POLYCRUISE: A Cross-Language Dynamic Information Flow Analysis

摘要

Despite the fact that most real-world software systems today are written in multiple programming languages, existing program analysis based security techniques are still limited to single-language code. In consequence, security flaws (e.g., code vulnerabilities) at and across language boundaries are largely left out as blind spots. We present POLYCRUISE, a technique that enables holistic dynamic information flow analysis (DIFA) across heterogeneous languages hence security applications empowered by DIFA (e.g., vulnerability discovery) for multilingual software. POLYCRUISE combines a light language-specific analysis that computes symbolic dependencies in each language unit with a language-agnostic online data flow analysis guided by those dependencies, in a way that overcomes language heterogeneity. Extensive evaluation of its implementation for Python-C programs against micro, medium-sized, and large-scale benchmarks demonstrated POLYCRUISE's practical scalability and promising capabilities. It has enabled the discovery of 14 unknown crosslanguage security vulnerabilities in real-world multilingual systems such as NumPy, with 11 confirmed, 8 CVEs assigned, and 8 fixed so far. We also contributed the first benchmark suite for systematically assessing multilingual DIFA.

尽管当今大多数现实世界的软件系统都是用多种编程语言编写的，但现有的基于程序分析的安全技术仍然仅限于单语言代码。因此，语言边界处和跨语言边界的安全漏洞（例如代码漏洞）在很大程度上被视为盲点。我们提出了 POLYCRUISE，这是一种能够跨异构语言进行整体动态信息流分析 (DIFA) 的技术，因此可以为多语言软件提供由 DIFA 授权的安全应用程序（例如漏洞发现）。 POLYCRUISE 将计算每个语言单元中符号依赖关系的轻量语言特定分析与由这些依赖关系引导的与语言无关的在线数据流分析相结合，以克服语言异质性的方式。针对微型、中型和大型基准测试对其 Python-C 程

序实现的广泛评估证明了 POLYCRUISE 的实际可扩展性和有前途的功能。它已经在 NumPy 等现实世界的多语言系统中发现了 14 个未知的跨语言安全漏洞，目前已确认 11 个，分配了 8 个 CVE，并修复了 8 个。我们还提供了第一个用于系统评估多语言 DIFA 的基准套件。

1. Introduction

> Real-world software systems today are mostly multilingualthey consist of integral code units written in different programming languages [28, 34, 44, 45]. Moreover, the past decade has seen growth in both the prevalence and dominance of multilingual software and the average number of languages used in each system [38]. Given their critical role in the modern cyberspace, there is an urgent quest for systematically assuring the security of multilingual software systems.

今天的现实世界软件系统大多是多语言的，它们由用不同编程语言编写的完整代码单元组成 [28, 34, 44, 45]。此外，在过去的十年中，多语言软件的流行和主导地位以及每个系统中使用的平均语言数量都在增长 [38]。鉴于它们在现代网络空间中的关键作用，迫切需要系统地确保多语言软件系统的安全性。

> Yet despite this criticality and urgency, technique and tool support for securing multilingual systems remains largely lacking. Unlike in single-language systems, insecure (e.g., vulnerable or malicious) code behaviors may exist not only within individual language units but also at and across the interfaces between different languages in multilingual systems. As a result, even if each single-language unit of a multilingual system is secure, the entire system may still not be as a whole. Thus, holistic (cross-language) validation of multilingual programs against insecure properties is essential.

然而，尽管如此重要和紧迫，保护多语言系统的技术和工具支持仍然很大程度上缺乏。与单语言系统不同，不安全的（例如，易受攻击的或恶意的）代码行为可能不仅存在于单个语言单元内，而且还存在于多语言系统中不同语言之间的接口处和之间。即使多语言系统的每个单语言单元都是安全的，整个系统可能仍然不是一个整体。因此，针对不安全属性对多语言程序进行整体（跨语言）验证是必不可少的。

```
    For another example, a language-agnostic dynamic slicer, ORBS
[5] computes backward dynamic dependencies for multilingual code
based on heuristic removal of program statements treated as text
lines. Unfortunately, this design suffers from intrinsic
scalability issues, as analytically confirmed by others [31] and
empirically validated by ourselves. On ten real-world
multilingual subjects of 2~17 KSLOC (on GitHub) with ten randomly
chosen slicing criteria in each and a 24-hour per-criterion
timeout, ORBS was only able to produce results for 27 (out of the
100 in total) criteria across the three smallest subjects at a
cost of 9.47 hours for each.
```

