: CG^{hi}
$V^i_{\text{rs4a}} \leftarrow \{\}$
$E^{hi} \leftarrow E^h$
for $v^h \in V^h$ **do**
　$T \leftarrow tokenize(v^h)$
　for $t \in T$ **do**
　　if $t = v^i \in V^i$ **then**
　　$V^i_{\text{rs4a}} \leftarrow V^i_{\text{rs4a}} \cup \{v^i\}$
　　$E^{hi} \leftarrow E^{hi} \cup \{(v^h, v^i)\}$
　　end
　end
end
$CG^{hi} = (V^h \cup V^i_{\text{rs4a}}, E^{hi})$

2. 外部函数节点融合（f1）

图变换之后，如图 5.4 所示，在 CG^{hi} 和 CG^i 中分别可以看到两个相同的函数节点 i_1。在表示为 CG^i 的外部映射关系 (h/c) 中，ffname 存储的是外部函数在宿主侧被调用的函数名字符串，而不是其对应的函数对象本身。算法 5.2 使用词法标记化而不是 AST 访问等更精确的分析。词法标记化简单快速，并且不区分 `torch.kthvalue` 和 `t.kthvalue`，其中 `t` 是一个张量对象。这种不精确反而弥补了 Pylance[234] 中常见的一种调用关系的召回损失，因为 `t` 的类型难以静态推断，尤其对于动态类型的宿主语言。对于 `t.kthvalue` 这类未知类实例类型的外部方法调用，Pylance 会从 (py/c-2) 退化到 (py/c-3)，即使外部方法本身

的类型存根已知。本章在跨语言调用图构建过程中希望可以避免这种召回损失。一个合理的假设是 `t.kthvalue` 和 `torch.kthvalue` 在外部实现中有相同或者至少相似的调用路径。对于 `torch.kthvalue` 和 `t.kthvalue` 的定义跳转，即使两者对应不同的 C++ 外部函数实现，作为 IDE 插件的 Frog 可以同时提供两者作为备选。这种处理提高了外部调用关系的召回精度但是可能降低准确性。如果 `torch.kthvalue` 和 `t.kthvalue` 共享相同的外部实现，那么 Frog 的分析是准确的；如果两者分别有各自的外部实现，Frog 对两者都提供两个可能的跳转位置。

算法 5.3　函数融合 $\mathcal{G}^{hi} = \mathbf{f1}(CG^{hi}, CG^{i})$

输入: $CG^{hi} = \{V^{hi}, E^{hi}\}, CG^{i}$
输出: $\mathcal{G}^{hi}$
$\mathcal{G}^{hi} \leftarrow CG^{hi}$
for $v^{i} \in V^{hi}$ **do**
　$\mathcal{G}^{hi} \leftarrow \mathcal{G}^{hi} \cup \{(v^{i}, v^{c})\}$, where $(v^{i}, v^{c}) \in CG^{i}$
end

算法 5.3 融合变换后的宿主侧调用子图 CG^{hi} 和跨语言接口层调用子图 CG^{i} 得到 $\mathcal{G}^{hi}$。外部函数节点融合本质上是一个图上的加法，其时间复杂度为 $O(|V^{hi}|)$。

3. C/C++ 侧函数节点融合（f2）

函数融合 f1 处理宿主侧和跨语言接口层之间外部函数调用名关联外部函数实现的问题。函数融合 f2 处理跨语言接口层和 C/C++ 侧之间的外部函数实现定位问题。分析阶段一（第 5.3 节）的分析结果预置为外部映射总结，每一条外部映射记录通过文件路径和起始行号保存了外部函数实现的定义位置信息。然而，分析阶段一是离线的，即外部映射总结一般来自多语言软件在特定版本和平台上的某个构建。例如，包裹函数 `THPVariable_kthvalue` 的定义位置是和 PyTorch 版本以及构建 PyTorch 的机器相关的。

算法 5.4　函数融合 $\mathcal{G} = \mathbf{f2}(\mathcal{G}^{hi}, CG^{c})$

输入: $\mathcal{G}^{hi}, CG^{c}, \mathcal{L} = \{v^{c} : (file, line)\}$ for each v^{c} in $\mathcal{G}^{hi}$
输出: $\mathcal{G}, \mathcal{L}$
$\mathcal{G} \leftarrow \mathcal{G}^{hi}$
for $(v^{i}, v^{c}) \in \mathcal{G}^{hi}$ **do**
　while $(v^{c}, v^{cn}) \in CG^{c}$ **do**
　　// 直到 v^{cn} 不存在后继
　　if $(v^{c}, v^{cn}) \notin \mathcal{G}$ **then**
　　　$\mathcal{G} \leftarrow \mathcal{G} \cup \{(v^{c}, v^{cn})\}$
　　　update($\mathcal{L}[v^{c}]$)
　　　$v^{c} \leftarrow v^{cn}$
　　else
　　　// 打破递归调用循环
　　　break
　　end
　end
end

函数融合 f2（算法 5.4）连接 $\mathcal{G}^{hi}$ 和 CG^c 以构建完整的跨语言调用图。对于每个外部函数实现，即外部侧的入口（包裹函数），算法 5.4 搜索它调用的库函数，并把 C/C++ 侧的子图与其连接。*update* 是一个 AST 访问过程，其根据分析阶段二所处的实际的多语言软件构建环境更新外部函数实现的定义位置。算法 5.4 的时间复杂度为 $O(|V^c| \cdot na_c)$，其中 na_c 是外部语言程序中函数的 AST 节点数目。

5.5　评估

本章提出的跨语言调用图构建分析 Frog 被实现为一个 VSCode 语言服务扩展。根据 JetBrains 公司 2022 年度的 C 语言开发者生态调查[239]，微软公司的 VSCode 是 C 程序开发使用最多的 IDE 或编辑器。基于其语言服务扩展编程接口[240]，VSCode 允许自定义分析程序提供 IDE 的动态特性，如代码格式化、自动补全、定义跳转、错误检查等。作为 IDE 服务的同时，Frog 的核心组件如外部映射构造（分析阶段一，第 5.3 节）和函数节点融合 f2 的函数定位（分析阶段二的一部分）可以作为独立的工具工作。本节通过回答以下研究问题（Research Question，RQ）评估本章提出的跨语言调用图构建方法。

RQ1　在多语言软件的跨语言调用图构建中，Frog 定位外部函数的外部实现的准确度和召回率如何?

RQ2　与针对特定 Python/C 多语言仓库定制的工具 ffi-navigator[229] 相比，Frog 外部映射构建的召回率如何?

RQ3　跨语言调用图构建的分析速度如何？通过离线计算分析阶段一的外部映射总结能够带来多少速度提升？以及考虑实际开发环境，如何提升分析阶段二的算法的时间效率?

5.5.1　实验设置

为了回答上述研究问题，本节从 GitHub 选择了 10 个带有 C/C++ 扩展模块的 Python 和 Node.js 应用作为测试集。如表 5.3 所示，该测试集覆盖了不同应用领域。RQ1 将分析结果与针对测试集人工构造的验证集对比。该验证集包含测试集中除 PyTorch 外的多语言软件中的所有外部函数声明，对于 PyTorch 则仅验证它的一个核心编程接口子集。

Python 和 JavaScript 的调用图工具只能得到单语言调用图（图 5.4 中的 CG^h）。Bogar 等人[241] 提出的方法以及 Chen 等人（TVM 等深度学习编译框架[242-244] 的作者）设计的工具 ffi-navigator[229] 也以构建跨语言的调用关系为目标，但是它们基于特定多语言应用的语法，需要由熟悉该应用的专家定制提取规则。例如，ffi-navigator 只编码了 PyTorch、TVM[244]、MXNet[242] 等 5 个 Python/C 多语言软

表 5.3　召回率和准确性精度的实验结果

宿主语言	应用	描述	外部函数	召回	准确
Python	pytorch	机器学习	146*	145/146 (99.3%)	145/300 (48.3%)
	python-ldap	目录存取	28	28/28 (100%)	28/28 (100%)
	pyaudio	音频 I/O	28	28/28 (100%)	28/28 (100%)
	python-krbV	网络授权	42	42/42 (100%)	42/42 (100%)
	trace-cruncher	内核跟踪	82	82/82 (100%)	82/82 (100%)
	rabbit	输入模拟	18	18/18 (100%)	18/18 (100%)
JavaScript	nblas	科学计算	112	112/112 (100%)	112/112 (100%)
	node-multi-hashing	密码学	52	52/52 (100%)	52/52 (100%)
	FyneWav	音乐创作	10	10/10 (100%)	10/10 (100%)
	nanostat	文件管理	2	2/2 (100%)	2/2 (100%)
	rabbit	输入模拟	18	18/18 (100%)	18/18 (100%)
总计			538	537/538 (99.8%)	537/692 (77.6%)

件的分析规则。ffi-navigator 不能支持随机的多语言软件，包括其他的 Python/C 多语言软件，以及所有宿主语言非 Python 的多语言软件。Bogar 等人[241]提出的方法的实现过于老旧，无法在 Python 3 上运行，因此 RQ2 选择 ffi-navigator 与 Frog 进行对比。此外，ffi-navigator 也是一个语言服务扩展，并且没有提供能直接单独工作的组件。同时 ffi-navigator 仅在 IDE 中提供跳转到 Python/C 跨语言接口层的能力，即其仅构造外部函数映射（图 5.4 中的 CG^{hi}），而非完整的跨语言调用图。RQ2 使用 Frog 的分析阶段一得到的外部映射总结的外部函数调用名来触发 ffi-navigator 的定义跳转功能，对比检查 ffi-navigator 外部调用关系的召回损失。

Frog 实现为 VSCode 的语言服务，分析阶段二的部分模块可以通过 IDE 能力减少第 5.4 节中的算法的时间复杂度。RQ3 首先计算分析测试集的外部映射总结的耗时，然后计算在 IDE 中基于已计算得到的外部映射总结构造外部实现调用关系的耗时。

5.5.2　外部实现定位精度（RQ1）

外部实现定位的实验结果如表 5.3 所示。外部实现定位实验关注定位外部函数的外部实现的精度。表 5.3 第四列记录多语言应用中的外部函数数量，即经过人工检查的外部函数全集。对于 PyTorch，RQ1 仅检查其核心的 `torch.*` 外部编程接口，该外部编程接口子集由深度学习框架 MindSpore[245]选择支持的 PyTorch 编程接口组成。MindSpore 基于 PyTorch 1.5.0 选择了共 146 个编程接口，但是 RQ1 的实验在当时最新的 PyTorch 1.9.0 版本上进行。表 5.3 召回列所

