通过摘要特化统一多语言互操作的指针分析

Jyoti Prakash
prakash@fim.uni-passau.de
帕绍大学，帕绍，德国

Abhishek Tiwari
tiwari@fim.uni-passau.de
帕绍大学，帕绍，德国

Christian Hammer
hammer@fim.uni-passau.de
帕绍大学，帕绍，德国

arXiv:2305.03916v1 [cs.SE] 6 May 2023

摘要

多语言应用的模块化分析颇具挑战，因为必须解析跨越语言边界的堆对象流。当前最先进的多语言应用分析存在两个根本局限。首先，这些分析假定宿主语言与客体语言之间存在显式边界，以确定跨语言数据流。其次，它们依赖于针对宿主语言和客体语言的特定分析。前一假设在近期多语言编程技术不断发展的背景下并不切实际，而后一局限则忽视了底层语言指针分析方面的进展。在本研究中，我们提出一种新颖的摘要特化技术来扩展现有指针分析，从而统一跨语言边界的指向集。我们提出的技术只需进行少量修改，即可利用宿主分析与客体分析的各种组合。我们通过在两种多语言模型上进行评估，展示了该方法的有效性和通用性：通过 Android NDK 实现的 Java-C 通信，以及 GraalVM 中的 Java-Python 通信。

ACM 参考格式：
Jyoti Prakash、Abhishek Tiwari 和 Christian Hammer。2023年。通过摘要特化统一多语言互操作的指针分析。载于 *ACM* 会议论文集（*Conference'17*）。ACM，纽约，美国纽约州，12页。
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 引言

开发者正迅速采用多语言编程模型。近年来，GraalVM 等运行时系统取得进展，加之编程语言和编程模型领域的创新，加速了多语言范式模型的普及 [6, 7, 12, 20, 25, 28, 31, 33, 37, 42, 52]。毫无疑问，多语言编程模型往往能为开发者带来益处。他们可以将以任意语言编写的现有模块复用或重新打包到新应用中。不幸的是，滥用的可能性也随之增加：由于对语言互操作性的了解有限，开发者可能引入重大缺陷，甚至安全漏洞 [25, 28, 37]。因此，开发能够促进多语言开发的工具和实践已至关重要。

允许为个人或课堂使用而免费制作本作品全部或部分内容的数字副本或纸质副本，前提是这些副本不得用于营利或商业利益，且每份副本的首页均须保留本声明及完整引文信息。对于本作品中归他人而非 ACM 所有的组成部分，必须尊重其版权。注明出处后可以摘录。若要以其他方式复制、再版、发布到服务器或重新分发到列表，必须事先获得具体许可和/或支付费用。请向 permissions@acm.org 申请许可。*Conference'17*，*2017*年7月，美国华盛顿特区© 2023年 计算机协会。ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

静态程序分析有助于开发此类工具。一种模块化技术颇具前景：分别研究每个语言模块，再将研究结果结合起来进行全程序分析，从而兼顾可扩展性和精度 [17]。近年来，模块化程序分析研究日益兴盛，针对流行语言开发了大量新技术 [40, 43]。然而，多语言应用的模块化分析仍处于起步阶段。跨语言分析的模块化方法可以利用语言的模型和优化 [32]。不过，实现这一目标仍面临诸多障碍。

以模块化方式整合多个语言模型的分析结果，是主要障碍之一。现有方法，例如 Lee 等人 [32]会将客体语言操作的数据流摘要转换为宿主语言。将这些摘要注入宿主程序后，再对该程序进行分析。然而，这种方法有两个主要缺点。首先，由于现代语言的语法和语义十分复杂，将客体语言所有可能的影响转换为宿主语言，是一项复杂且容易出错的工作。这需要对客体语言的每个方面进行建模，并将其转换为宿主语言的语义。因此，Lee等人 [32]只将客体语言语义的一个子集转换为宿主语言。其次，转换后的程序使用宿主语言模型进行分析，因此无法利用客体语言的独特特性，也就无法针对客语言模块实现语言特定的优化，以获得最合适的分析结果。

现有技术的另一个缺点是依赖明确的宿主语言与客语言边界 [39]。这些边界通过明确的语法标记来识别，通常是参与跨语言通信的预定义函数调用、签名或数据结构。静态分析可以检测这些预定义的语法结构，并定义能够在宿主语言中复现其行为的模型。然而，当前的多语言编程环境，例如 *GraalVM* 或 Android 的*Web-View*，允许混合编写两种语言的代码，使这些明确的边界变得模糊。例如，WebView 也允许客体向宿主发送消息，这些消息并不依赖明确的语法标记，而是通过共享对象进行通信。先前的研究表明，这些功能在当代应用程序中十分普遍 [53]。此外，未来的多语言编程环境不可避免地会采用类似模式，因此完全依赖静态语言边界并不现实。

我们的方法。 我们提出了一种新策略来应对这些挑战。我们的模块化技术不受特定语言模型的限制：它可以灵活组合不同的宿主分析和客体分析，以充分利用特定语言模型，

$m \in$ **宿主方法 :=** $m_1 \mid m_2 \mid m_3 \mid \cdots \mid m_n i \in$ **InterfaceVars :=** $i_1 \mid \cdots \mid i_n v \in$ **变量 :=** $v_1 \mid v_2 \mid v_3 \mid \cdots \mid v_n \mid$ **InterfaceVars** $g \in$ **字段 :=** $g_1 \mid g_2 \mid g_3 \mid \cdots \mid g_n s \in$ **语句 :=** $v =$ **新建** $T \mid i = T \mid v = v.m \mid v = v.g \mid v$ **() 新建 () ' () ' '** $.g = v \mid$ **如果 (** e **) {** s **}[否则 {** s **}]** $v =$ **eval(** $program$ **)** $\mid$ **op** $s_1\ s_2 \mid s_1; s_2 \mid$ **当 (** e **) {** s **} 宿主语言** $f \in$ **客体方法 :=** $f_1 \mid f_2 \mid f_3 \mid \cdots \mid f_n w \in$ **变量 :=** $v_1 \mid v_2 \mid v_3 \mid \cdots \mid v_n \mid$ **InterfaceVars** $h \in$ $= h_1 \mid h_2 \mid h_3 \mid \quad \mid h_n$ **字段 : · · · 语句 :: 新建** $t \in$ $= v = T$ **()** $\mid w \quad \mid' = w.f$ **() ' () ' 如果 (** $w = i.m \mid w = w.h \mid e \quad t\ [\quad t\ ]$ **) { } 否则 { } ' op ; 当 ( )** $w\ .h = w \mid \quad t_1\ t_2 \mid t_1\ t_2 \mid \quad e \quad t$ **{ } 客体语言**

图 1：宿主语言与客体语言模型

从而继承底层分析的精度、可扩展性和可靠性改进。值得注意的是，我们的技术扩展了现有的宿主语言与客体语言分析，使其能够包含有关双方调用流和数据流的关键信息。给定一个多语言模型，我们的分析能够找出客体可以使用的宿主通信端点和特性（共享类/函数）。利用这些信息，通过将宿主和客体各自调用图中缺失的调用流边连接起来，便可创建一个跨语言调用图。接下来，我们的技术为跨语言调用图中发现的所有互操作函数构建过程内函数摘要。根据*Andersen* 风格 [48] 约束方案，使用过程内指向分析确定这些函数摘要。随后，对这些摘要进行解析，使跨语言共享变量的指向集包含正确且完整的引用。为此，我们提出了创新的摘要特化和统一技术。统一是一种别名感知技术，用于统一宿主与客体之间变量的必别名。然后，将解析后的摘要转换为与宿主和客体的分析模型保持一致的形式。借助已解析的调用边和统一后的指向集，通过独立合并宿主与客体的分析结果，整体执行分析。

我们将我们的方法实现为名为 *Conflux* 的工具。为评估我们的方法的有效性和通用性，我们将*Conflux* 应用于两种多语言模型：1）通过 Android NDK [1]进行的 Java-C 通信，以及 2）GraalVM [2] 中的 Java-Python 通信。*Conflux* 优于此前的当前最先进水平，在使用 Android NDK 进行 Java-C 通信的 39 个应用程序中，成功分析了 31 个。此外，*Conflux* 使用 GraalVM API 成功分析了三个 Java-Python 应用程序中的两个。据我们所知，*Conflux* 是首个可应用于多种多语言模型的多语言分析工具。

总而言之，本文作出了以下贡献：(i) 采用摘要特化进行模块化分析。我们提出了一种新颖的模块化分析框架，用于分析多种多语言语言模型。我们的框架可用于宿主与客体分析的各种组合，以充分利用特定语言模型。

清单 1：宿主语言代码

```
foo() {
  i = 新建 BridgeClass();      // obj@iface
  eval("设置()");     // invoke the 客体代码
  // do some 计算
  x = eval("getAndLeak()");
  返回 x;
}
BridgeClass.setSecret(secret)          {
    此.secret =秘密;
}
BridgeClass.getSecret(){
    返回 this.secret;
}
```