再举一个例子，与语言无关的动态切片器 ORBS [5] 基于启发式删除作为文本行的程序语句来计算多语言代码的后向动态依赖关系。不幸的是，这种设计存在内在的可扩展性问题，正如其他人分析证实的那样 [31] 并由我们自己进行了经验验证。在 2~17 KSLOC（在 GitHub 上）的 10 个真实世界多语言主题中，每个主题中有 10 个随机选择的切片标准和 24 小时的每个标准超时，ORBS 只能产生 27 个结果（总共 100 个））三个最小科目的标准，每个科目的成本为 9.47 小时。

To fill the present gap, we take the first step to enable practical security defense support for multilingual software through cross-language dynamic information flow analysis (DIFA). We target DIFA as it has been a fundamental technique [66] underlying a range of security applications (e.g., vulnerability discovery [18], intrusion detection [46], security policy validation [71]). However, developing a holistic DIFA for multilingual systems faces two major challenges:

Semantics disparity. The heterogeneous languages used in a multilingual system represent disparate language semantics. Thus, neither can existing (single-language) DIFA be applied immediately, nor can we perform a DIFA on each language unit separately and then stitch the results.
Analysis cost-effectiveness. To be practically useful, the DIFA must scale to large, real-world systems—in fact, a multilingual system tends to be larger in (code) size than single-language ones. Also, the analysis needs to be effective (offering a reasonable level of accuracy) for the target security application (e.g., discovering new vulnerabilities).

为了填补目前的空白，我们迈出了第一步，通过跨语言动态信息流分析（DIFA）为多语言软件提供实用的安全防御支持。我们的目标是 DIFA，因为它是一系列安全应用程序（例如，漏洞发现 [18]、入侵检测 [46]、安全策略验证 [71]）的基础技术 [66]。但是，为多语言系统开发整体 DIFA 面临两个主要挑战：

- 语义差异。多语言系统中使用的异构语言代表不同的语言语义。因此，既不能立即应用现有的（单语言）DIFA，也不能单独对每个语言单元执行 DIFA，然后拼接结果。
- 分析成本效益。为了实际有用，DIFA 必须扩展到大型的现实世界系统——事实上，多语言系统的（代码）大小往往比单语言系统大。此外，分析需要对目标安全应用程序（例如，发现新漏洞）有效（提供合理的准确度）。

Meanwhile, we cannot simply bypass these challenges by resorting to single-language software construction. As different languages have unique advantages in facilitating the implementation of certain functionalities, developers do benefit from multi-language development for better productivity. In this paper, we aim to tackle the challenges of cross-language DIFA by exploiting two key insights outlined below.

While the differences in languages' semantics greatly impede purely static semantic analysis (e.g., statically computing data/control dependencies) across those languages, computation of dynamic data/control flow facts required for DIFA can be more readily unified, so as to overcome the semantics disparity challenge hence enable holistic DIFA.
The unified hence language-agnostic dynamic analysis of DIFA can still be guided and reduced by a static syntactic analysis specific to each language, which helps ensure the overall scalability and efficiency of the DIFA. Meanwhile, the imprecision of the syntactic analysis can be compensated by the dynamic analysis. This will overcome the cost-effectiveness challenge without sacrificing scalability.