示为 Frog 能够分析得到外部实现定位的外部函数比例。对于大多数 Python 和 JavaScript 外部函数，Frog 都成功给出了其跨语言调用的外部实现的位置。唯一遗漏的 `torch.Tensor` 实则是一个类对象而非函数对象。表 5.3 准确列计算外部实现定位和所有备选外部定位位置的比值。如函数节点融合算法 f1 所描述的，Frog 刻意地不区分同名的模块函数（`torch.x`）和类方法（`t.x`，其中 `t` 是一个张量对象）。这种处理提升了召回精度，但是降低了准确性精度，尤其是当多语言应用对同名外部函数使用不同外部实现时，PyTorch 中存在一些这样的例子。作为 IDE 的语言服务，提供多个备选的外部定位位置供开发者选择是可以接受的。对于错误定位和性能优化等任务，同名外部函数（方法）的外部实现一般面临相似的问题。

5.5.3　外部映射召回率（RQ2）

ffi-navigator[229] 为特定的 Python/C 多语言软件提供跳转定义语言服务特性。RQ2 使用 Frog 分析得到的 PyTorch 外部映射总结的外部函数调用名触发 ffi-navigator 的跳转定义功能，从而对比 ffi-navigator 外部映射关系的召回损失。表 5.4 记录了实验结果。ffi-navigator 依赖针对特定 Python/C 多语言软件的提取规则，需要熟悉被分析的目标应用的专家预先定制基于正则的规则。例如，ffi-navigator 会匹配包含表 5.2 中的编程接口 `PyMethodDef` 和 `def` 所在的代码行。然而，这种基于语法的分析是不精确的，一些 `PyMethodDef` 函数声明如 `torch._initExtension` 被遗漏了，同时它难以支持任意 Python/C 多语言程序中用到的其他外部函数声明方式如 `def_static`，更加难以支持其他宿主语言如 JavaScript。另一方面，ffi-navigator 需要指明搜索的路径，其在 `aten` 和 `torch` 路径之外的外部映射召回会定位到错误的外部位置，因为在其他路径下存在同名的外部函数。平均地，在 PyTorch 上 ffi-navigator 相比 Frog 遗漏了 35.9% 的外部映射关系。

表 5.4　外部映射召回对比

PyTorch 路径	Frog	ffi-navigator (损失%)
./aten/	2	2 (0%)
./torch/	2038	1843 (9.6%)
./caffe2/	81	9 (88.9%)
./third_party/	823	48 (94.2%)
./test/	41	12 (70.7%)
./benchmark/	2	0 (100%)
./build/	3	3 (0%)
总计	2990	1917 (35.9%)

5.5.4　分析时间（RQ3）

语言服务协议[246]（Language Server Protocol，LSP）作用于 IDE 客户端和程序分析服务之间，以将代码补全、跳转定义等特性集成到客户端中。通过使用基于标准 JSON-RPC 的进程通信，LSP 解耦了程序分析服务的目标语言和 IDE 客户端的实现语言。开发实践中常见的 IDE 大多支持 LSP。LSP 的架构也理想地适合本章提出的跨语言调用图分析，基于 LSP 可以简化第 5.4 节中的跨语言调用图构建算法。

图变换（t）中的词法标记化对于不同的宿主语言而言都快速简便，它可以一次性完成对一个文件中所有函数的分析，因此算法 5.2 中的外部循环实际上可以被展开。基于 LSP，内层循环也可以忽略。Frog 实现为一个语言服务扩展，其跳转定义是由指针位置给定的名字触发的。这一特性避免了匹配函数标记和外部映射总结中的外部函数名的搜索过程。两层循环展开后，Frog 实际对于算法 5.2 实现了近似常数时间复杂度。

函数融合算法 5.3（f1）和算法 5.4（f2）描述了多语言软件的完整的调用图构建。然而，实际在语言服务中的分析任务更关心外部函数的外部实现定位。定位外部函数的外部实现对于安全和性能调试等分析任务都更有帮助。基于实际使用行为，函数融合 f1 和 f2 两者可以合并在一起，同时 f2 的 `while` 循环可以展开为仅包含一次迭代。基于 LSP 和实际使用行为的函数融合算法（f1+f2）的时间复杂度可以优化为 $O(|V^{hi}| \cdot na_i)$，其中 na_i 是外部函数实现（包裹函数）的 AST 节点数，na_i 往往远小于整个 C 侧的函数的 AST 节点数 na_c。

表 5.5　分析时间的统计结果

应用	文件	外部函数	平均耗时 (ms/ff)	
			阶段一	阶段二
pytorch	Module.cpp	68	103.6	0.3
python-ldap	functions.c	5	19.2	0.1
pyaudio	_portaudiomodule.c	28	3.1	0.1
python-krbV	krb5module.c	42	3.0	0.1
trace-cruncher	ksharkpy.c	9	9.3	0.2
rabbit	rabbit_python.cpp	18	7.6	0.1
nblas	index.cc	112	4.3	0.1
node-multi-hashing	multihashing.cc	52	9.5	0.3
FyneWav	wrapper.cpp	10	55.5	0.4
nanostat	nanostat.cc	2	251.5	0.2
rabbit	rabbit_javascript.cpp	18	21.2	0.05
总计		364	27.4	0.2

这些基于 LSP 的现实开发环境的特性解释了表 5.5 中分析阶段二的平均用时（毫秒每外部函数，ms/ff）很短的原因。完成一个外部函数的外部实现定位平均只需 0.2 ms。在语言服务中对给定的外部函数触发跳转定位分析提高了分析阶

段二的时间效率。

分析阶段一比较耗时。分析得到一个外部函数的外部映射关系平均耗时 27.4 ms。分析阶段一作用于跨语言接口层的每个文件，根据多语言软件的架构组织的不同，跨语言接口层可能在一个大文件中包含许多需要分析的包裹函数，导致需要若干秒才能返回分析结果。然而，在 Frog 中分析阶段一被设计为离线的。一个多语言软件的外部映射总结可以提前缓存，从而极大地加速跨语言调用图的分析过程。

5.5.5　评估效果总结

本章提出的跨语言调用图构建工具 Frog 能够支持基于语言接口 Python/C API 和接口生成工具 pybind11 的 Python/C 多语言软件，以及基于语言接口 Node.js C++ addons 的 JavaScript/C 多语言软件。在 10 个不同领域的多语言软件组成的测试集上的实验表明，Frog 给出了 99.8% 的外部函数的外部实现位置。针对无法精确定位的同名外部函数，Frog 会同时给出不同位置作为备选，并取得了平均 77.6% 的外部实现定位精度。与仅支持特定 Python/C 多语言软件的工具 ffi-navigator 对比，Frog 在 PyTorch 上成功构建了 2990 个外部映射关系，而 ffi-navigator 丢失了其中的 35.9%。基于 VSCode 的语言服务评估分析测试集所话费的时间，实验表明 Frog 离线的阶段一分析一个外部函数平均耗时 27.4 ms，在线的阶段二分析一个外部函数平均耗时 0.2 ms，具有较为即时的时间效率。

5.6　讨论

5.6.1　外部对象

术语“外部函数接口”和“外部接口”可以相互替换，因为它们都不局限于函数[76]。本章关注调用图构建这一问题，其中图中的节点是函数，但是实际外部映射可以支持函数以外的其他对象。(h/c) 的语义抽象可以进一步泛化为：

$$\text{ffname}_{\text{h}} \rightarrow_{\text{c}} \text{ffobj}$$

其中 ffobj 包含外部函数、外部变量、外部类。通过抽象描述更多声明非函数外部对象的外部接口和生成器编程接口，该语义模型可以扩展支持更多外部对象。例如，考虑 Python 等高级语言的面向对象特性，Python/C API `PyTypeObject` 把一个 C 类型绑定到宿主侧的一个类，`PyMemberDef` 结构体描述类属性和外部结构体成员间的映射关系，pybind11 编程接口 `def_readonly*` 被用来声明外部模块或外部类的成员变量。

这种扩展对于多语言程序分析任务是有益的。第 4 章使用一种特化的外部

映射作为 Python 的 C 外部函数的类型推断的前提之一。基于外部对象的扩展可以设计外部对象的类型推断。此外，外部对象扩展还可以区分模块函数和类方法，从而提高跨语言调用图构建的准确性。

5.6.2　回调行为

在跨语言调用过程中，宿主语言调用外部函数，外部函数调用外部实现包裹函数，包裹函数将底层计算进一步分发到 C/C++ 库函数，这一单向的跨语言调用过程如 (py/c-4) 所示。本章已经说明了这种常见行为可以通过以下分析方法支持：（1）第 5.3 节介绍的外部映射语义模型，（2）单语言调用图构建工具或 IDE 能力，（3）第 5.4 节介绍的图变换和函数节点融合算法。

然而，还有一种对程序分析方法更具挑战的行为，即包裹函数中存在 C/C++ 程序反向调用宿主语言对象的情形。高级宿主语言调用低级外部语言的行为常常被称作外部调用（call-out），如果低级语言可以通过调用高级语言的函数来响应，则将该调用行为称为回调（call-back）。考虑回调时，跨语言调用过程不再仅单向地跨越语言边界一次，而是先由宿主侧通过外部调用进入外部侧，再由外部侧通过回调返回宿主侧。对于调用图构建分析任务，不妨假设被回调的宿主对象是一个可调用对象，包括在宿主侧声明和实现的本地函数和类方法。对于回调行为，可以使用一个额外的回调边和被调节点来扩展外部映射关系 (h/c)：

$$\text{ffname}_{\text{h}}\rightarrow_{\text{c}}\text{ffimpl}_{\text{c}}\rightarrow_{\text{h}}\text{hfimpl}$$

宿主函数 hfimpl 是宿主语言实现的函数或方法。这里仅考虑发生在外部调用上下文中的回调行为，即回调 $\text{ffimpl}_{\text{c}}\rightarrow_{\text{h}}\text{hfimpl}$ 始终绑定于一个外部函数调用 $\text{ffname}_{\text{h}}\rightarrow_{\text{c}}\text{ffimpl}$，而不考虑把宿主语言运行时嵌入外部函数这种特殊情形，即独立的 $\text{ffimpl}_{\text{c}}\rightarrow_{\text{h}}\text{hfimpl}$。后者可以通过将原本的外部语言视作新的宿主语言来支持，这种对调不会影响本章跨语言调用图构建方法的有效性。