清单 2：访客语言示例

```
set(){16v =新建Secret();//
ob
19
r
```

(ii) 多语言分析工具。 我们将我们的方法实现为一个名为 *Conflux* 的工具。*Conflux* 支持两种广泛使用的多语言模型，并且可以轻松扩展以分析其他多语言模型。*Conflux* 已公开提供 [8]，可用于实验和扩展。

（iii）广泛评估。 我们在多个应用程序上评估*Conflux* 来自两种多语言模型的应用程序，并展示其有效性和普遍适用性。

2 模块化跨语言分析

一种核心多语言模型。 现有多种多语言模型，例如 Java-C、Java-Javascript、Java-python、Javascript-wasm、Java-Javascript-wasm，以及不同的通信设置。本文提出首个能够处理多种多语言模型的模块化分析框架。为此，我们定义了一种语言模型，它从通用多语言模型的语义中充分抽象出来，以便于介绍和解释我们的方法。我们的语言模型如图 1 所示，遵循外部函数接口（FFI）模型，涵盖多语言模型中宿主到客体以及客体到宿主通信所需的语义[1]。该语言模型基于对现有多语言编程环境的深入考察，特别是 GraalVM 和 Android 混合应用。宿主语言和客体语言被定义为标准的命令式/面向对象程序，并加入语言间通信构造。为专注于分析，我们省略了类和字段语法等标准细节。

宿主与客体语言中的函数、变量、字段变量和语句都是等价的。接口变量（*InterfaceVars*）和 *eval* 是使宿主与客体能够相互通信的两个关键特性。宿主语言构造 *eval* 执行客体语言程序或表达式，并将相应结果返回给宿主。

[1]其他潜在的通信机制，例如基于中间件的远程方法调用，已在先前的研究中有所涵盖（第 6 节），因此不属于本方法的讨论范围。

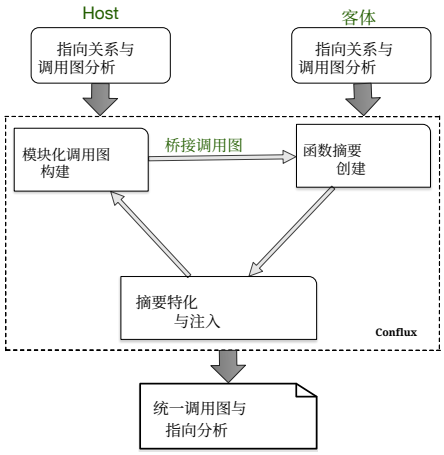


图2：分析概述

宿主语言定义 *InterfaceVars*，供客体访问宿主功能。InterfaceVars 是客体可以向其发送消息的宿主对象。从技术上讲，它们是充当宿主类与客体之间桥梁的普通对象。允许客体访问宿主语言特性子集的多语言环境，是这一前提的一个具体实例。清单 1 和清单 2 使用我们的语言构造来示例说明一个简化的多语言模型。第 2 行声明了一个宿主类 *BridgeClass* 的接口变量 i，使客体语言能够访问其方法。在第 3 行和第 5 行，宿主使用 *eval* 方法调用客体语言代码。相应地，在客体语言的第 17 行和第 20 行，可以访问 *BridgeClass* 的 *setSecret* 和 *getSecret* 函数。

分析概述。 图 2 展示了基于前述语言构造的分析组件。该框架建立在面向宿主语言和客体语言的可扩展上下文敏感预分析之上。随后，它会计算分析的其他构成模块：调用图（通过模块化分析构建，见第 2.2 节）和过程内函数摘要（见第 2.3.1 节）。接下来，我们将详细介绍自底向上分析策略。首先，我们通过图示说明针对单个方法内发起的桥接调用所进行的分析（第 2.1 节至第 2.3.3 节）。然后，我们扩展我们的方法，以便对多个方法中的桥接函数调用进行建模（第 2.3.5 节）。

2.1 预分析

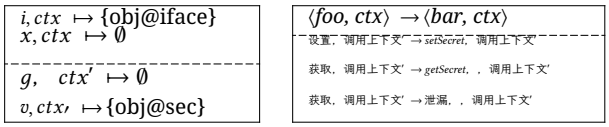
在我们的方法中，预分析 模块利用分别针对宿主语言和客体语言（即不考虑外部函数接口）独立计算的指向分析和调用图分析。该模块能够灵活地设定各语言底层分析的敏感度级别。因此，它可以采用针对特定语言、精度最高且可扩展的分析。接下来，我们将介绍预分析模型，并以清单 1 和 2 中的示例为例进行说明。

预分析如图 3 所示。它包括一个变量集合 V、一个堆对象集合 H，以及一个上下文集合 C。C 可以通过多种概念进行建模：例如，将对象用作上下文（对象敏感），通过调用函数调用点（调用点敏感），以及其他许多方式。上下文集合 C 通过移除

V：变量集合　　C：上下文集合

H：堆对象集合　　$^{\text{调用上下文}}{}_{\prime} \subseteq c_1^j \times c_2^j \times \ldots \times c_n^j,\ c_i^j \in C$

图3：预分析模型



（a）指向分析　　**（b）调用图分析**

图 4：宿主与客体语言的预分析

不可实现路径（例如来自不匹配函数调用的路径）。上下文 调用上下文 是一个由 C中的元素构成的字符串。为便于说明，后文中我们可能会省略上下文。

指向分析会生成一个从变量到堆对象集合的映射，称为变量指向集（在调用上下文下）。变量指向映射*VarPointsTo*：$V \mapsto \mathrm{P}(H)$，是从变量集合到堆对象的映射。变量 $v \in V$ 的指向集是堆对象集合 $h \subseteq H$。在编程语言中存在字段时，需要进一步解析，将对象和字段对映射到堆对象集合，这称为字段指向集。如果 f 是堆对象 o_i 的一个字段，那么它的指向集就是堆对象集合 O若，即通过堆对象 o_i 的字段 f 引用的堆对象。字段指向映射*FldPointsTo*：$H \times$ 字段 $\mapsto \mathrm{P}(H)$ 将每个堆对象和字段对映射到一个堆对象集合。

在清单 1 和 2 的示例中，宿主语言的预分析推断出，在某些上下文中，i和 v分别引用 obj@iface 和 obj@sec（图 4a）。由于 x捕获了外部函数接口调用的返回值，分析会忽略此操作。同样，在客体语言中，g捕获了外部函数接口调用的值，该操作也会被忽略。正如预期，预分析结果并不完整，遗漏了关键的数据流。清单 1 和 2 表明，方法 set 和 get 可从 foo 到达。第 2.2 节通过构建跨语言调用图，扩展预分析以解析这些边。此外，第 16 行定义的客体对象 obj@sec 通过第 17 行的桥接调用被写入 BridgeClass 的字段 secret。第 20 行从该字段读取值到 g中，并在末尾将其返回给 foo 中的变量 x。因此，g和 x应当引用 obj@sec。我们通过模块化调用图构建和摘要特化方法（第 2.3.3 节）解析这些数据流。

2.2 模块化调用图构建

模块化调用图构建确定了通过*eval* 和接口变量发生的跨语言函数调用的边。它基于预分析阶段获得的不完整调用图构建。该过程从不完整的宿主调用图开始，并将所有桥接类方法纳入其中；我们的分析假设所有桥接类方法都可以从客体语言调用。最初，扩展后的宿主调用图（G_{host}）包含可能不连通的节点（即使宿主代码没有调用桥接类的方法，我们也会将这些方法纳入其中）。接下来的目标是补全宿主和客体调用图（G_{guest}）中缺失的边，即添加从宿主到客体（通过*eval*）以及从客体到宿主（通过接口变量）的相应边。

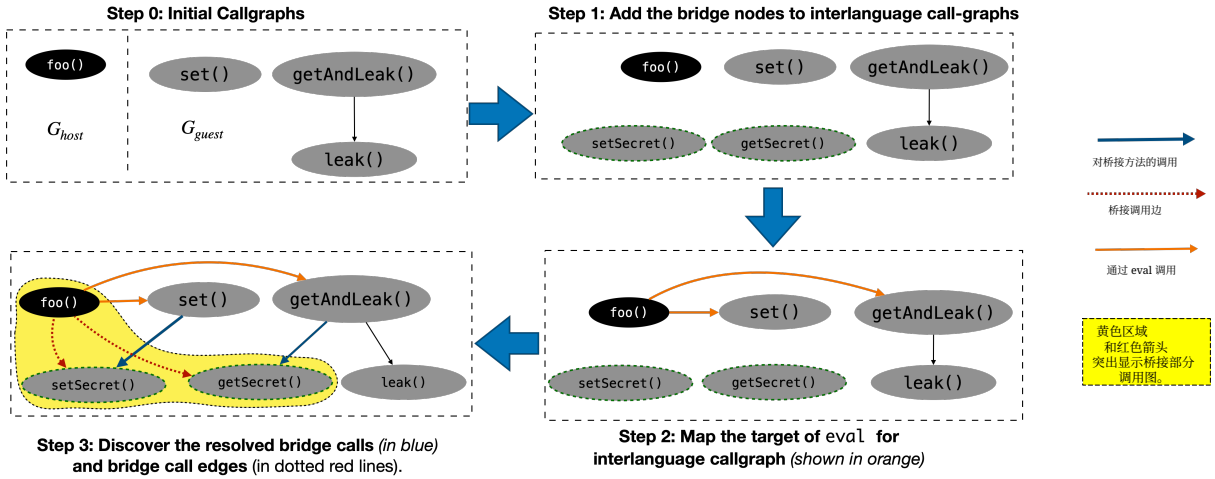


图5：清单 1 和清单 2 中程序的调用图

算法 1： 模块化调用图分析

输入： 宿主与访客调用图 G_{host}, G_{guest}
输出： 跨语言调用图 G
输出： 桥接调用图集合 B_{m_h}， $m_h \in G_{host}$