同时，我们不能简单地通过单语言软件构建来绕过这些挑战。由于不同的语言在促进某些功能的实现方面具有独特的优势，因此开发人员确实可以从多语言开发中受益，从而提高生产力。在本文中，我们旨在通过利用下面概述的两个关键见解来应对跨语言 DIFA 的挑战。

- 虽然语言语义的差异极大地阻碍了这些语言之间的纯静态语义分析（例如，静态计算数据/控制依赖关系），但 DIFA 所需的动态数据/控制流事实的计算可以更容易地统一，从而克服语义差异挑战因此可以实现整体 DIFA。
- DIFA 的统一因此与语言无关的动态分析仍然可以通过特定于每种语言的静态句法分析来指导和减少，这有助于确保 DIFA 的整体可扩展性和效率。同时，动态分析可以弥补句法分析的不精确性。这将在不牺牲可扩展性的情况下克服成本效益挑战。

Following these insights, we have developed POLYCRUISE, a cross-language DIFA with an application to DIFA-based vulnerability discovery in multilingual systems written in Python and C. The overarching principle of our POLYCRUISE design is to combine minimal, semantics-independent languagespecific analyses with a language-agnostic dynamic analysis. The hybrid analysis strategy is justified as follows: the latter computes the eventual output of a DIFA (i.e., information flow paths between given sources and sinks), while the former computes approximate dependencies to determine the scope of instrumentation. In particular, each language-specific analysis translates a language unit's code to a language-independent symbolic representation (LISR). Then, POLYCRUISE uses a light syntactic analysis called symbolic dependence analysis (SDA) to compute the approximate dependencies based on the LISR with respect to the sources/sinks. At runtime, the instrumentation guided by these symbolic dependencies generates cross-language execution events, from which a dynamic information flow graph (DIFG) is incrementally built on the fly. Meanwhile, different kinds of vulnerabilities are discovered based on the DIFG via respective plugins.

根据这些见解，我们开发了 POLYCRUISE，这是一种跨语言 DIFA，可应用于用 Python 和 C 编写的多语言系统中基于 DIFA 的漏洞发现。我们 POLYCRUISE 设计的总体原则是将最小的、语义独立的语言特定分析与与语言无关的动态分析。混合分析策略的合理性如下：后者计算 DIFA 的最终输出（即给定源和接收器之间的信息流路径），而前者计算近似依赖关系以确定检测范围。特别是，每种特定语言的分析都将语言单元的代码转换为与语言无关的符号表示（LISR）。然后，POLYCRUISE 使用称为符号依赖分析（SDA）的轻量级句法分析来计算基于 LISR 对源/汇的近似依赖。在运行时，由这些符号依赖项引导的工具会生成跨语言执行事件，动态信息流图（DIFG）会从这些事件中动态构建。同时，通过相应的插件基于 DIFG 发现不同类型的漏洞。

> To validate our design, we implemented POLYCRUISE for Python-C programs given the consistently popular use of Python [21, 56], as backed by C for many functionalities, in impactful (e.g., machine learning) systems. While the popularity/impact may make our tool more significant, supporting Python is more challenging (e.g., compared to analyzing JNI programs). First, unlike for C/C++ and Java, analysis facilities for Python are largely lacking. Second, Python has rich dynamic constructs, making Python code incomplete until at runtime. We overcame these challenges by developing new analysis support for Python while using dynamic instrumentation to deal with its dynamic nature. We also implemented seven vulnerability detection plugins to assess POLYCRUISE's ability to support the discovery of various types of vulnerabilities (e.g., buffer overflow, sensitive data leak). As described in Section 4, the approaches described in this paper should be transferable to other language combinations.

为了验证我们的设计，我们为 Python-C 程序实现了 POLYCRUISE，因为 Python [21, 56] 在有影响力的（例如，机器学习）系统中一直很受欢迎，并由 C 支持许多功能。虽然流行/影响可能使我们的工具更加重要，但支持 Python 更具挑战性（例如，与分析 JNI 程序相比）。首先，与 C/C++ 和 Java 不同，Python 的分析工具在很大程度上缺乏。其次，Python 具有丰富的动态构造，使得 Python 代码在运行时之前是不完整的。我们通过为 Python 开发新的分析支持来克服这些挑战，同时使用动态工具来处理它的动态特性。我们还实现了七个漏洞检测插件来评估 POLYCRUISE 支持发现各种类型漏洞（例如缓冲区溢出、敏感数据泄漏）的能力。如第 4 节所述，本文中描述的方法应该可以转移到其他语言组合。