回调行为的外部映射语义模型 $\mathcal{M}'$ 可以表示为以下扩展形式：

$$\begin{aligned}\mathcal{M}' : \{((&\text{ffname}_{\text{h}}\rightarrow_{\text{c}}\text{ffimpl}), \mathit{Kind}, \mathit{Pattern})\}\\ &\rightarrow (\text{ffname}_{\text{h}}\rightarrow_{\text{c}}\text{ffimpl}_{\text{c}}\rightarrow_{\text{h}}\text{hfimpl})\end{aligned}$$

回调始终发生在外部函数 ffname 的外部实现 ffimpl 中。

以基于 Python/C API 的 Python/C 多语言互操作为例，`PyObject_Call*` 族 Python/C API 可以发起一个从外部 C/C++ 侧向宿主 Python 侧的调用：

```
PyObject *PyObject_Call(PyObject *callable,
    PyObject *args, PyObject *kwargs)
```

`callable` 对象一般是以 ffname 为调用名的外部函数的参数之一，并和其他参数一起打包传入外部函数实现，这种特性可以视作多语言互操作中的高阶函数。

基于参数解析 Python/C API 的分析（与第 4 章参数类型转换的参数解析分析类似）可以恢复宿主侧的参数信息，进而将其中的 `callable` 对象和回调函数实现 `hfimpl` 相关联。使用 `PyObject_Call` 特化的外部映射语义模型可以表示为：

$$\mathcal{M}' : \{((\text{ffname}_{\text{h}} \rightarrow_{\text{c}} \text{ffimpl}), \mathit{func}, \texttt{PyObject\_Call} < \text{hfimpl}, \cdots >)\}$$
$$\rightarrow (\text{ffname}_{\text{h}} \rightarrow_{\text{c}} \text{ffimpl}_{\text{c}} \rightarrow_{\text{h}} \text{hfimpl})$$

对于回调类方法的外部接口，抽象模式 *Pattern* 有形式 $\mathit{apiname} < \mathit{hobj}, \mathit{name}, \cdots >$，$\text{hfimpl} = \mathit{hobj.name}$。

考虑发起回调的 Node.js C++ addons 如

```
Call(context, cbfunc, argc, argv)
```

其模式和特化的外部映射语义模型类似：

$$\mathcal{M}' : \{((\text{ffname}_{\text{h}} \rightarrow_{\text{c}} \text{ffimpl}), \mathit{func}, \texttt{Call} < \_, \text{hfimpl}, \cdots >)\}$$
$$\rightarrow (\text{ffname}_{\text{h}} \rightarrow_{\text{c}} \text{ffimpl}_{\text{c}} \rightarrow_{\text{h}} \text{hfimpl})$$

第 5.4 节所述的方法也可以做相应地扩展以支持带有回调节点的跨语言接口层调用子图 CG^i。在表 5.3 列举的多语言软件测试集中，仅有 PyTorch 包含 22 个 `PyObject_Call*` 外部接口调用，并且其中部分发生回调的外部函数并不属于 PyTorch 的核心编程接口子集。

5.6.3　有效性威胁

大多数调用图分析工具不支持多语言软件的程序分析。可用的多语言工具 ffi-navigator 并不提供语言服务之外的独立工具。出于这一原因，RQ1 将 Frog 的分析结果与手工构建的验证集进行比较，这影响了实验的规模；同时 RQ2 只能用 Frog 的分析结果触发 ffi-navigator 的定义跳转功能，导致只能说明 ffi-navigator 相比 Frog 的召回损失。此外，MindSpore 提供的 PyTorch 核心编程接口子集属于旧版本 PyTorch 1.5.0，而实验以新版本 PyTorch 1.9.0 为分析目标。尽管核心编程接口是稳定的，但也不能排除一些后续引入但使用频繁的编程接口被遗漏的可能。

Frog 复用了 VSCode 中的默认语言服务扩展来完成单语言的调用子图构建。在提取宿主侧调用关系时，Frog 使用 Pylance 扩展[234]分析 Python 程序，使用 JavaScript and TypeScript Nightly 扩展[236]分析 JavaScript 程序。因为这些扩展不是开源的，所有它们耗费的分析时间没有计算在 RQ3 的实验结果中。

本节对本章提出的方法进行了理论上的扩展，给出了支持回调行为的解决方案，但是没有对这一部分扩展进行实现和评估。此外，本章没有考虑一些动态特性的处理，如基于反射（reflection）获取函数对象的互操作方法。

5.7　本章小结

本章提出了跨语言调用图构建分析 Frog，其能够支持不同的宿主语言和互操作方法。Frog 实现为语言服务扩展，并包含模块化可独立工作的分析阶段。多语言软件的外部映射关系可以预先计算形成总结，并在跨语言调用图构建或其他程序分析任务中复用。在 10 个带有 C/C++ 扩展模块的 Python 和 JavaScript 多语言软件上的实验表明，Frog 相比已有的工具能够提供更好的泛化能力和更优的召回精度。Frog 可以作为 IDE 语言服务高效地分析多语言软件，并和已有的 IDE 扩展集成工作。

第 6 章　总结与展望

6.1　全文总结

基于编程语言互操作的多语言软件架构已经被广泛使用于现代软件工程不同领域的主流应用中。编程语言互操作能够混合和匹配不同语言的语言特性和编程范式，并复用已有的代码，从而兼具高级语言的开发效率和安全机制，以及低级语言的执行性能等优点。然而，由于语言特性的差异和互操作性的设计，具有不同的异常处理、内存管理、并发机制、类型系统等特性的语言在互操作时需要考虑跨语言编程的额外约束，导致多语言软件更加易错、存在更多的安全隐患。静态分析作为安全检查、程序理解的重要手段，需要支持跨语言互操作的多语言软件以满足现代软件工程的发展。

本文围绕基于语言接口的多语言软件的静态分析，系统地研究了编程语言互操作中语言接口的设计使用与漏洞模式、外部函数的类型推断、跨语言的调用图构建等问题，分析总结了多语言软件的漏洞模式，设计实现了一系列支持编程语言互操作的程序分析技术，在大量使用的开源仓库中发现了多种漏洞，提高了多语言软件的安全性、可靠性和可维护性。本文具体的研究工作包括:

针对由于编程语言互操作的语言接口的差异性，导致缺少对新兴多语言软件架构的设计迭代、使用行为、漏洞模式的分析和了解的问题，本文研究了语言特性差异更大且近年来广泛使用的 Python 和 C/C++ 互操作的多语言软件，设计实现了其语言接口 Python/C API 的提取、分析、漏洞检查的工具集 PyCEAC。首先，PyCEAC 基于编译前端定制了预处理器、宏提取器、AST 解析器等部件，从不同版本的 Python 编译器实现中提取 Python/C API，分析其随编译器迭代的设计变化。接着，PyCEAC 通过跨语言接口分离器、分词器、筛选器等部件从主流的 Python/C 多语言软件，如 Pillow、NumPy、PyTorch、TensorFlow 等中，提取并分析 Python/C API 的使用行为。然后，结合 Python/C API 的设计迭代、使用行为，以及对 Python/C 语言特性差异的分析，本文分析总结了 Python/C 多语言软件潜在的 9 类漏洞模式。最后，PyCEAC 通过语法模式匹配、集合运算、接入第三方工具的方法设计实现了其中 6 类漏洞的轻量级的漏洞检查工具，并在大量使用的 Python/C 多语言软件 Pillow 和 NumPy 中发现了漏洞实例。

针对已有的确定性的、不基于类型标注的类型推断方法难以有效推断动态类型语言的外部函数的问题，本文研究了以类型系统为代表的复杂语言特性的跨语言程序分析技术，提出了一种不依赖类型标注的、确定性的 Python 的 C 外部函数的静态类型推断方法 PyCType。首先，PyCType 形式建模了 Python/C 跨语言程序的静态语义，包括抽象语法、类型系统和子定型规则。然后，PyCType

把外部函数类型推断的前提表示为三类可组合的假设判断，分别对应外部函数声明、参数类型转换、返回类型转换三类语言接口，根据其具体形式和组合推断外部函数的函数签名。其中，外部函数声明只有一种形式，参数类型转换包含基于未使用参数分析的调用惯例分析和基于格式化串的参数解析分析两种形式，返回类型转换包含四种不同的分析形式，分别是值构建分析、显式转换分析、类型转换分析、可达定义分析。最后，作为类型推断的副产物，PyCType 可以通过未使用参数分析中门限语义谓词的真假检查外部函数声明与实现不一致的漏洞，该漏洞会导致无参外部函数可以接收任意类型的参数。在大量使用的 Python/C 多语言软件 Pillow、NumPy、CPython 标准库上的实验表明，PyCType 可以无误报地推理外部函数的类型签名并发现了多个一致性漏洞。同时在多个基于 Pillow 的主流应用程序上的实验表明，作为对 Google 的 Python 单语言静态类型推断工具 Pytype 的增强，PyCType 可以将其类型召回率提高 27.5%。

针对跨语言程序分析技术在不同的宿主语言和互操作方法上泛化能力不足、缺乏控制流等跨语言基础信息的问题，本文提出了一种支持多种宿主语言和外部接口的跨语言调用图构建方法 Frog。Frog 使用一个两阶段的程序分析：离线的阶段一定义了外部映射的语义模型，抽象描述不同互操作机制下的外部函数声明，提取跨语言的映射关系；阶段二基于语言无关的图变换与节点融合算法，在线地构建完整的跨语言调用图。在 Python/C 的语言接口 Python/C API、接口生成工具 pybind11、JavaScript/C 的语言接口 Node.js C++ addons 上的实验表明，Frog 在具有泛化性的同时，相比已有的基本专家定制提取规则的工具 ffi-navigator 取得了更好的精度，并保持了较好的时间效率。