1 $G \leftarrow G_{host} \cup G_{guest}$
2 **对每个 eval**(m_g) $\in m_h$ 其中
3 $m_g \in G_{guest}$ // 客户方法
4 $m_h \in G_{host}$ /* 宿主方法 */ **执行**
5 扩展 G，使其包含一条边 (m_h， m_g)
6 **如果** v ∈接口变量（m_h）// v 在 m_h 中定义
并且 v ∈使用（m_g）// v 在m_g中的某个表达式中使用
then
7 $G, B_{m_h} \leftarrow$ 发现桥接调用（v, m_h, m_g）
8 **end**
9 **结束**

算法 1 描述了模块化调用图的生成过程。首先，我们合并相互独立的宿主和客体调用图。接下来，识别宿主方法中通过*eval* 调用的客体方法（第 2 行），并添加一条从宿主方法指向该客体方法的边（第 5 行）。客体语言也可以通过接口对象访问宿主方法。此类流在第 6 行添加（算法 2 将对此进行说明）。核心思路是，在客体方法中搜索通过接口变量对桥接宿主方法的调用。如果存在此类调用，就在相应的客体方法和宿主方法之间添加边，同时添加从原始宿主方法指向桥接宿主方法的边。后者称为桥接调用边。它们表示在客体语言中通过桥接对象间接调用宿主语言方法的关系。

定义1（桥接调用边）。设 i 是从宿主方法 $m_h \in G_{host}$共享的接口变量， $m_g \in G_{guest}$ 使得m_g通过接口变量 i （或其别名）调用桥接方法 $m_b \in G_{host}$。则桥接调用边（记为 E_{bc}）是跨语言调用图 G中从 m_h 到 m_b 的传递调用边。

算法 2： 发现桥接调用

输入： 接口变量 v、宿主语言方法 m_h、客体语言方法m_g
输出： 跨语言调用图 G
输出： 桥接调用图 B_{m_h}

1 **对于每个** 调用 $x.m()$ 在^{m_g} 其中 $x \in MayAlias(v)$
2 并且 $m \in M_b$/* M_b 是桥接类方法的集合 */ **执行**
3 将边 $(^{m_g}, m)$ 添加到 G
4 添加桥接调用边 (m_h, m) 到 B_{m_h}
5 **结束**
6 //别名作为参数传递给另一个客体函数
7 **对每个**调用 $m\prime g(*, x, *)$ 在 ^{m_g} 中，其中 $x \in MayAlias(v)$
do
8 发现桥接调用$(x, m_h, m\prime g)$
9 **结束**

图5展示了为清单 1中的示例创建跨语言调用图时涉及的转换。步骤0描述了在预分析阶段获得的调用图。最初， G_{host} 从*foo* 出发没有任何边，因为预分析不会关注外部函数调用产生的边。宿主调用图经过扩展后纳入桥接方法（步骤1）。随后，调用图进一步扩展，以纳入通过eval间接调用客体方法的边（步骤2）。最后，调用图扩展以纳入桥接调用边（步骤3）；*setSecret* 和*getSecret* 通过接口变量*i*调用。因此，桥接调用边（*foo*， *setSecret*）和（*foo, getSecret*）被添加到 G中。桥接调用边用于确定我们的分析中进行摘要特化和统一所需的方法。我们在定义2中将桥接调用图定义为由桥接边和节点构成的有向图。

定义2（桥接调用图）。桥接调用图$B_{m_h}(N, E_{bc})$是一个连通的语言内图，其中 E_{bc} 包含所有起源于 $m_h \in N$，的桥接调用边。

赋值
$$\frac{x = y}{[y] \subseteq [x]}$$

Load
$$\frac{v = x.f}{[x.f] \subseteq [v]}$$

存储
$$\frac{\text{x.f} = \text{v}}{[v] \subseteq [x.f]}$$

New
$$\frac{\text{v} = \text{新建 Obj()} \quad pts(v) = \text{P}(v, \text{调用上下文}) \quad \text{调用上下文} = CallContext(v)}{\forall o_i \in pts(v): \{o_i\} \subseteq [v]}$$

Func
$$\frac{\text{x} = \text{函数(参数)} \quad pts(x) = \text{P}(x, \text{调用上下文}) \quad \text{调用上下文} = CallContext(x)}{\forall o_i \in pts(x): \{o_i\} \subseteq [x]}$$

图 6：过程内函数摘要的约束系统（C_s）

图 5 中阴影部分（步骤 4）展示了清单 1 和清单 2 示例中起源于 *foo()* 的桥接调用图。从 *foo* 到 *setSecret* 和 *getSecret* 的间接调用经由桥接对象 *i*完成。桥接调用图是摘要特化、统一和注入中使用的核心数据结构。

2.3 模块化分析框架

现在我们介绍分析的核心支柱——摘要特化。摘要特化用于解析桥接调用图中从客体到宿主以及反向发生的数据流。该过程分为三个步骤。首先，它利用预分析期间获得的信息计算函数摘要。函数摘要通过过程内指向分析确定。第二步，将这些过程内摘要合并并传播到参与桥接调用图的函数中，以解析通过桥接发生的数据流。在此阶段，分析已识别出通过接口变量从客体流向宿主的数据流。第三步，将这些已解析的摘要注入相应的客体方法中。将摘要插入桥接方法后，分析便自然转向全程序分析。这些技术详见 SpecializeSummary、Unification 和 InjectSummary。

2.3.1 函数摘要。 我们分析中的函数摘要通过轻量级过程内指针分析获得。函数摘要的形式为 $\text{AP} \mapsto H \cup \emptyset$，其中$H$表示堆对象的集合，AP 表示访问路径的集合，而 $\emptyset$ 表示相关摘要尚未解析。访问路径由基变量（通常是局部变量）后接一系列字段访问组成（记为 x.f.g.h）。我们选择访问路径表示法，因为它天然能够建模字段敏感性 [34] 且有界，因此适用于数据流分析 [27]。

该分析基于一种直接明了的基于约束的方法，其中的约束通过预分析得出的已解析值加以强化。我们使用 Andersen 风格分析（即基于包含关系的分析）中的约束来创建函数摘要 [48]。事实证明，这些约束具有较高的精度，并已成为支持堆操作的编程语言中指针分析的事实标准表示 [47]。一般而言，基于包含关系的约束由于在变量数量上的时间复杂度为立方、空间复杂度为平方而面临可扩展性问题，

算法 3： SpecializeSummary

输入： 桥接调用图 $B_{m_h}(N' E_{bc})$
输入： 客体预分析 P_g
输出： 函数摘要约束 F

```
对每个 函数节点 N_i ∈ N' 执行
    ▷ 根据图6中的约束系统生成约束
    C_i^N ← 生成约束(N_i, C_s)
    对于每个 客体函数节点 N_g 调用 N_i(q̄) (调用桥接方法) /* q̄ 是形式参数列表 */ 执行
        设 N_i(t̄) 为 N_g 中的函数调用// t̄ is the 实际参数列表
        将 P_g(t_j) ⊆ [q_j] 添加到 C_i^N，其中 t_j ∈ t̄, q_j ∈ q̄
    end
    遍历 c_i ∈ C_i^N 执行
        如果 c_i 包含 this.fld 则
            将 c_i 中的 this 替换为接口变量 b
        end
    end
    F_i^N = 立方求解器(C_Ni)
    F = F ∪ F_i^N
结束
统一摘要（F, B_{m_h}）
```

但此阶段的函数摘要是在过程内计算的，因此涉及的变量数量较少。

图6展示了生成函数摘要的约束系统。我们使用符号 $[var]$ 表示var的约束变量，P 表示预分析，而 $pts(var)$ 表示变量 var 的指向集。赋值、存储和加载是标准的 Andersen 风格约束。新建规则会查询预分析中该变量的指向集，并针对每个堆对象构造一条约束。对于函数调用，Func 会直接从预分析中获取结果变量的指向集（用于捕获返回值）；预分析中已包含过程间分析。随后，将收集到的约束解析为函数摘要。算法 3 展示了获取函数摘要约束的过程。给定一个桥接调用图和预分析，它首先为桥接调用图中的函数节点生成函数摘要约束。请注意，除根节点外，桥接调用图中的函数节点由宿主的（桥接）函数构成，这些函数为客体提供功能。获得这些函数的约束后，下一步是将其形式参数映射到客体中的实际调用参数。图7展示了从第 4 行到第 12 行的处理过程。随后，这些约束会被送入立方求解器，由其生成实际的函数摘要约束。

图 8 展示了为我们的示例程序生成的约束和函数摘要。第二列展示约束，第三列展示经过立方求解器处理后解析得到的摘要。在函数 foo 中，我们有一个约束 $\{obj@iface\} \subseteq [i]$，该约束被解析为 $[i] \mapsto \{obj@iface\}$。在函数 setValue 中，我们有两个约束 $\{obj@sec\} \in [secret]$ 和 $[secret] \subseteq$

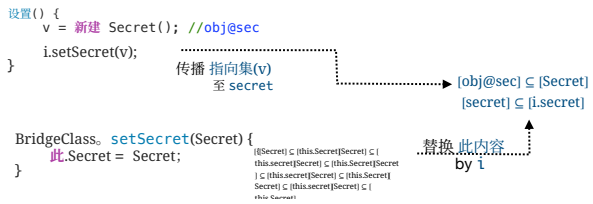


图 7：setSecret 的函数摘要约束

函数	约束	函数摘要
foo	{对象@接口} ∈ [i] ∅ ⊆ [x]	[i] ↦ {obj@iface} [x] ↦ ∅
setSecret	[Secret] ⊆ [i.Secret] {obj@sec} ∈ [secret]	[i.secret] ↦ {obj@sec}
getSecret	[i.secret] ⊆ [$ret]	[i.Secret] ↦ ∅ [$返回值] ↦ ∅

Figu图 8：摘要约束与已解析的摘要 **aries**

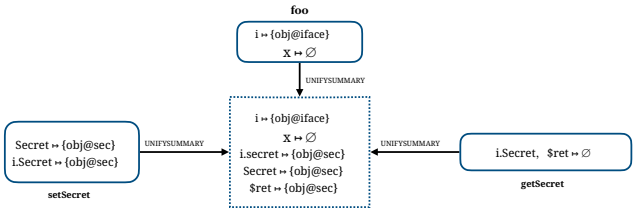


图 9：图 5 中桥接调用图的统一

[*i.secret*]；它们被解析为 [*i.secret*] ↦ {*obj@sec*}。在函数 getValue 中，我们有一个约束 [*i.secret*] ⊆ [$*ret*]，解析后得到 [$*ret*] ↦ ∅ 和 [*i.secret*] ↦ ∅。

2.3.2 统一。如图 8 所示，在 setSecret 函数中，i.secret 对应于客体对象 *obj@sec*；然而，在 getSecret 函数中，i.secret 仍然未知，尽管基于相同接口变量 *i*，访问路径完全相同。为了解决这一问题，我们根据访问路径合并这些摘要。

具有相同访问路径的两个摘要通过并集操作合并，即如果 $ap \mapsto \phi_1$ 和 $ap \mapsto \phi_2$，则统一结果为 $ap \mapsto \phi_1 \cup \phi_2$。统一过程也具有别名感知能力，即 ap 中基变量的别名也会被统一。为简单起见，我们假设访问路径不嵌套。这一假设是合理的，因为大多数数据流分析都在中间语言上进行，而中间语言将语言简化为便于分析的语法。图 9 展示了图 5 所示桥接调用图的统一结果。此步骤根据访问路径及其别名访问路径统一摘要。首先，分析通过比较访问路径解析桥接函数间的数据流，然后更新相应的指向集。因此，在参与桥接调用图的所有函数中，*i.secret*都指向客体对象 *obj@sec*。

2.3.3 摘要注入。经过摘要统一后，分析已解析桥接调用图中的跨桥接数据流。具体而言，分析已解析桥接方法中的操作，即从客体接口调用的宿主语言函数。还剩两项任务：第一，将分析摘要转换为与宿主语言和客体语言所采用的分析模型兼容的形式。第二，将转换后的摘要传播到客体侧的调用函数。

通过统一获得的摘要采用访问路径格式；然而，为了在宿主和客体中复用这些摘要，我们需要将其转换为宿主与客体语言的分析模型。值得注意的是，我们会将访问路径格式转换为它们各自的原生格式。底层分析涉及的两个敏感性层面——调用上下文和字段敏感性——尤为重要。由于桥接函数是通过客体应用程序中的接口变量调用的，因此宿主方法的上下文会被赋予桥接调用图。随后，我们根据底层分析处理字段操作的方式，将字段敏感性划分为字段敏感、基于字段或字段不敏感。分析会根据底层分析的类型，确定适当的字段敏感性表示形式。

以运行示例为例，分析得出 getSecret 函数返回 obj@sec。返回值由 getAndLeak中的变量 g 捕获；然而，在此阶段，g 尚未解析，仍指向 ∅。因此，分析未能发现 g 在第 21 行暴露了敏感值。摘要注入通过将统一后的摘要补入预分析，完成这一步。首先，将统一摘要信息转换为宿主语言的分析模型。预分析会更新受注入数据流影响的变量，从而触发增量更新。接下来，这些更新（转换后的摘要）会传播至受影响的客体函数。

算法4展示了摘要注入的步骤。对于给定的函数 f，它以 f 的分析快照作为输入。我们将此快照称为函数的投影 。形式上，我们将投影 定义为：

定义3（投影）。函数 f 的投影 $\mathcal{R}^f = \bigcup_{v \in def(f)} \mid h = pointsTo(v, ctxt)\}$ 是在调用上下文 $ctxt$ 下，函数 f 中定义的变量对应的指向集合。

对于桥接图中的函数 f，我们的分析获取该函数的上下文，并生成分析结果的一个投影。直观地说，对于上下文 *ctxt*，投影就是函数中定义的变量的指向集。算法4使用分析结果的投影来注入通过统一获得的数据流事实。除了投影之外，该算法还接受采用访问路径格式表示的函数 f 的函数摘要，以及函数处理字段操作的方式信息——即 FieldSensitive、Field-Based 或 FieldInsensitive。早期研究 [13, 21, 24, 38, 45, 48] 表明，这些是处理字段操作的主要技术，因此我们将算法限定于这些技术。第3行至第19行将以访问路径表示的所获摘要信息转换为分析所支持的表示形式。在第5行，将简单变量的翻译后指向集添加到投影中。第9行纳入字段敏感分析的翻译后指向集，第12行纳入基于字段变量的翻译后指向集，第15行纳入字段不敏感变量的翻译后指向集。下一步，将修改后的宿主投影（桥接方法的投影）传播到调用方客户方法（该方法通过桥接方法边连接于

算法 m 4: InjectSummary

输入： 投影 R^f，预分析 $\mathcal{P}$
输入： 函数 f 的函数摘要 F^f
输出： 更新后的预分析 $P_{modular}$

变更变量 $\leftarrow \emptyset$
更新的投影 $R^f_\Delta \leftarrow \emptyset$
对所有 (访问路径, 堆) $\in \mathcal{F}^f$ **执行**
  如果 访问路径是一个简单变量 *var* **那么**
    $\mathcal{R}^f_\Delta = \mathcal{R}^f_\Delta \cup \{(var, 堆, 调用上下文)\}$ 对于任意 调用上下文 $\in \mathcal{R}^f$
  end
  如果访问路径是字段访问 *x.f* **那么**
    情况字段敏感分析 **做**
      $\mathcal{R}^f_\Delta = \mathcal{R}^f_\Delta \cup \{(o_i, f, heap) \mid \forall o_i \in P(x)\}$
    end
    情况基于字段分析 **做**
      $\mathcal{R}^f_\Delta = \mathcal{R}^f_\Delta \cup \{(f, heap)\}$
    end
    情况字段不敏感分析 **做**
      $\mathcal{R}^f_\Delta = \mathcal{R}^f_\Delta \cup \{(o_i, heap) \mid \forall o_i \in P(x)\}$
    end
  end
  变更变量 $\leftarrow$ 变更变量 $\cup$ {访问路径}
end
// 更新客体投影，以包含已解析的宿主投影
对 $f' \in G_{客体}$ 调用 f，它是一个桥接方法 **执行**
  // 将宿主函数的（f）返回变量传播到客体（f'），调整变量名称。
  $R^{f'} = R^{f'} \cup R^f_\Delta[\$ret]$
结束
$P_{modular} = P \cup R^{f'}$

桥接图中）。最后，修订预分析，并纳入翻译后的投影。经过这些修订，分析成功解析了（一个）桥接调用图的跨语言操作中的所有关键操作。

图 10 总结了摘要注入到预分析中的输出。第二列和第三列分别展示了摘要注入之前和之后的分析快照。由于摘要统一，getSecret 方法的返回值已更新为 {obj@sec}。由于 *getAndLeak* 调用了函数*getSecret*，摘要注入会自动将更新后的返回值传播到 *getAndLeak*，而g 的指向集现在包含 {obj@sec}。桥接调用图包含从*getAndLeak* 到 *foo*的连接，因此，{obj@sec} 会传播到变量 x。