6.2　未来展望

多语言软件的静态分析吸引了学界和业界越来越多的关注和研究，但是仍然存在许多尚未解决的问题和挑战，尤其是随着新的语言特性和编程范式的不断发展和出现。

6.2.1　高性能的语言接口

跨语言的外部调用存在额外的性能开销。使用外部语言实现单语言软件的部分组件能够获得正向收益依赖于两部分的性能表现：（1）外部代码实现，（2）桥接宿主语言程序和外部代码的跨语言接口代码。在多数情况下，高级语言语法更加灵活、开发效率更高，更适合进行应用软件的原型实现；但是在系统优化过程中，尤其是在发现性能瓶颈后，开发者倾向于使用性能更加高效的的语言如 C/C++ 重写高级宿主语言的组件。当宿主语言本身性能表现相对较优（如 Go 或

Rust），且重写部分不是计算密集型的时候，多语言软件系统的性能表现将取决于跨语言接口代码的性能。与多语言软件的安全性相比，语言接口的性能优化没有得到足够的研究，一些互操作性的安全机制研究甚至会降低多语言软件的性能表现。例如，跨语言接口的自动生成工具[121,212]具有更好的易用性，但是也损失了跨语言接口代码的灵活性和性能调优空间。而将外部对象移动到宿主堆上的内存管理机制[86-87]虽然提高了安全性但是却可能更频繁地触发宿主语言的垃圾收集，影响多语言软件的运行时性能。设计实现高性能的语言接口可以进一步扩大多语言软件架构的适用范围，使得已有的外部库能够得到广泛的复用。

6.2.2　多语言软件的静态内存模型

作为一个基础程序分析，跨语言的调用图构建回答多语言软件中关于控制流的问题，而另一个尚未得到仔细研究的基础程序信息则是跨语言的数据流。经典的指针分析可以提供程序中基础的数据流信息，是对堆内存的静态建模，单语言程序上的指针分析研究已经说明指向信息可以提高多种程序分析任务的精度。由宿主语言或通用运行时提供外部接口是最常见的多语言互操作机制，其中宿主语言和外部语言共用堆内存，已有的跨语言研究往往在涉及指针类型和混合内存时需要做保守的假设或特化的处理，存在精度不足、缺乏一般性的问题。此外，C、C++、Go 等语言带有显式的指针数据类型，以及指针的解引用，而 Java、Python 等语言虽然存在引用类型却没有显式指针，这种类型系统的差异性涉及指针和引用类型的分析，在已有的多语言类型研究中仍未得到详尽的分析。设计跨语言的指针分析能够回答宿主语言对象和外部语言对象之间的引用关系，给出共享堆的静态建模，为跨语言的内存、类型等复杂语言特性的程序分析提供基础。

6.2.3　复杂语言特性的互操作语义

互操作性的形式语义研究基于形式化方法严格描述跨语言程序的语义和安全规范，已有的工作已经研究了编程语言互操作的类型安全[76-77]、上下文等价[74,247]、可变状态、异常处理[46]等程序性质。然而，仍然存在一些涉及复杂语言特性的编程语言互操作语义尚未得到仔细研究。例如，惰性求值（lazy evaluation）语言和急迫求值（eager evaluation）语言之间的互操作、单线程语言和并发语言之间的互操作、所有权（ownership）语言和垃圾收集语言之间的互操作，等等。

参 考 文 献

[1] SARMANNE M. Language performance and productivity: Python versus c, c++ and java [D]. Aalto University, 2010.

[2] SKALICKY S, MONSON J, SCHMIDT A, et al. Hot & spicy: Improving productivity with python and hls for fpgas[C]//Proceedings of the 26th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). Boulder, CO, USA: IEEE, 2018: 85-92.

[3] WOODS J, BAKER M, THAVAPPIRAGASAM M, et al. Using python for improved productivity in hpc and data science applications: The time is now[C]//Proceedings of the 2nd Collegeville Workshop on Scientific Software (CW). 2020.

[4] JUNG R, JOURDAN J H, KREBBERS R, et al. Safe systems programming in rust[J]. Communications of the ACM, 2021, 64(4): 144-152.

[5] TOGASHI N, KLYUEV V. Concurrency in go and java: Performance analysis[C]// Proceedings of the 4th International Conference on Information and Software Technologies (ICIST). IEEE, 2014: 213-216.

[6] MUMFORD S J, CHRISTE S, PÉREZ-SUÁREZ D, et al. Sunpy—python for solar physics [J]. Computational Science & Discovery, 2015, 8(1): 014009.

[7] DAOUDA T, PERREAULT C, LEMIEUX S. pygeno: A python package for precision medicine and proteogenomics[J]. F1000Research, 2016, 5(381): 381.

[8] RASCHKA S, PATTERSON J, NOLET C. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence[J]. Information, 2020, 11(4): 193.

[9] HARRIS C R, MILLMAN K J, VAN DER WALT S J, et al. Array programming with numpy [J]. Nature, 2020, 585(7825): 357-362.

[10] JETBRAINS. The state of developer ecosystem 2022[EB/OL]. 2022. https://www.jetbrains.com/lp/devecosystem-2022/.

[11] BLACKFORD L S, PETITET A, POZO R, et al. An updated set of basic linear algebra subprograms (blas)[J]. ACM Transactions on Mathematical Software (TOMS), 2002, 28(2): 135-151.

[12] GAILLY J L, ADLER M. Zlib compression library[Z]. 2004.

[13] ABADI M. Tensorflow: Learning functions at scale[C]//Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFL). 2016: 1-1.

[14] PASZKE A, GROSS S, MASSA F, et al. Pytorch: An imperative style, high-performance

deep learning library[C]//Proceedings of the 32nd Annual Conference on Neural Information Processing Systems (NeurIPS). 2019: 8026-8037.

[15] FAN H, ZHU F, LIU C, et al. Baidu apollo em motion planner[A]. 2018. arXiv: 1807.08048.

[16] JOHN K, O'HARA M, SALEH F. Bitcoin and beyond[J]. Annual Review of Financial Economics, 2022, 14(1): 95-115.

[17] WALSH L, AKHMECHET V, GLUKHOVSKY M. Rethinkdb—rethinking database storage [J]. Hexagram 49, Inc., 2009.

[18] PALANIAPPAN M, YANKELOVICH N, FITZMAURICE G, et al. The envoy framework: An open architecture for agents[J]. ACM Transactions on Information Systems (TOIS), 1992, 10(3): 233-264.

[19] LUGARESI C, TANG J, NASH H, et al. Mediapipe: A framework for building perception pipelines[A]. 2019. arXiv: 1906.08172.

[20] 冯森林, 张玺. Weex 跨平台开发方案研究与应用[J]. 信息通信, 2017(4): 2.

[21] KREDPATTANAKUL K, LIMPIYAKORN Y. Transforming javascript-based web application to cross-platform desktop with electron[C]//Proceedings of the International Conference on Information Science and Applications (ICISA). Springer, 2019: 571-579.

[22] LING M, YU Y, WU H, et al. In rust we trust: A transpiler from unsafe c to safer rust [C]//Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). Pittsburgh, PA, USA, 2022: 354-355.

[23] KIRTH P, DICKERSON M, CRANE S, et al. Pkru-safe: Automatically locking down the heap between safe and unsafe languages[C]//Proceedings of the 17th European Conference on Computer Systems (EuroSys). Rennes, France, 2022: 132-148.

[24] GRICHI M, ABIDI M, JAAFAR F, et al. On the impact of inter-language dependencies in multi-language systems[C]//Proceedings of the 21st International Conference on Software Quality, Reliability and Security (QRS). Macau, SAR, China: IEEE, 2020: 428-440.

[25] LI W, LI L, CAI H. On the vulnerability proneness of multilingual code[C]//Proceedings of the 30th ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE). Singapore: ACM, 2022: 847-859.

[26] Cve-2023-4785[EB/OL]. 2023. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-4785.

[27] Cve-2019-29571[EB/OL]. 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29571.

[28] Cve-2019-29572[EB/OL]. 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29572.

[29] Cve-2023-38703[EB/OL]. 2023. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-20

23-38703.

[30] Cve-2019-25055[EB/OL]. 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-25055.

[31] Cve-2015-7551[EB/OL]. 2015. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7551.

[32] Cve-2007-4528[EB/OL]. 2007. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4528.

[33] LI S, TAN G. Finding reference-counting errors in python/c programs with affine analysis[C]// Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP). Uppsala, Sweden: Springer, 2014: 80-104.

[34] MAO J, CHEN Y, XIAO Q, et al. Rid: Finding reference count bugs with inconsistent path pair checking[C]//Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). Atlanta, GA, USA: ACM, 2016: 531-544.

[35] LI S, TAN G. Finding bugs in exceptional situations of jni programs[C]//Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS). Chicago, IL, USA: ACM, 2009: 442-452.

[36] LI S, TAN G. Jet: Exception checking in the java native interface[C]//Proceedings of the 26th ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Portland, Oregon, USA: ACM, 2011: 345-358.

[37] KONDOH G, ONODERA T. Finding bugs in java native interface programs[C]//Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA). Seattle, WA, USA: ACM, 2008: 109-118.

[38] TAN G, CROFT J. An empirical security study of the native code in the jdk[C]//Proceedings of the 17th Usenix Security Symposium. San Jose, CA, USA: USENIX, 2008: 365-378.

[39] LI Z, WANG J, SUN M, et al. Detecting cross-language memory management issues in rust[C]//Proceedings of the 27th European Symposium on Research in Computer Security (ESORICS). Copenhagen, Denmark: Springer, 2022: 680-700.

[40] FURR M, FOSTER J S. Checking type safety of foreign function calls[C]//Proceedings of the 26th ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI). Chicago, IL, USA: ACM, 2005: 62-72.

[41] TAN G, APPEL A W, CHAKRADHAR S, et al. Safe java native interface[C]//Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE). McLean, VA, USA: Citeseer, 2006: 106.

[42] LI S, LIU Y D, TAN G. Jato: Native code atomicity for java[C]//Proceedings of the 9th Asian

Symposium on Programming Languages and Systems (APLAS). Kenting, Taiwan: Springer, 2012: 2-17.

[43] 张健, 张超, 玄跻峰, 等. 程序分析研究进展[J]. 软件学报, 2019, 30(1): 30.

[44] COUSOT P, COUSOT R, FERET J, et al. The astrée analyzer[C]//Proceedings of the 14th European Symposium on Programming (ESOP). Springer, 2005: 21-30.