2.3.4 总体工作流程。 算法 5 展示了前文所述算法的工作流程。我们的算法通过计算调用图并通过摘要特化统一预分析来运行，直到其不再发生变化，即达到不动点。

2.3.5 多个桥接调用。目前，源自不同宿主函数的桥接调用会生成多个桥接调用图，

清单 3. 宿主语言代码

```
1 foo() {
2   i = 新建 BridgeClass(); // 对象@iface eval("set()"); // 调用客体
3
4   // 执行一些计算
5   x = eval("getAndLeak()");
6   返回 x;
7 } 8 BridgeClass.setSecret(secret) {
9     此.Secret = Secret;
```

```
18 }
19 g = eval("getSecret()");
20
21
22 返回 g; 23 }
BridgeClass.setSecret(secret) {
  this.secret = secret;
10 }
11 BridgeClass.getSecret() { 言和访客语言
12   return this.secret; {obj@sec}
13 }
```

(a) 方法中的摘要注入

方法	注入前摘要	注入后摘要
foo	$i, ctx \mapsto$ {obj@iface} $\langle x, ctx\rangle \mapsto \emptyset$	$i, ctx \mapsto$ obj@iface} $\langle x, ctx\rangle \mapsto$ {obj@sec}
setSecret	$i.secret, ctx' \mapsto$ {obj@sec}	$i, secret, ctx \mapsto$ {obj@sec}
getSecret	$i.secret, ctx' \mapsto$ {obj@sec}	$i, secret, ctx \mapsto$ {obj@sec} $\$ret, ctx' \mapsto$ {obj@sec} $\$ret, ctx \mapsto$ {obj@sec}
集合	$v, ctx' \mapsto$ {obj@sec}	$i.secret, ctx' \mapsto$ {obj@sec}
getAndLeak	$\emptyset$	$g, ctx' \mapsto$ {obj@sec}

（b）摘要注入前后的分析

图 10：清单 1 和清单 2 的摘要注入

算法 5： 指针分析摘要特化

输入： P 预分析**输入：** $G_{宿主}G_{客体}$ 宿主与访客语言调用图
输出： 分析 $\mathcal{P}_{模块化}$

重复
  $G, B \leftarrow$ 模块化调用图分析($G_{宿主}G_{客体}$)
  $F \leftarrow$ SpecializeSummary(B, Pg)
  对所有 $f \in$函数(B) **执行**
    $\mathcal{R}^f \leftarrow$ 投影(f)
    $P_{模块化} \leftarrow$ InjectSummary(R^f, P, F^f)
  end
直到$\mathcal{P}_{模块化}$保持不变（即达到不动点）；

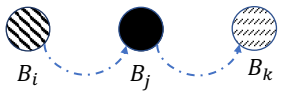


图 11：多个桥接调用图

因此（可能）会产生一组彼此不连通的桥接调用图，如图 11 所示。为了帮助预分析桥接调用图之间实际的数据流，各桥接调用图中解析出的摘要应按正确顺序传播到其他图中。我们的方法采用传统的拓扑排序，即预分析生成的执行顺序。例如，在图 11 中，分析会先解析桥接调用图 B_i 的数据流，然后通过摘要注入沿调用图向下传播修改（传递到 B_j，随后传递到 B_k）。

2.3.6 典型的数据流分析。给定一个传统的安全格，我们框架的核心部分会处理带有安全标签注释的变量所对应的指向集。解析出带注释的指向集后，污点分析便可查询这些指向集以获取别名集，随后将其用于污点传播 [18, 24, 46]。详细的信息流/污点分析作为补充材料提供。

3 局限性

图 1 仅允许将程序作为参数传递给 eval，因此不支持动态计算目标。在实践中，这并非当代程序的限制，但可以通过增加额外的分析来解除这一限制 [9, 23]。我们的方法使用客体侧接口对象的 may 别名。因此，当桥接方法与客户方法具有相同的名称和元数时，我们的分析可能会产生不精确的结果。不过，这是静态分析的一般局限性；此外，我们在数据集中尚未发现此类情况。该语言假设桥接接口已被指定，且其类型为分析所知，以便识别 *InterfaceVars*。这是一个合理的假设，因为现代语言会明确规定这些标记的语法和语义，例如，Android 为桥接接口定义了 WebView API。基于这一点，我们的分析假设通过接口对象调用的方法均可进行静态解析，即动态绑定不适用于通过接口对象调用的桥接类方法。我们的分析从预分析中获取这些静态解析的接口对象。

4 实现

我们开发了所提方法的原型实现，称为*Conflux*。*Conflux* 目前支持分析 Java-C 程序和 Java-Python 程序。*Conflux* 以 Java 实现，包含约 5k 行代码，并基于 WALA 分析框架 [16] 构建，用于分析 Java 和 Python。对于 C/C++ 程序的分析，我们选择了 SVF [49, 50], ，这是一款流行的 C/C++数据流分析工具。

可扩展性：撰写本文时，*Conflux* 支持 Wala 中的 WALA-SVF 和 Java-Python 分析。不过，*Conflux* 可扩展到其他语言组合（只需为语言绑定做一些实现工作）。*Conflux* 提供了一组接口（适配器），以便扩展到其他分析框架。包含接口详细信息的 API 文档将与 *Conflux* 一同开源。

要分析 Android NDK 程序，*Conflux* 以 Android APK 文件作为输入。它使用 *retdec* 反编译器 [11] 将 APK 文件中捆绑的 NDK 库提升为 LLVM。对于 Graal Polyglot API，它以 Java JAR 文件和 Python 源代码作为输入。

5 评估

我们通过实证评估 *Conflux* 在解析多语言应用中跨语言操作方面的有效性。为展示 *Conflux* 对其他可互操作语言的普适性，我们在两类多语言模型上评估 *Conflux*：(1) Android NDK 应用中的 Java-C 通信；(2) 使用 GraalVM 的 Java-Python 应用程序。Java-C 通信体现在使用系统级功能的 Java（或 Android）应用程序中，其原因各不相同，例如提升性能或与硬件交互。我们数据集中的 Java-Python 应用程序使用 GraalVM Polyglot API。具体而言，我们的评估旨在回答以下研究问题：

RQ1 *Conflux* 在精度和可扩展性方面与当前最先进的多语言分析相比如何?
RQ2 *Conflux* 能否推广到多种多语言模型?

所有实验均在一台运行 macOS Monterey 的笔记本电脑上进行，该电脑配备 10 核 M1 Pro 处理器和 32 GB 内存。

5.1 RQ1：与当前最先进水平的比较

近年来，已有一些研究工具 [15, 32, 39, 55] 用于应对多语言应用程序的分析。C-summary [32] 是最新的多语言分析工具，研究表明它克服了以往工作中的各种局限。因此，我们将 *Conflux* 与 C-summary [30] 的有效性进行比较。

受测应用程序。 为评估第一个研究问题，我们依据两个目标来选择受测应用程序：（1）纳入来自 C-summary 数据集的应用程序，以避免选择偏差；（2）评估 *Conflux* 在多种 Java 和 C 通信模式下的表现。综合这两个目标，我们从广泛使用的 NativeFlowBench [10, 55]中选取了 23 个 Android JNI 应用，并从 Google Play 商店中选取了 16 个热门应用程序。NativeFlowBench 是一个 Android 应用基准测试集，包含多种跨语言通信特性。该基准测试集中的每个应用程序都有真实结果。

接下来，我们根据以下参数评估所有工具：(1) 是否能够发现从宿主语言到客体语言以及反向的调用图边；(2) 是否能够发现客体语言中的可达方法，以生成全程序调用图。

表 1 包含了受测应用程序，并展示了 *Conflux* 和 C-summary 在这些应用程序上的评估结果。第一列显示受测应用程序的名称，第二列包含通过 C-summary 获得的分析结果。列 N_C 和 E_C 分别表示原生组件调用图中的节点数和边数；列 中间语言 列出了每个工具中从宿主语言指向客体语言的调用图边数。*Conflux* 被配置为对 Java 采用单调用点敏感和字段敏感分析，对 C/C++ 采用字段敏感的上下文不敏感分析。

我们使用 *C-summary* 的经验。 表 1 使用若干符号表示对 *Conflux* 和 C-summary 的评估结果。符号 □ 表示分析成功识别函数签名，但未能合成函数定义。在我们的实验中，19 个应用程序中有 11 个的 C-summary 未能合成函数定义。当程序包含复杂的指针操作（例如指针运算）或方法重载时，就会出现这种情况；例如，程序使用 strcpy 等字符串库操作，而这些操作没有源代码。符号 ⊤ 表示工具能够识别函数签名，但生成了一个空程序，例如 Java.Top()，它本质上会返回所有堆分配的集合（分析结果不精确）。在 19 个应用程序中，有 6 个应用程序的 C-summary 生成了空程序。当原生代码访问对象字段加载和存储方法时，就会出现这种情况。例如，在基准测试应用程序 *complexdata_stringop*中，C-summary 会为 Java 方法 GetFieldID 和 GetObjectClass 生成空程序。符号 表示工具既无法识别函数签名，也无法合成程序。19 个应用程序中，有 6 个应用程序中的 C-summary 出现了这种情况。这发生在两种情况下：(1) 对原生类执行 Intent 操作；(2) 执行 C-summary 未处理的操作，例如数组或全局变量，这些都是其框架已报告的局限性 [32]。因此，C-summary 无法合成 Android 分析工具后续分析所需的程序，从而隐藏了可通过桥接接口访问的函数。在

表 1：对来自 NativeFlowBench 微基准测试 [10]和真实世界JNI应用的JNI进行分析。T 表示底层工具合成了不精确的程序，例如 Java.Top()。 □ 表示该工具生成了空程序。

基准测试	C-summary	Conflux						
	生成	分析	N_J	E_J	N_C	E_C	IL	时间（秒）
NativeFlowBench								
复杂数据	□	✓	364	940	8	10	2	< 1
复杂数据 字符串操作	✗	✓	5802	13015	9	10	1	< 1
动态注册多个	✗	✓	382	979	9	13	3	< 1
堆修改	□	✓	6889	15272	11	12	1	< 1
ICC Java 到本地代码	✗	✗	366	895	13	14	0	< 1
ICC 本地代码到 Java	□	✓	7865	14891	8	9	1	< 1
leak	□	✓	367	894	3	4	1	< 1
数组泄漏	□	✓	383	968	4	5	1	< 1
动态注册泄漏	□	✓	367	895	9	11	1	< 1
方法重载	□	✓	307	751	3	5	2	< 1
多次交互	□	✓	384	980	11	13	2	< 1
多个库	□	✓	382	978	3	5	1	< 1
无泄漏	□	✓	367	894	1	2	1	< 1
无泄漏数组	□	✓	5795	13008	4	5	1	< 1
无源代码	⊤	✓	306	746	1	2	1	< 1
pure	✗	✗	6581	14686	39	42	0	< 1
纯直接调用	✗	✗	6581	14686	30	48	0	< 1
纯直接调用（定制）	✗	✗	380	958	30	48	0	< 1
从参数设置字段	⊤	✓	373	908	4	5	1	< 1
从参数字段设置字段	⊤	✓	314	763	4	5	1	< 1
从原生代码设置字段	⊤	✓	310	762	14	15	1	< 1
来源	⊤	✓	6892	15271	7	8	1	< 1
源代码清理	⊤	✓	385	985	4	5	1	< 1
Google Play 商店应用								
Airmessage	-	✓	53987	30886	169	141	4	30
Apple Music	-	✓	51867	28760	8334	3055	8	72
Arte	-	✗	57224	31237	74	31	0	480
ClearScore	-	✗	50834	31063	344	321	0	30
CommonsLab	-	✓	28817	16576	447	348	14	15
FPLayAndroid	⊤	✓	5052	2316	294	238	14	240
FairEMail	⊤	✓	59766	40214	3854	1116	5	85
JioMoney	-	✓	50002	22904	299	232	2	27
NetGuard	-	✓	12159	6448	333	168	2	5
PCAPdroid	⊤	✓	54270	28835	613	303	3	29
Proton VPN	⊤	✓	56217	31253	56217	8	1	54
Spotify	-	✗	59690	33736	439	177	0	600
Spotify Lite	-	✓	51728	31234	843	434	8	390
TermOne	⊤	✗	47352	24365	48	30	0	22
Termux	⊤	✓	15656	9603	77	36	6	6
TimidityAE	⊤	✓	15278	8275	187	127	6	1

许多情况下，它只能识别用于从 Java（Android）侧查找可达本机函数的函数定义。

Conflux的结果。*Conflux* 使用 SVF（针对 C/C++）和 Wala（针对 Java）作为预分析。我们将 SVF 配置为使用宿主语言分析中定义的入口点运行。基于这些入口点，它会触发 SVF 分析环境中的客体函数并收集结果。因此，SVF 会识别客体语言中可达的函数，*Conflux* 将分析结果与 Wala 得到的结果相结合，构建跨语言调用图。列 N_J 和 E_J 显示了相应预分析识别出的调用图节点数和边数。考虑到基准测试的规模，这些数字明显偏高，因为 Wala 包含若干库默认情况下，[41], 这会增加调用图的规模。列 中间语言报告调用图中的跨语言边，即从 Java 指向 C，反之亦然的边。列 E_C 显示 SVF 识别出的边。在我们的评估中，*Conflux* 能够在 23 个基准测试中的 19 个里识别跨语言边，也就是说，它能识别出此前被 Wala 报告为不可达的方法。在其余四种情况下，*Conflux* 无法识别通过意图调用原生代码的入口点；这是预期之中的，因为基于中间件的跨语言通信不在本研究的范围内。意图解析工具 [35, 51, 56] 可以补充*Conflux*，以解析这些入口点。

表 2：GraalVM 中的 Java-Python 应用

基准测试	Wala-Java	Wala-Python	Conflux
BM-1	✓	✓	✓
BM-2	✓	✓	✓
BM-3	✗	✓	✗

RQ1(a)： 在 23 个研究对象中的 19 个中，*Conflux* 构建了完整的跨语言调用图。*Conflux* 利用各项分析的优势，映射宿主到客体以及客体到宿主所需的调用边。

可扩展性。 为了评估 *Conflux* 在真实应用程序中的有效性，我们从 Google Play 商店下载了 50 款热门且近期推出的应用程序。为了筛除 NDK 应用，我们检查了它们的反编译代码，以确认其中是否存在共享对象文件，最终筛选出 16 款应用程序。这些应用程序包括 Spotify 和 Apple Music 等热门应用。我们的数据集中有少数应用与 C-summary 评估过的应用相同。

表 1 下半部分的第一部分展示了 C-summary 在这些应用程序上取得的结果。由于 C-summary 专为源代码设计，我们在七款开源应用程序上运行了它。在所有这些应用程序中，C-summary 都合成了不精确的程序（包含 Java.Top() 的语句）。

表 1 的第二部分展示了我们的分析在这些应用程序上的结果。我们的分析能够扩展到许多热门应用程序，并能在 10 分钟内分析相对大型的应用程序，例如 *Spotify*（中位数约为 2 分钟）。在这些应用程序中，*Conflux* 能够在 16 款应用程序中的 12 款识别出跨语言调用图边。同样，在其余四款应用程序中，客体函数是通过意图调用的，这超出了本文的研究范围。为验证跨语言边的数量，我们扫描了反编译文件，以检查二进制文件中是否存在原生标记。两位作者分别独立验证了这些结果。

RQ1(b)： 在 16 个大型应用程序中的 12 个里，*Conflux* 成功构建了跨语言调用图，平均耗时约 2 分钟。这表明，通过 *Con-flux* 结合各项独立分析，是实现可扩展的多语言应用数据流分析的一条有前景的路径。

5.2 RQ2：使用 GraalVM 分析 Java-Python 程序

为评估我们方法的泛化能力，我们将*Con- flux* 应用于基于 Java-Python 的 GraalVM 程序。GraalVM 的多语言 API 相对较新，我们在 GitHub 上没有找到许多使用基于 Java-Python 通信的项目。为了筛选应用程序，我们选取了包含关键词graalpython 的项目，以及导入org.graalvm.polyglot.Context类（或其子类）的项目。在 GitHub 上获得的七个结果中，我们搜索了是否包含 Python 代码片段和文件。最终，我们确认了两个以 Java 为宿主语言、Python 为客体语言的应用程序（表 2）。此外，我们还开发了一个微基准测试用于分析，因此总共研究了三个微基准测试。在 BM-1 [3]，中，Python 调用了 Java 的 setter 和 getter 方法。BM-2 [4] 从 Python 调用 Java 库方法。BM-3 [5]，中，Java 方法由 Python（作为宿主）调用。

Conflux 将 Wala 框架用作预分析工具，并针对两种语言配置为单调用点敏感和字段敏感，以分析 Java 和 Python 程序。*Conflux* 成功分析了 BM-1 和 BM-2，并正确解析了跨语言函数调用。BM3 使用 Java 作为客体语言、Python 作为宿主语言。因此，Wala-Java 无法分析该程序，因为它要求程序具有主方法，而这里客体语言被用作库。

RQ2： *Conflux* 可以配置为支持多种多语言框架。例如，将*Conflux* 配置为分析 GraalVM 程序后，它成功解析了三个 Java-Python 应用中的两个所使用的桥接接口。

6 相关工作

模块化多语言分析。Lee 等人 [32] 提出了一种模块化语义摘要提取技术，用于分析 Java-C/C++ 互操作。Monët 等人 [39] 使用抽象解释分析 Python 和 C 程序。Wei 等人提出的 JN-SAF [55] 采用基于函数摘要的自底向上数据流分析方法来分析 Java Native 应用。所有这些工作都依赖于宿主与客体之间的显式边界。此外，这些工作均与特定的宿主语言和客体语言绑定，而我们的工作不依赖于特定语言。Buro 等人提出了一种基于抽象解释的框架 [15]，通过结合 while 语言和表达式语言的分析来分析多语言程序。与他们的工作不同，我们的工作能够结合具有更复杂操作的语言。