[45] ARZT S, RASTHOFER S, FRITZ C, et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps[J]. ACM Sigplan Notices, 2014, 49(6): 259-269.

[46] TRIFONOV V, SHAO Z. Safe and principled language interoperation[C]//Proceedings of the 8th European Symposium on Programming (ESOP). Amsterdam, The Netherlands: Springer, 1999: 128-146.

[47] ZHOU W, ZHAO Y, ZHANG G, et al. Harp: Holistic analysis for refactoring python-based analytics programs[C]//Proceedings of the 42nd International Conference on Software Engineering (ICSE). Seoul, Korea: IEEE, 2020: 506-517.

[48] RICE H G. Classes of recursively enumerable sets and their decision problems[J]. Transactions of the American Mathematical Society, 1953, 74(2): 358-366.

[49] CHEN Z, LI Y, CHEN B, et al. An empirical study on dynamic typing related practices in python systems[C]//Proceedings of the 28th IEEE International Conference on Program Comprehension (ICPC). 2020: 83-93.

[50] WANG X, CHEN H, CHEUNG A, et al. Undefined behavior: What happened to my code? [C]//Proceedings of the 3rd Asia Pacific Workshop on Systems (APSys). 2012: 1-7.

[51] LI W, MENG N, LI L, et al. Understanding language selection in multi-language software projects on github[C]//Proceedings of the 43th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). Pittsburgh, PA, USA: IEEE, 2021: 256-257.

[52] LI W, LI L, CAI H. Polyfax: A toolkit for characterizing multi-language software[C]// Proceedings of the 30th ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE). Singapore: ACM, 2022: 1662-1666.

[53] LI W, MING J, LUO X, et al. Polycruise: A cross-language dynamic information flow analysis [C]//Proceedings of the 31st USENIX Security Symposium. Boston, MA, USA: USENIX, 2022: 2513-2530.

[54] ZHANG X, MA X, YAN J, et al. Improving tese case generation for python native libraries through constraints on input data structures[A]. 2022. arXiv: 2206.13828.

[55] ZHANG X, YAN R, YAN J, et al. Excepy: A python benchmark for bugs with python built-in types[C]//Proceedings of the 29th IEEE International Conference on Software Analysis,

Evolution, and Reengineering (SANER). Honolulu, HI, USA: IEEE, 2022: 856-866.

[56] JIANG C, HUA B, OUYANG W, et al. Pyguard: Finding and understanding vulnerabilities in python virtual machines[C]//Proceedings of the 32nd IEEE International Symposium on Software Reliability Engineering (ISSRE). Wuhan, China: IEEE, 2021: 468-475.

[57] LIN X, HUA B, FAN Q. On the security of python virtual machines: An empirical study [C]//Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME). Limassol, Cyprus: IEEE, 2022: 223-234.

[58] HAN X, HUA B, WANG Y, et al. Rusty: Effective c to rust conversion via unstructured control specialization[C]//Proceedings of the 22nd International Conference on Software Quality, Reliability and Security (QRS). Guangzhou, China: IEEE, 2022: 760-761.

[59] HU S, HUA B, XIA L, et al. Crust: Towards a unified cross-language program analysis framework for rust[C]//Proceedings of the 22nd International Conference on Software Quality, Reliability and Security (QRS). Guangzhou, China: IEEE, 2022: 970-981.

[60] MAYER P, KIRSCH M, LE M A. On multi-language software development, cross-language links and accompanying tools: A survey of professional software developers[J]. Journal of Software Engineering Research and Development (JSERD), 2017, 5: 1-33.

[61] SULTANA N, MIDDLETON J, OVERBEY J, et al. Understanding and fixing multiple language interoperability issues: The c/fortran case[C]//Proceedings of the 38th International Conference on Software Engineering (ICSE). Austin, Texas, USA: ACM, 2016: 772-783.

[62] ADAMS J C, BRAINERD W S, HENDRICKSON R A, et al. The fortran 2003 handbook: The complete syntax, features and procedures[M]. Springer Science & Business Media, 2008.

[63] MAGMA. Exploiting memory corruptions in fortran programs under unix/vms[J/OL]. Phrack Inc., 2010, 67. http://phrack.org/issues/67/11.html.

[64] SALTZER J H, SCHROEDER M D. The protection of information in computer systems[J]. Proceedings of the IEEE, 1975, 63(9): 1278-1308.

[65] BINKLEY D, GOLD N, HARMAN M, et al. Orbs: Language-independent program slicing [C]//Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). 2014: 109-120.

[66] YANG H, LI W, CAI H. Language-agnostic dynamic analysis of multilingual code: Promises, pitfalls, and prospects[C]//Proceedings of the 30th ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE). Singapore: ACM, 2022: 1621-1626.

[67] GRIMMER M, SEATON C, SCHATZ R, et al. High-performance cross-language interoperability in a multi-language runtime[C]//Proceedings of the 11th Dynamic Languages Symposium (DLS). Pittsburgh, PA, USA: ACM, 2015: 78-90.

[68] JOUVELOT P, GIFFORD D. Algebraic reconstruction of types and effects[C]//Proceedings

of the 18th ACM-SIGACT Symposium on Principles of Programming Languages (POPL). 1991: 303-310.

[69] TALPIN J P, JOUVELOT P. Polymorphic type, region and effect inference[J]. Journal of Functional Programming, 1992, 2(3): 245-271.

[70] TALPIN J P, JOUVELOT P. The type and effect discipline[J]. Information and Computation, 1994, 111(2): 245-296.

[71] REYNOLDS J C. Types, abstraction and parametric polymorphism[C]//Information Processing 83, Proceedings of the IFIP 9th World Computer Congress. 1983: 513-523.

[72] ABADI M. Protection in programming-language translations[C]//Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP). Springer, 1998: 868-883.

[73] KENNEDY A. Securing the .net programming model[J]. Theoretical Computer Science, 2006, 364(3): 311-317.

[74] AHMED A, BLUME M. An equivalence-preserving cps translation via multi-language semantics[C]//Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP). Tokyo, Japan: ACM, 2011: 431-444.

[75] LARMUSEAU A, PATRIGNANI M, CLARKE D. Operational semantics for secure interoperation[C]//Proceedings of the 9th Workshop on Programming Languages and Analysis for Security (PLAS). Uppsala, Sweden: ACM, 2014: 40-52.

[76] MATTHEWS J, FINDLER R B. Operational semantics for multi-language programs[C]// Proceedings of the 34th ACM-SIGACT Symposium on Principles of Programming Languages (POPL). Nice, France: ACM, 2007: 3-10.

[77] PATTERSON D, MUSHTAK N, WAGNER A, et al. Semantic soundness for language interoperability[C]//Proceedings of the 43rd ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI). San Diego, CA, USA: ACM, 2022: 609-624.

[78] BENTON N, ZARFATY U. Formalizing and verifying semantic type soundness of a simple compiler[C]//Proceedings of the 9th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP). Wroclaw, Poland: ACM, 2007: 1-12.

[79] BENTON N, TABAREAU N. Compiling functional types to relational specifications for low level imperative code[C]//Proceedings of the 4th ACM SIGPLAN International Workshop on Types In Languages Design And Implementation (TLDI). Savannah, GA, USA: ACM, 2009: 3-14.

[80] LIANG S. The java native interface: Programmer's guide and specification[M]. Boston, MA, USA: Addison-Wesley Professional, 1999.

[81] STROM R E, YEMINI S. Typestate: A programming language concept for enhancing soft-

ware reliability[J]. IEEE Transactions on Software Engineering (TSE), 1986(1): 157-171.

[82] CALCAGNO C, DISTEFANO D. Infer: An automatic program verifier for memory safety of c programs[C]//Proceedings of the 3rd NASA Formal Methods International Symposium (NFM). Pasadena, CA, USA: Springer, 2011: 459-465.

[83] CHRISTOPHER T W. Reference count garbage collection[J]. Software: Practice and Experience, 1984, 14(6): 503-507.

[84] SIMONS A J. Borrow, copy or steal? loans and larceny in the orthodox canonical form[C]// Proceedings of the 13th ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Vancouver, British Columbia, Canada: ACM, 1998: 65-83.

[85] LAL A, RAMALINGAM G. Reference count analysis with shallow aliasing[J]. Information Processing Letters, 2010, 111(2): 57-63.

[86] SORENSEN J, BIKEL D. Improved jni memory management using allocations from the java heap[C]//USENIX Annual Technical Conference (USENIX ATC). Santa Clara, CA, USA: USENIX, 2007.

[87] GRIMMER M, SCHATZ R, SEATON C, et al. Memory-safe execution of c on a java vm [C]//Proceedings of the 10th ACM SIGPLAN International Workshop on Programming Languages and Analysis for Security (PLAS). Prague, Czech Republic: ACM, 2015: 16-27.

[88] LI D, SRISA-AN W. Quarantine: A framework to mitigate memory errors in jni applications [C]//Proceedings of the 9th International Conference on Managed Programming Languages & Runtimes (MPLR). Kongens Lyngby, Denmark: ACM, 2011: 1-10.

[89] DISSELKOEN C, RENNER J, WATT C, et al. Position paper: Progressive memory safety for webassembly[C]//Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP). Phoenix, AZ, USA: ACM, 2019: 1-8.

[90] FURR M, FOSTER J S. Polymorphic type inference for the jni[C]//Proceedings of the 15th European Symposium on Programming (ESOP). Vienna, Austria: Springer, 2006: 309-324.

[91] MARLOW S, JONES S P, THALLER W. Extending the haskell foreign function interface with concurrency[C]//Proceedings of the ACM SIGPLAN International Symposium on Haskell (Haskell). Snowbird, UT, USA: ACM, 2004: 22-32.

[92] ARMSTRONG J, VIRDING R. One pass real-time generational mark-sweep garbage collection[C]//Proceedings of the International Workshop on Memory Management (IWMM). Kinross, Scotland, UK: Springer, 1995: 313-322.

[93] PARR T, HARWELL S, FISHER K. Adaptive ll (*) parsing: The power of dynamic analysis [C]//Proceedings of the 29th ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). 2014: 579-598.

[94] MALONE T. Interoperability in programming languages[J]. Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal, 2014, 1(2): 3.

[95] MATTHEWS J B. The meaning of multilanguage programs[D]. The University of Chicago, 2008.

[96] IBM. Fortran ii for the ibm 704 data processing system[EB/OL]. 1958. http://bitsavers.org/pdf/ibm/704/C28-6000-2_704_FORTRANII.pdf.

[97] BURROUGHS. Burroughs algebraic compiler: A representation of algol for use with the burroughs 220 data-processing system[EB/OL]. 1961. http://www.bitsavers.org/pdf/burroughs/electrodata/220/220-21011-D_BALGOL_Jan61.pdf.

[98] HOFFMAN H J. The ibm os/360 algol 60 compiler[EB/OL]. 1998. http://www.h-j-hoffmann.de/PU/docs/HJH-199804xx-Algol60.rtf.

[99] ICHBIAH J D. Preliminary ada reference manual[J]. ACM Sigplan Notices, 1979, 14(6a): 1-145.

[100] OSPINA G A, CHARLIER B L. Towards precise descriptions for programming language interoperability: A general approach based on operational semantics[C]//Proceedings of the Interoperability for Enterprise Software and Applications (I-ESA). Funchal, Madeira Island, Portugal: Springer, 2007: 581-586.

[101] LEE S, LEE H, RYU S. Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis[C]//Proceedings of the 35th International Conference on Automated Software Engineering (ASE). Melbourne, Australia: IEEE, 2020: 127-137.

[102] SUTTER H. Defining a portable c++ abi[J/OL]. C++ Standards Committee Papers, 2014. http://vps-93-90-116-65.vpsforce.eu/JTC1/SC22/WG21/docs/papers/2014/n4028.pdf.

[103] KLOCK II F. The layers of larceny's foreign function interface[C]//Proceedings of the Annual Workshop on Scheme and Functional Programming. Victoria, BC, Canada: ACM, 2008: 69-79.

[104] JONES S P. Green card: A foreign-language interface for haskell[EB/OL]. 1997. https://www.haskell.org/nhc98/greencard.html.

[105] DOWD T, HENDERSON F, ROSS P. Compiling mercury to the .net common language runtime[J]. Electronic Notes in Theoretical Computer Science (ENTCS), 2001, 59(1): 73-88.

[106] CARLISLE M C, SWARD R E, HUMPHRIES J W. Weaving ada 95 into the .net environment [C]//Proceedings of the Annual ACM SIGAda International Conference on Ada (AIGAda). Houston, Texas, USA: ACM, 2002: 22-26.

[107] BENTON N, KENNEDY A, RUSSO C V. Adventures in interoperability: The sml .net ex-

perience[C]//Proceedings of the 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP). Verona, Italy: ACM, 2004: 215-226.

[108] GORDON A D, SYME D. Typing a multi-language intermediate code[C]//Proceedings of the 28th ACM-SIGACT Symposium on Principles of Programming Languages (POPL). London, UK: ACM, 2001: 248-260.

[109] KENNEDY A, SYME D. Design and implementation of generics for the .net common language runtime[C]//Proceedings of the 22nd ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI). Snowbird, Utah, USA: ACM, 2001: 1-12.

[110] YU D, KENNEDY A, SYME D. Formalization of generics for the .net common language runtime[C]//Proceedings of the 31st ACM-SIGACT Symposium on Principles of Programming Languages (POPL). Venice, Italy: ACM, 2004: 39-51.

[111] TAN G, MORRISETT G. Ilea: Inter-language analysis across java and c[C]//Proceedings of the 22nd ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Montreal, Quebec, Canada: ACM, 2007: 39-56.

[112] DUBOSCQ G, STADLER L, WÜRTHINGER T, et al. Graal ir: An extensible declarative intermediate representation[C]//Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop (APPLC). Shenzhen, China: IEEE, 2013.

[113] WÜRTHINGER T, WIMMER C, WÖSS A, et al. One vm to rule them all[C]//Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). Indianapolis, IN, USA: ACM, 2013: 187-204.

[114] GRIMMER M, SEATON C, WÜRTHINGER T, et al. Dynamically composing languages in a modular way: Supporting c extensions for dynamic languages[C]//Proceedings of the 14th International Conference on Modularity (MODULARITY). Fort Collins, CO, USA: ACM, 2015: 1-13.

[115] GRIMMER M, SCHATZ R, SEATON C, et al. Cross-language interoperability in a multi-language runtime[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2018, 40(2): 1-43.

[116] RICHTHOFER S. Garbage collection in jyni - how to bridge mark/sweep and reference counting gc[A]. 2016. arXiv: 1607.00825.

[117] BIRRELL A D, NELSON B J. Implementing remote procedure calls[J]. ACM Transactions on Computer Systems (TOCS), 1984, 2(1): 39-59.

[118] BERSHAD B N, CHING D T, LAZOWSKA E D, et al. A remote procedure call facility for interconnecting heterogeneous computer systems[J]. IEEE Transactions on Software Engi-

neering (TSE), 1987, 13(8): 880-894.

[119] YANG Z, DUDDY K. Corba: A platform for distributed object computing[J]. ACM SIGOPS: Operating Systems Review, 1996, 30(2): 4-31.

[120] SESSIONS R. Com and dcom: Microsoft's vision for distributed objects[M]. New York, NY, USA: John Wiley & Sons, Inc., 1997.

[121] BEAZLEY D M, et al. Swig: An easy to use tool for integrating scripting languages with c and c++[C]//Proceedings of the Tcl/Tk Workshop. Monterey, California, USA: USENIX, 1996: 74.

[122] DIMMICH D J, JACOBSEN C L. A foreign function interface generator for occam-pi[C]// Proceedings of the 28th Communicating Process Architectures Conference (CPA). Eindhoven, The Netherlands: IOS Press, 2005: 235-248.

[123] HOARE C A R. Communicating sequential processes[J]. Communications of the ACM, 1978, 21(8): 666-677.

[124] MILNER R, PARROW J, WALKER D. A calculus of mobile processes, i[J]. Information and computation, 1992, 100(1): 1-40.

[125] REPPY J, SONG C. Application-specific foreign-interface generation[C]//Proceedings of the 5th International Conference on Generative Programming: Concepts and Experiences (GPCE). Portland, Oregon, USA: ACM, 2006: 49-58.

[126] RAVITCH T, JACKSON S, ADERHOLD E, et al. Automatic generation of library bindings using static analysis[C]//Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Dublin, Ireland: ACM, 2009: 352-362.

[127] ALEKSYUK A, ITSYKSON V. Automated cross-language integration based on formal model of components[C]//Proceedings of the 1st Frontiers in Software Engineering Education (FISEE). Villebrumier, France: Springer, 2019: 357-370.

[128] ITSYKSON V. Libsl: Language for specification of software libraries[J]. Softw. Eng, 2018, 9: 209-220.

[129] ZHU S, ALAWAR N, EREZ M, et al. Dynamic generation of python bindings for hpc kernels [C]//Proceedings of the 36th International Conference on Automated Software Engineering (ASE). Melbourne, Australia: IEEE, 2021: 92-103.

[130] MØLLER A, SCHWARTZBACH M I. Static program analysis[R]. Aarhus University, 2012.

[131] KILDALL G A. A unified approach to global program optimization[C]//Proceedings of the 1st ACM-SIGACT Symposium on Principles of Programming Languages (POPL). Boston, Massachusetts, USA: ACM, 1973: 194-206.

[132] KAM J B, ULLMAN J D. Monotone data flow analysis frameworks[J]. Acta Informatica, 1977, 7(3): 305-317.

[133] COUSOT P, COUSOT R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints[C]//Proceedings of the 4th ACM-SIGACT Symposium on Principles of Programming Languages (POPL). Los Angeles, California, USA: ACM, 1977: 238-252.

[134] COUSOT P, COUSOT R. Static determination of dynamic properties of generalized type unions[J]. ACM SIGOPS Operating Systems Review, 1977, 11(2): 77-94.

[135] AIKEN A, MURPHY B. Static type inference in a dynamically typed language[C]// Proceedings of the 18th ACM-SIGACT Symposium on Principles of Programming Languages (POPL). 1991: 279-290.

[136] AHO A V, LAM M S, SETHI R, et al. Compilers: Principles, techniques and tools[M]. Pearson Education, 1986.

[137] ANDERSEN L O. Program analysis and specialization for the c programming language[D]. University of Copenhagen, 1994.

[138] 谭添, 马晓星, 许畅, 等. Java 指针分析综述[J]. 计算机研究与发展, 2023, 60(2022-20901): 274.

[139] REPS T, HORWITZ S, SAGIV M. Precise interprocedural dataflow analysis via graph reachability[C]//Proceedings of the 22nd ACM-SIGACT Symposium on Principles of Programming Languages (POPL). 1995: 49-61.

[140] BODDEN E. Inter-procedural data-flow analysis with ifds/ide and soot[C]//Proceedings of the 1st ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP). Beijing, China: ACM, 2012: 3-8.

[141] IBM T.J. Watson Research Center. Watson libraries for analysis (wala)[Z]. 2017.

[142] TIOBE Software. Tiobe index[EB/OL]. 2023. https://www.tiobe.com/tiobe-index/.

[143] CAI X, LANGTANGEN H P, MOE H. On the performance of the python programming language for serial and parallel scientific computations[J]. Scientific Programming, 2005, 13 (1): 31-56.

[144] WAGNER M, LLORT G, MERCADAL E, et al. Performance analysis of parallel python applications[J]. Procedia Computer Science, 2017, 108: 2171-2179.

[145] MEIER R, GROSS T R. Reflections on the compatibility, performance, and scalability of parallel python[C]//Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS). 2019: 91-103.

[146] CLARK A. Pillow python imaging library[EB/OL]. 2019. https://pillow.readthedocs.io/en /5.4.1/index.html.

[147] LU K, PAKKI A, WU Q. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences[C]//Proceedings of the 28th USENIX Security Symposium

(USENIX Security). Santa Clara, CA, USA: USENIX Association, 2019: 1769-1786.

[148] HOLKNER A, HARLAND J. Evaluating the dynamic behaviour of python applications[C]// Proceedings of the 32nd Australasian Conference on Computer Science. 2009: 19-28.

[149] MARTINSEN J K, GRAHN H, ISBERG A. A comparative evaluation of JavaScript execution behavior[C]//Proceedings of the 11th International Conference on Web Engineering (ICWE). Paphos, Cyprus, 2011: 399-402.

[150] BOLZ C F, CUNI A, FIJALKOWSKI M, et al. Tracing the meta-level: Pypy's tracing jit compiler[C]//Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS). Genoa, Italy: ACM, 2009: 18-25.