单体式多语言分析。JET [36], Li 和 Tan [44], Kondoh 与 Onodera [29] 分析 Java JNI 接口中出现的错误。Fourtounis 等人 [22] 扫描原生二进制文件以提取数据流事实，并将其用于 Java 中的分析。与这些工作相比，我们的方法具有通用性，并不局限于 JNI 调用。Bae 等人 [12], [31], Lee 等人 [33] 提出了分析 Android WebView（Java-Javascript）程序的技术。Samhi 等人 [42] 将原生代码映射到 Soot 中间表示，以实现统一分析。Turcotte 等人 [54] 实现了一个类型检查器，用于检查 Lua-C 程序中的类型正确性。Brucker 和 Herzberg [14] 提出了一种静态分析方法，用于检测基于 Cordova（Java-Javascript）的应用中的安全漏洞。不同于这些多语言程序分析方面的工作，我们的工作定义了一种结合现有分析的模块化程序分析。

单语言应用的模块化分析。 这与我们的工作正交，但这些工作构成了我们分析的基础，作为预分析。Nielsen 等人提出了 JAM [40]，用于分析 *Node.js* 应用的模块化调用图构建。Tiwari 等人 [51]使用持久化摘要来解决 Android 中的组件间通信。Schubert 等人提出的 ModAnalyzer [43] 利用 C++ 开发者采用的模式，对 C++ 应用和库进行模块化分析。Dilling 等人 [19] 以及 Whaley 和 Rinard [57] 分别提出了针对 C 和 Java 的组合式指针分析。Illous 等人 [26] 提出了一种基于分离逻辑的组合式形状分析技术。

7 结论

在本项工作中，我们利用现有的语言分析来分析多语言程序。我们为此提出了一系列算法，并通过一个原型实现展示了我们的方法的有效性

，该实现名为 *Conflux*。我们的分析既不要求我们为多语言应用重新构建分析，也不需要将客体语言的现有语义转换为宿主语言的语义，从而节省了这一过程中的大量工作。

参考文献

[1] 2022. https://developer.android.com/ndk [2] 2022. https://www.graalvm.org [3] 2022. https://github.com/vpaz11/python-graal-java [4] 2022. https://github.com/shelajev/graalpython-java-example [5] 2022. https://github.com/vspaz/graalvm-java-python-interoperation/tree/main/src/main/java/org/vspaz [6] 2022用于无缝衔接 R 与 C 的 Rcpp++ 集成。http://www.rcpp.org [7] 2022Rpy2：R 中的 Python。https://rpy2.github.io/doc/v2.9.x/html/index.html [8] 匿名作者。2022。Conflux 工件。https://figshare.com/s/ 872fc296f1a0827352ac[9] Vincenzo Arceri 和 Isabella Mastroeni。2021。动态代码分析：一种针对邪恶 eval 的健全抽象解释器。ACM 隐私与安全汇刊。24，2，文章 10（1月 2021 38 3426470），页码。[10] [ ]。https://doi.org/10.1145/ ArgubLab。无日期。NativeFlowBench。https://github.com/arguslab/ NativeFlowBench[11] Avast。2022。Avast/RETDEC：Retdec 是一种基于 LLVM 的可重定向机器代码反编译器。https://github.com/avast/retdec [12] Sora Bae、Sungho Lee 和 Sukyoung Ryu。2019。迈向对 Android 互操作的理解与推理。载于 2019 IEEE/ACM 第 41 届软件工程国际会议（ICSE）。223–233。https://doi.org/10.1109/ ICSE.2019.00038[13] Martin Bravenboer 和 Yannis Smaragdakis。2009。复杂指向分析的严格声明式规范。载于第 24 届 ACM SIGPLAN 面向对象编程系统、语言与应用会议论文集（美国佛罗里达州奥兰多）（OOPSLA '09）。ACM，纽约，NY，美国，243 262 . 10.1145 1640089.1640108 –。https://doi org/ / Achim D. [14] 2016 Brucker 和 Michael Herzberg。混合移动应用的静态分析。载于《工程安全软件与系统》，Juan Caballero、Eric Bodden 和 Elias Athanasopoulos（编）。施普林格国际出版社，沙姆，72–88。[15] Samuele Buro、Roy L. Crole 和 Isabella Mastroeni。2020。多语言抽象。载于《静态分析》，David Pichardie 和 Mihaela Sighireanu（编）。施普林格国际出版社，沙姆，310–332。[16] IBM T.J. Watson 研究中心。2022。用于分析的 Wala T.J. Watson 库。https://github.com/wala/WALA。[17] Patrick Cousot 和 Radhia Cousot。2002。模块化静态程序分析。载于《编译器构造》，R. Nigel Horspool（编）。施普林格柏林海德堡出版社，柏林、海德堡，159–179。[18] Dipanjan Das、Priyanka Bose、Aravind Machiry、Sebastiano Mariani、Yan Shoshi-taishvili、Giovanni Vigna 和 Christopher Kruegel。2022。混合剪枝：迈向精确的指针与污点分析。载于《入侵与恶意软件检测及漏洞评估》，Lorenzo Cavallaro、Daniel Gruss、Giancarlo Pel-legrino 和 Giorgio Giacinto（编）。施普林格国际出版社，沙姆，1–22。[19] Isil Dillig、Thomas Dillig、Alex Aiken 和 Mooly Sagiv。2011。堆操作程序的精确且紧凑的模块化过程摘要。载于第 32 届 ACM SIGPLAN 编程语言设计与实现会议论文集（美国加利福尼亚州圣何塞）（PLDI '11）。计算机协会，纽约，NY，美国，567 577 . 10.1145 1993498.1993565 –。[20] 2022 https://doi org/ / Stack Exchange。为何在开发一个产品或一段软件时会使用多种编程语言？ https://softwareengineering.stackexchange.com/questions/370135/why-are-multiple-programming-languages-used-in-the-development-of-one-product-or/ 370146 370146 # Asger Feldthaus、Max Schäfer、Manu Sridharan、[21] Julian Dolby 和 Frank Tip。2013。为 JavaScript IDE 服务高效构建近似调用图。载于 ICSE。https://doi . 10.1109 .2013.6606621 [22] org/ /ICSE George Fourtounis、Leonidas Triantafyllou 和 Yannis Smaragdakis。2020。通过二进制扫描识别原生代码中的 Java 调用。计算机协会，纽约，NY，美国。https://doi . 10.1145 3395363.3397368 [23] 2018 org/ / Neville Grech、George Kastrinis 和 Yannis Smaragdakis。高效的基于图着色的反射字符串分析。载于第 32 届欧洲面向对象编程会议（ECOOP 2018）（莱布尼茨国际信息学会议录（LIPIcs），第 109 卷），Todd Millstein（编）。施洛斯·达格施图尔—莱布尼茨信息学中心，达格施图尔，德国，26:1–26:25。https://doi.org/10.4230/LIPIcs.ECOOP.2018.26[24] Neville Grech 和 Yannis Smaragdakis。2017。P/Taint：统一的指向分析与污点分析。ACM 程序设计语言汇刊。1，OOPSLA，文章 102（2017年10月），28页。https://doi.org/10.1145/3133926