[151] POWER R, RUBINSTEYN A. How fast can we make interpreted python?[A]. 2013. arXiv: 1306.6047.

[152] LAM S K, PITROU A, SEIBERT S. Numba: A llvm-based python jit compiler[C]// Proceedings of the 2nd Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). Austin, Texas: IEEE, 2015: 7:1-7:6.

[153] CHUGH R, RONDON P M, JHALA R. Nested refinements: A logic for duck typing[C]// Proceedings of the 39th ACM-SIGACT Symposium on Principles of Programming Languages (POPL). Philadelphia, Pennsylvania, USA: ACM, 2012: 231-244.

[154] LEVKIVSKYI I, LEHTOSALO J, LANGA Ł. Pep 544 – protocols: Structural subtyping (static duck typing)[EB/OL]. 2017. https://www.python.org/dev/peps/pep-0544.

[155] FRITZ L, HAGE J. Cost versus precision for approximate typing for python[C]//Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM). 2017: 89-98.

[156] HASSAN M, URBAN C, EILERS M, et al. Maxsmt-based type inference for python 3[C]// Proceedings of the 30th International Conference on Computer Aided Verification (CAV). Oxford, UK: Springer, 2018: 12-19.

[157] MONAT R, OUADJAOUT A, MINÉ A. Static type analysis by abstract interpretation of python programs[C]//Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP). Berlin, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[158] ROSSUM G V, LEHTOSALO J, LANGA Ł. Pep 484 – type hints[EB/OL]. 2014. https: //www.python.org/dev/peps/pep-0484.

[159] XU Z, ZHANG X, CHEN L, et al. Python probabilistic type inference with natural language support[C]//Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). 2016: 607-618.

[160] ALLAMANIS M, BARR E T, DUCOUSSO S, et al. Typilus: Neural type hints[C]// Proceedings of the 41st ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI). London, UK: ACM, 2020: 91-105.

[161] PENG Y, GAO C, LI Z, et al. Static inference meets deep learning: A hybrid type inference approach for python[C]//Proceedings of the 44th International Conference on Software Engineering (ICSE). 2022: 2019-2030.

[162] GINSBACH P A. From constraint programming to heterogeneous parallelism[D]. The University of Edinburgh, 2020.

[163] ESPINHA T, ZAIDMAN A, GROSS H G. Web api growing pains: Loosely coupled yet strongly tied[J]. Journal of Systems and Software (JSS), 2015, 100: 27-43.

[164] BENDERSKY E. On parsing c, type declarations and fake headers[EB/OL]. 2015. https://eli.thegreenplace.net/2015/on-parsing-c-type-declarations-and-fake-headers.

[165] CLANG. libclang: C interface to clang[EB/OL]. 2019. https://clang.llvm.org/doxygen/group__CINDEX.html.

[166] JETBRAINS. Python developers survey 2021 results[EB/OL]. 2022. https://lp.jetbrains.com/python-developers-survey-2021.

[167] SELIVANOV Y. Pep 567 – context variables[EB/OL]. 2018. https://peps.python.org/pep-0567.

[168] STINNER V, COGHLAN N. PEP 587 – python initialization configuration[EB/OL]. 2019. https://peps.python.org/pep-0587.

[169] DWINTERGRUEN. Numpy test fails with python3.7 - pytracemalloc_untrack[EB/OL]. 2017. https://github.com/numpy/numpy/issues/9227.

[170] HAMERS L, HEMERYCK Y, HERWEYERS G, et al. Similarity measures in scientometric research: The jaccard index versus salton's cosine formula[J]. Information Processing and Management, 1989, 25(3): 315-318.

[171] ANDERSON T W. The statistical analysis of time series[M]. John Wiley & Sons, 2011.

[172] ROSSUM G V, DRAKE F L. Python/c api reference manual[EB/OL]. 2019. https://docs.python.org/3/c-api/index.html.

[173] DIJKSTRA E. A case against the goto statement[J]. Communications of the ACM, 1968, 11: 147.

[174] PILLOW. Return after error[EB/OL]. 2019. https://github.com/python-pillow/Pillow/pull/3967.

[175] PILLOW. Fixed freeing unallocated pointer when resizing with height too large[EB/OL]. 2019. https://github.com/python-pillow/Pillow/pull/4116.

[176] PILLOW. Pass the correct types to pyarg_parsetuple[EB/OL]. 2019. https://github.com/pyt

hon-pillow/Pillow/pull/3880.

[177] MALLOY B A, POWER J F. Quantifying the transition from python 2 to 3: An empirical study of python applications[C]//Proceedings of the 11th International Symposium on Empirical Software Engineering and Measurement (ESEM). Markham, Ontario, Canada: IEEE, 2017: 314-323.

[178] JETBRAINS. Python developers survey 2019 results[EB/OL]. 2020. https://www.jetbrains.com/lp/python-developers-survey-2019.

[179] LÖWIS M V. Pep 353 – using ssize_t as the index type[EB/OL]. 2005. https://www.python.org/dev/peps/pep-0353.

[180] PILLOW. Fixed deprecation warnings[EB/OL]. 2019. https://github.com/python-pillow/Pillow/pull/3749.

[181] NUMPY. Added missing return after raising error[EB/OL]. 2020. https://github.com/numpy/numpy/pull/16777.

[182] BESSEY A, BLOCK K, CHELF B, et al. A few billion lines of code later: Using static analysis to find bugs in the real world[J]. Communications of the ACM, 2010, 53(2): 66-75.

[183] JUNKER M, HUUCK R, FEHNKER A, et al. Smt-based false positive elimination in static program analysis[C]//Proceedings of the 14th IEEE International Conference on Formal Engineering Methods (ICFEM). Kyoto, Japan: Springer, 2012: 316-331.

[184] COUSOT P. Types as abstract interpretations[C]//Proceedings of the 24th ACM-SIGACT Symposium on Principles of Programming Languages (POPL). 1997: 316-331.

[185] FROMHERZ A, OUADJAOUT A, MINÉ A. Static value analysis of python programs by abstract interpretation[C]//Proceedings of the NASA Formal Methods Symposium (NFM). Newport News, VA, USA: Springer, 2018: 185-202.

[186] SEBASTIANI R. Lazy satisfiability modulo theories[J]. Journal on Satisfiability, Boolean Modeling and Computation, 2007, 3(3-4): 141-224.

[187] PALSBERG J, SCHWARTZBACH M I. Object-oriented type inference[C]//Proceedings of the 6th ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Phoenix, Arizona, USA: ACM, 1991: 146-161.

[188] MAIA E, MOREIRA N, REIS R. A static type inference for python[C]//Proceedings of the 5th Workshop on Dynamic Languages and Applications (DYLA). 2011.

[189] COSTA D, MUJAHID S, ABDALKAREEM R, et al. Breaking type safety in go: An empirical study on the usage of the unsafe package[J]. IEEE Transactions on Software Engineering, 2021, 48(7): 2277-2294.

[190] LEHTOSALO J. Mypy: Optional static typing for python[EB/OL]. 2017. http://mypy-lang.org.

[191] MICROSOFT. Pyright: Static type checker for python[EB/OL]. 2021. https://github.com/microsoft/pyright.

[192] FACEBOOK. Pyre: Performant type checker for python[EB/OL]. 2021. https://github.com/facebook/pyre-check.

[193] GOOGLE. Pytype: Static type analyzer for python code[EB/OL]. 2021. https://github.com/google/pytype.

[194] WANG Y. Pysonar2: An advanced semantic indexer for python[EB/OL]. 2019. https://github.com/yinwang0/pysonar2.

[195] GREEN A, et al. libffi: A portable foreign function interface library[EB/OL]. 2019. http://sourceware.org/libffi/.

[196] WINTER C, LOWNDS T. Pep 3107: Function annotations[EB/OL]. 2006. https://www.python.org/dev/peps/pep-3107/.

[197] PYTHON. Typeshed: Collection of library stubs for python with static types[EB/OL]. 2019. https://github.com/python/typeshed.

[198] FREEMAN T, PFENNING F. Refinement types for ml[C]//Proceedings of the 12th ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI). Toronto, Ontario, Canada: ACM, 1991: 268-277.

[199] MALIK R S, PATRA J, PRADEL M. Nl2type: Inferring javascript function types from natural language information[C]//Proceedings of the 41th International Conference on Software Engineering (ICSE). Montreal, QC, Canada: IEEE, 2019: 304-315.

[200] PEYTON JONES S, VYTINIOTIS D, WEIRICH S, et al. Simple unification-based type inference for gadts[C]//Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP). Portland, Oregon, USA: ACM, 2006: 50-61.

[201] BENDERSKY E. Pycparser: Complete c99 parser in pure python[EB/OL]. 2014. https://github.com/eliben/pycparser.

[202] GCC. Using the gnu compiler collection (gcc): Options to request or suppress warnings [EB/OL]. 2004. https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Warning-Options.html.

[203] NUMPY. ml_flag of nditer.close mismatches its implementation and documentation[EB/OL]. 2021. https://github.com/numpy/numpy/issues/18665.

[204] PILLOW. Calling conventions of some python foreign functions mismatch with their c implementations[EB/OL]. 2021. https://github.com/python-pillow/Pillow/issues/5487.

[205] Rust. Rust foreign function interface.[EB/OL]. 2021. https://doc.rust-lang.org/nomicon/ffi.html.

[206] ORACLE. Java native interface specification.[EB/OL]. 2021. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.

[207] OpenJS. Node.js c++ addons.[EB/OL]. 2021. https://nodejs.org/api/addons.html.

[208] DECAN A, MENS T, CONSTANTINOU E. On the impact of security vulnerabilities in the npm package dependency network[C]//Proceedings of the 15th IEEE Working Conference on Mining Software Repositories (MSR). Gothenburg, Sweden: ACM, 2018: 181-191.

[209] ZIMMERMANN M, STAICU C A, TENNY C, et al. Small world with high risks: A study of security threats in the npm ecosystem[C]//Proceedings of the 28th USENIX Security Symposium (USENIX Security). Santa Clara, CA, USA: USENIX Association, 2019: 995-1010.

[210] HONG S, KWAK T, LEE B, et al. Museum: Debugging real-world multilingual programs using mutation analysis[J]. Information & Software Technology, 2017, 82: 80-95.

[211] RYDER B G. Constructing the call graph of a program[J]. IEEE Transactions on Software Engineering, 1979, 5(3): 216-226.

[212] JAKOB W, et al. Pybind11: Seamless operability between c++ 11 and python[EB/OL]. 2021. https://pybind11.readthedocs.io/en/stable/.

[213] YANG B, WU J, LIU C. Mining data chain graph for fault localization[C]//Proceedings of the 36th IEEE Annual Computer Software and Applications Conference (COMPSACW). Izmir, Turkey: IEEE, 2012: 464-469.

[214] CHEN X, JIANG J, ZHANG W, et al. Fault diagnosis for open source software based on dynamic tracking[C]//Proceedings of the 7th International Conference on Dependable Systems and Their Applications (DSA). Xi'an, China: IEEE, 2020: 263-268.

[215] NIELSEN B B, TORP M T, MØLLER A. Modular call graph construction for security scanning of node.js applications[C]//Proceedings of the 30th International Symposium on Software Testing and Analysis (ISSTA). Virtual, Denmark: ACM, 2021: 29-41.

[216] ALIMADADI S, MESBAH A, PATTABIRAMAN K. Hybrid dom-sensitive change impact analysis for javascript[C]//Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP). Prague, Czech Republic: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015: 321-345.

[217] HANAM Q, MESBAH A, HOLMES R. Aiding code change understanding with semantic change impact analysis[C]//Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME). Cleveland, OH, USA: IEEE, 2019: 202-212.

[218] FELDTHAUS A, SCHÄFER M, SRIDHARAN M, et al. Efficient construction of approximate call graphs for javascript ide services[C]//Proceedings of the 35th International Conference on Software Engineering (ICSE). San Francisco, CA, USA: IEEE, 2013: 752-761.

[219] MADSEN M, LIVSHITS B, FANNING M. Practical static analysis of javascript applications in the presence of frameworks and libraries[C]//Proceedings of the 9th ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE). Saint Petersburg, Russian

Federation: IEEE, 2013: 499-509.

[220] YIN H, et al. js-callgraph: Approximate static call graph for javascript & typescript[EB/OL]. 2019. https://github.com/Persper/js-callgraph.

[221] SALIS V, SOTIROPOULOS T, LOURIDAS P, et al. Pycg: Practical call graph generation in python[C]//Proceedings of the 43rd International Conference on Software Engineering (ICSE). Madrid, Spain: IEEE, 2021: 1646-1657.

[222] HORNER E, et al. Pyan3: Offline call graph generator for python 3[EB/OL]. 2021. https://github.com/davidfraser/pyan.

[223] EADS D, et al. Pycallgraph2: Python call graph[EB/OL]. 2019. https://github.com/daneads/pycallgraph2.

[224] FABRY O, et al. go-callvis: A development tool to help visualize call graph of a go program using interactive view[EB/OL]. 2021. https://github.com/ofabry/go-callvis.

[225] ROGOWSKI S, et al. Code2flow: Call graphs for dynamic languages[EB/OL]. 2021. https://github.com/scottrogowski/code2flow.

[226] ARZT S, KUSSMAUL T, BODDEN E. Towards cross-platform cross-language analysis with soot[C]//Proceedings of the 5th ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP). Santa Barbara, CA, USA: ACM, 2016: 1-6.

[227] VALLÉE-RAI R, CO P, GAGNON E, et al. Soot: A java bytecode optimization framework [M]//CASCON First Decade High Impact Papers. 2010: 214-224.

[228] FOURTOUNIS G, TRIANTAFYLLOU L, SMARAGDAKIS Y. Identifying java calls in native code via binary scanning[C]//Proceedings of the 29th International Symposium on Software Testing and Analysis (ISSTA). Virtual Event, USA: ACM, 2020: 388-400.

[229] CHEN T, et al. Ffi navigator[EB/OL]. 2021. https://github.com/tqchen/ffi-navigator.

[230] PYTORCH. torch.kthvalue returns random value when the k is invalid[EB/OL]. 2021. https://github.com/pytorch/pytorch/issues/68813.

[231] MIN K, CORSO J J. Adversarial background-aware loss for weakly-supervised temporal activity localization[C]//Proceedings of the 16th European Conference on Computer Vision (ECCV). Glasgow, UK: Springer, 2020: 283-299.

[232] YANG G, DUAN T, HU J E, et al. Randomized smoothing of all shapes and sizes[C]// Proceedings of the 37th International Conference on Machine Learning (ICML). Virtual: PMLR, 2020: 10693-10705.

[233] RYMARCZYK D, STRUSKI Ł, TABOR J, et al. Protopshare: Prototypical parts sharing for similarity discovery in interpretable image classification[C]//Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD). Singapore: ACM, 2021: 1420-1430.

[234] MICROSOFT. Pylance: Fast, feature-rich language support for python[EB/OL]. 2021. https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance.

[235] MICROSOFT. Python extension for visual studio code[EB/OL]. 2021. https://marketplace.visualstudio.com/items?itemName=ms-python.python.

[236] MICROSOFT. Javascript and typescript nightly[EB/OL]. 2021. https://marketplace.visualstudio.com/items?itemName=ms-vscode.vscode-typescript-next.

[237] TALIN. Pep 3102 – keyword-only arguments[EB/OL]. 2006. https://www.python.org/dev/peps/pep-3102/.

[238] GIANOLIO M, et al. Node.js c++ bindings for cblas[EB/OL]. 2021. https://github.com/nperf/nblas.

[239] JETBRAINS. C programming - the state of developer ecomsystem in 2022[EB/OL]. 2022. https://www.jetbrains.com/lp/devecosystem-2022/c/.

[240] MICROSOFT. Vscode language server extension[EB/OL]. 2021. https://code.visualstudio.com/api/language-extensions/overview.

[241] BOGAR A M, LYONS D, BAIRD D. Lightweight call-graph construction for multilingual software analysis[R]. Fordham University, 2018.

[242] CHEN T, LI M, LI Y, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems[A]. 2015. arXiv: 1512.01274.

[243] CHEN T, GUESTRIN C. Xgboost: A scalable tree boosting system[C]//Proceedings of the 22nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD). San Francisco, CA, USA: ACM, 2016: 785-794.

[244] CHEN T, MOREAU T, JIANG Z, et al. Tvm: An automated end-to-end optimizing compiler for deep learning[C]//Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Carlsbad, CA, USA: USENIX Association, 2018: 578-594.

[245] CHEN L. Deep learning and practice with mindspore[M]. Springer Nature, 2021.

[246] MICROSOFT. Vscode language server extension[EB/OL]. 2021. https://code.visualstudio.com/api.

[247] MATTHEWS J. Equation-preserving multi-language systems[R]. University of Chicago, 2008.

致　谢

感谢研究生期间给予指导和帮助的老师们，包括但不限于中国科学技术大学张昱教授、熊焰教授、黄文超教授、薛吟兴研究员、华蓓教授，中国科学院软件研究所张健研究员，等等。

感谢所有实验室同门的陪伴、帮助和合作，包括但不限于郭兴、张宇翔、龚磊、黄奕桐、丁伯尧、刘硕、赵琦等等。

感谢父母和所有亲人给予的支持、理解和帮助。感谢朋友们的陪伴与帮助，包括但不限于曹子闻、濮维凡、陈星豪、张润程、何甜甜等等。

在读期间发表的学术论文与取得的研究成果

已发表论文

1. **Mingzhe Hu**, Qi Zhao, Yu Zhang, Yan Xiong. Cross-Language Call Graph Construction Supporting Different Host Languages[C]//*Proceedings of the 30th IEEE International Conference on Software Analysis Evolution and Reengineering* (SANER), 2023: 156–166.（CCF B 类会议，EI 检索号：20232214171915）
2. **Mingzhe Hu**, Yu Zhang. An Empirical Study of the Python/C API on Evolution and Bug Patterns[J]. *Journal of Software: Evolution and Process* (JSEP), 2023, 35(2): e2507.（CCF B 类期刊，EI 检索号：20223712742159）
3. **Mingzhe Hu**, Yu Zhang, Wenchao Huang, Yan Xiong. Static Type Inference for Foreign Functions of Python[C]//*Proceedings of the 32nd International Symposium on Software Reliability Engineering* (ISSRE), 2021: 423–433.（CCF B 类会议，EI 检索号：20221211807801）
4. **Mingzhe Hu**, Yu Zhang. The Python/C API: Evolution, Usage Statistics, and Bug Patterns[C]//*Proceedings of the 27th IEEE International Conference on Software Analysis Evolution and Reengineering* (SANER ERA), 2020: 532–536.（CCF B 类会议短文，EI 检索号：20201708531247）
5. Boyao Ding, Yu Zhang, Jinbao Chen, **Mingzhe Hu**, Qingwei Li. CGORewritter: A better way to use C library in Go[C]//*Proceedings of the 30th IEEE International Conference on Software Analysis Evolution and Reengineering* (SANER ERA), 2023: 688–692.（CCF B 类会议短文，EI 检索号：20232214171987）
6. Yun Peng, Yu Zhang, **Mingzhe Hu**. An Empirical Study for Common Language Features Used in Python Projects[C]//*Proceedings of the 28th IEEE International Conference on Software Analysis Evolution and Reengineering* (SANER), 2021: 24–35.（CCF B 类会议，EI 检索号：20212210433677）

发明专利申请

1. 张昱, **胡明哲**. 一种 Python 外部函数的静态类型推断方法及系统 [P]. 申请号: CN202111105813.3A, 申请日: 20210922, 公开号: CN113885854A, 公开日: 20220104.
2. 张昱, 彭昀, **胡明哲**. 一种 Python 语言特征自动识别系统和方法 [P]. 专利号: ZL202010663123.9, 申请日: 20200710, 公开日: 20201030, 授权公告日:

20220111, 公开号: CN111858322B, 证书号: 4889779.

参与项目

1. 国家自然科学基金面上项目，62272434，多语言软件分析与优化研究，2023/01–2026/12，项目负责人：张昱
2. 国家自然科学基金面上项目，61772487，移动和云平台软件的系统资源使用行为分析与改进，2018/01–2021/12，项目负责人：张昱
3. 安徽省自然科学基金面上项目，1808085MF198，移动和云平台软件的系统资源使用行为分析，2018/07–2021/06，项目负责人：张昱