[25] Manel Grichi、Ellis E. Eghan 和 Bram Adams。2020。多语言开发对机器学习框架的影响。载于 2020 年 IEEE 国际软件维护与演化会议（ICSME）。546–556。https://doi . 10.1109 46990.2020.00058 org/ /ICSME Hugo Illous、Matthieu [26] Lemerre 和 Xavier Rival。2020。使用基于分离逻辑的转换器摘要进行过程间形状分析。载于《静态分析》，David Pichardie 和 Mihaela Sighireanu（编）。施普林格国际出版社，沙姆，248–273。[27] Uday P. Khedker、Amitabha Sanyal 和 Amey Karkare。2007。使用访问图进行堆引用分析。《ACM 程序设计语言与系统汇刊》30，1（2007 年 11 月），1 . 10.1145 1290520.1290521 –页。https://doi org/ / Pavneet Singh [28] Kochhar、Dinusha Wijedasa 和 David Lo。2016。多种编程语言与代码质量的大规模研究。载于 2016 年 IEEE 第 23 届国际软件分析、演化与再工程会议（SANER），第 1 563 573 . 10.1109 .2016.112 卷。–。https://doi org/ /SA [29] 2008 NER Goh Kondoh 和 Tamiya Onodera。寻找 Java 本地接口程序中的错误。载于 2008 年国际软件测试与分析研讨会论文集（美国华盛顿州西雅图）（ISSTA '08）。计算机协会，纽约州纽约，美国，109 118 . 10.1145 1390630.1390645 –。https://doi [30] 2022 . org/ / Sungho Lee。C 摘要工件。https://github com/SunghoLee/c-summary [31] Sungho Lee、Julian Dolby 和 Sukyoung Ryu。2016。HybriDroid：Android 混合应用的静态分析框架。载于第 31 届 IEEE/ACM 自动化软件工程国际会议论文集（新加坡）（ASE 2016）。计算机协会，纽约州纽约，美国，250 261 . 10.1145 2970276.2970368 –。https://doi org/ / Sungho Lee、[32] Hyogun Lee 和 Sukyoung Ryu。2020。拓展多语言静态分析的视野：从 C 代码中提取用于 JNI 程序分析的语义摘要。载于第 35 届 IEEE/ACM 自动化软件工程国际会议论文集（线上活动，澳大利亚）（ASE '20）。计算机协会，纽约州纽约，美国，127–137。https://doi.org/10.1145/ 3324884.3416558[33] Sungho Lee 和 Sukyoung Ryu。2019。Adlib：移动广告平台库分析器。计算机协会，纽约州纽约，美国，262–272。https://doi . 10.1145 3293882.3330562 org/ / Johannes Lerch、Johannes Späth、Eric [34] Bodden 和 Mira Mezini。2015。访问路径抽象：利用无界访问路径扩展字段敏感数据流分析。载于第 30 届 IEEE/ACM 自动化软件工程国际会议论文集（内布拉斯加州林肯）（ASE '15）。IEEE 出版社，619– 629 . 10.1109 .2015.9 。https://doi org/ / [35] ASE Li Li、Alexandre Bartel、Tegawendé F. Bissyandé、Jacques Klein、Yves Le Traon、Steven Arzt、Siegfried Rasthofer、Eric Bodden、Damien Octeau 和 Patrick McDaniel。2015。IccTA：检测 Android 应用中的组件间隐私泄漏。载于 2015 年 IEEE/ACM 第 37 届 IEEE 软件工程国际会议，第 1 280 291 . 10.1109 .2015.48 卷。–。https://doi org/ /ICSE [36] 2011 Siliang Li 和 Gang Tan。JET：Java 本地接口中的异常检查。载于 2011 年 ACM 面向对象编程系统、语言与应用国际会议论文集（美国俄勒冈州波特兰）（OOPSLA '11）。计算机协会，纽约州纽约，美国，345 358 . 10.1145 2048066.2048095 –。[37] https://doi org/ / Zengyang Li、Xiaoxiao Qi、Qinyi Yu、Peng Liang、Ran Mo 和 Chen Yang。2021。开源软件中的多编程语言提交：关于 Apache 项目的实证研究。载于 2021 年 IEEE/ACM 第 29 届国际程序理解会议（ICPC）。219 229 . 10.1109 52881.2021.00029 –。https://doi org/ / [38] 2005 ICPC Ana Milanova、Atanas Rountev 和 Barbara G. Ryder。面向 Java 指向分析的参数化对象敏感性。《ACM 软件工程与方法学汇刊》14，1（2005 年 1 月），1–41。10.1145 https://doi org/ / 1044834.1044835 Raphaël [39] Monat、Abdelraouf Ouadjaout 和 Antoine Miné。2021。带原生 C 扩展的 Python 程序的多语言静态分析。载于《静态分析》，Cezara Drăgoi、Suvam Mukherjee 和 Kedar Namjoshi（编）。施普林格国际出版社，沙姆，323–345。[40] Benjamin Barslev Nielsen、Martin Toldam Torp 和 Anders Møller。2021。面向 Node.js 应用安全扫描的模块化调用图构建。载于第 30 届软件测试与分析国际研讨会（ISSTA）。[41] Jyoti Prakash、Abhishek Tiwari 和 Christian Hammer。2021。程序表示对指针分析的影响——一项实证研究。载于《软件工程基础方法》，Esther Guerra 和 Mariëlle Stoelinga（编）。施普林格国际出版社，沙姆，240–261。[42] Jordan Samhi、Jun Gao、Nadia Daoudi、Pierre Graux、Henri Hoyez、Xiaoyu Sun、Kevin Allix、Tegawendé F. Bissyandé 和 J acques Klein。2021。Ju-Cify：迈向 Android 代码统一以增强静态分析。arXiv: 2112 10469 [ ]。cs.SE Philipp Dominik Schubert、Ben Hermann 和 Eric [43] Bodden。2021。静态调用图、指向分析和数据流分析的无损持久化摘要。载于第 35 届欧洲面向对象编程会议（ECOOP 2021）（《莱布尼茨国际信息学会议录》（LIPIcs），第 194 卷），Anders Møller 和 Manu Sridharan（编）。达格施图尔城堡——莱布尼茨信息学中心，德国达格施图尔，2:1–2:31。https://doi.org/10.4230/LIPIcs.ECOOP.2021.2

[44] Li Siliang 和 Tan Gang。2009。在 JNI 程序的异常情况下发现错误。载于第 16 届 ACM 计算机与通信安全会议论文集（美国伊利诺伊州芝加哥）（CCS '09）。计算机协会，纽约州纽约，美国，442 452 . 10.1145 1653662.1653716 –。https://doi
[45] org/ / Johannes Späth、Karim Ali 和 Eric Bodden。使用同步下推系统进行上下文敏感、流敏感和字段敏感的数据流分析。ACM 程序设计语言会议录（3, 2019 48 1 48 29 3290361 POPL），（），：–：。
[46] https://doi.org/10.1145/ Johannes Späth、Lisa Nguyen Quang Do、Karim Ali 和 Eric Bodden。2016。Boomerang：面向 Java 的需求驱动流敏感与上下文敏感指针分析。载于第 30 届欧洲面向对象编程会议（ECOOP 2016）（《莱布尼茨国际信息学会议录》（LIPIcs），第 56 卷），Shriram Krishnamurthi 和 Benjamin S. Lerner（编）。达格施图尔城堡——莱布尼茨信息学中心，德国达格施图尔，22:1–22:26。https://doi.org/ 10.4230/LIPIcs.ECOOP.2016.22[47] Manu Sridharan、Satish Chandra、Julian Dolby、Stephen J. Fink 和 Eran Yahav。2013。面向对象程序的别名分析。施普林格柏林海德堡，柏林、海德堡，196 232 978 3 642 36946 9_8 –。
[48] https://doi.org/10.1007/ - - - - Manu Sridharan 和 Stephen J. Fink。Andersen 分析在实践中的复杂性。载于《静态分析》，Jens Palsberg 和 Zhendong Su（编）。施普林格柏林海德堡，柏林、海德堡，205–221。[49] Yulei Sui 和 Jingling Xue。2016。SVF：LLVM 中的过程间静态值流分析。载于第 25 届国际编译器构造会议论文集（西班牙巴塞罗那）（CC 2016）。计算机协会，纽约州纽约，美国，265 266 . 10.1145 2892208.2892235 –。
[50] https://doi org/ / Yulei Sui、Ding Ye 和 Jingling Xue。使用完全稀疏值流分析静态检测内存泄漏。《IEEE 软件工程汇刊》40，2（2014），107–122。[51] Abhishek Tiwari、Sascha Groß 和 Christian Hammer。2019。IIFA：Android 应用的模块化应用间意图信息流分析。载于《通信网络中的安全与隐私》，Songqing Chen、Kim-Kwang Raymond Choo、Xinwen Fu、Wenjing Lou 和 Aziz Mohaisen（编）。施普林格国际出版社，沙姆，335–349。[52] A. Tiwari、J. Prakash、S. GroB 和 C. Hammer。2019。LUDroid：Android—Web 混合化的大规模分析。载于 2019 年 IEEE 第 19 届国际源代码分析与操作会议（SCAM）。IEEE 计算机学会，美国加利福尼亚州洛斯阿拉米托斯，256 267 . 10.1109 .2019.00036 –。https://doi org/ /
[53] SCAM Abhishek Tiwari、Jyoti Prakash、Sascha Groß 和 Christian Hammer。Android—Web 混合化的大规模分析。《系统与软件杂志》170 2020 110775 . 10.1016 . .2020.110775 （），。https://doi org/ /j
[54] jss Alexi Turcotte、Ellen Arteca 和 Gregor Richards。在不建模外部语言的情况下推理外部函数接口。载于第 33 届欧洲面向对象编程会议（ECOOP 2019）（《莱布尼茨国际信息学会议录》（LIPIcs），第 134 卷），Alastair F. Donaldson（编）。达格施图尔城堡——莱布尼茨信息学中心，德国达格施图尔，16 1 16 32 . 10.4230 . .2019.16 : – :。
[55] https://doi org/ /LIPIcs ECOOP Fengguo Wei、Xingwei Lin、Xinming Ou、Ting Chen 和 Xiaosong Zhang。2018。JN-SAF：用于对含原生代码的 Android 应用进行安全审查的精确高效 NDK/JNI 感知跨语言静态分析框架。载于 2018 年 ACM SIGSAC 计算机与通信安全会议论文集（加拿大多伦多）（CCS '18）。计算机协会，纽约州纽约，美国，1137 1150 . 10.1145 3243734.3243835 –。https://doi org/ /
[56] Fengguo Wei、Sankardas Roy、Xinming Ou 和 Robby。Amandroid：用于 Android 应用安全审查的精确通用组件间数据流分析框架。《ACM 隐私与安全汇刊》21，3，文章 14（2018 年 4 月），32 3183575 页。https://doi.org/10.1145/ John
[57] Whaley 和 Martin Rinard。面向 Java 程序的组合式指针与逃逸分析。载于第 14 届 ACM SIGPLAN 面向对象编程、系统、语言与应用会议论文集（美国科罗拉多州丹佛）（OOPSLA '99）。计算机协会，纽约州纽约，美国，187–206。https://doi.org/10.1145/320384.320400