With this implementation, we evaluated POLYCRUISE on 12 real-world multilingual systems of diverse domains and scales against various executions. Our results show its high scalability and cost-effectiveness, as well as its enabling capabilities for cross-language DIFA. The static analysis part took in total <3 seconds for systems of 220 KSLOC or smaller and <3 minutes for our largest subject (of 6,419 KSLOC). The (online) dynamic analysis incurred 2.71~11.96x run-time slowdown. POLYCRUISE has found 14 unknown, cross-language vulnerabilities, including those in NumPy [49], with 11 confirmed, 8 CVEs assigned, and 8 fixed by developers so far. We also built the first multilingual dynamic analysis benchmark PyCBench, with which we validated the high precision (93.5%) and recall (100%) of the DIFA in POLYCRUISE.

通过这个实现，我们在 12 个不同领域的真实世界多语言系统上评估了 POLYCRUISE，并针对各种执行进行了扩展。我们的结果显示了它的高可扩展性和成本效益，以及它对跨语言 DIFA 的支持能力。对于 220 KSLOC 或更小的系统，静态分析部分的总耗时不到 3 秒，对于我们最大的主题（6,419 KSLOC）而言，总耗时不到 3 分钟。 （在线）动态分析导致 2.71~11.96 倍的运行时间减速。 POLYCRUISE 已发现 14 个未知的跨语言漏洞，包括 NumPy [49] 中的漏洞，目前已确认 11 个，分配了 8 个 CVE，8 个已由开发人员修复。我们还构建了第一个多语言动态分析基准 PyCBench，我们用它验证了 POLYCRUISE 中 DIFA 的高精度（93.5%）和召回率（100%）。

Through POLYCRUISE, we have demonstrated a novel methodology for practical DIFA of multilingual dynamic analysis, which can empower applications other than detecting information flow vulnerabilities and even beyond the security domain. In sum, our contributions include:

A scalable dynamic analysis technique for multilingual software written in Python and C, POLYCRUISE, which exploits light language-specific static analyses and online language-agnostic dynamic analysis to enable the first crosslanguage DIFA to the best of our knowledge (§3).
An open-source implementation of POLYCRUISE for Python-C programs, which works with large-scale, realworld multilingual systems in different domains (§4).
An open DIFA test suite covering a variety of analysis features, which is the first cross-language dynamic analysis benchmark suite publicly available as we know of (§5).
An extensive evaluation of POLYCRUISE, which demonstrates the promising efficiency, cost-effectiveness, and vulnerability discovery capabilities of our technique (§6).

通过 POLYCRUISE，我们展示了一种用于多语言动态分析的实用 DIFA 的新方法，该方法可以增强除检测信息流漏洞甚至超出安全域之外的应用程序。总之，我们的贡献包括：

- 一种用于用 Python 和 C 编写的多语言软件的可扩展动态分析技术 POLYCRUISE，它利用轻量级语言特定的静态分析和与语言无关的在线动态分析，据我们所知，启用第一个跨语言 DIFA（第 3 节）。
- 用于 Python-C 程序的 POLYCRUISE 的开源实现，可与不同领域中的大规模、真实世界多语言系统一起使用（第 4 节）。
- 一个开放的 DIFA 测试套件，涵盖各种分析功能，这是我们所知的第一个公开可用的跨语言动态分析基准套件（第 5 节）。
- 对 POLYCRUISE 的广泛评估，证明了我们技术的高效、成本效益和漏洞发现能力（第 6 节）。

我们已经发布了我们的数据集和代码，以方便复制和重用，正如所有发现的那样

2. Background and Motivation

> Different languages may interact via diverse interfacing mechanisms. This diversity partly makes cross-language DIFA challenging. Meanwhile, the need for tackling the diversity motivates and justifies our design for POLYCRUISE.

不同的语言可以通过不同的接口机制进行交互。这种多样性在一定程度上使跨语言 DIFA 具有挑战性。同时，应对多样性的需求激发并证明了我们对 POLYCRUISE 的设计。

> Interfacing mechanisms. At a high level, the ways different language units interact fall into two main categories:
>
> Uniform mechanism: via interprocess communication (IPC). This is universally applicable and totally languageindependent. For instance, Remote Procedure Call (RPC), which is widely used in middleware, is of this kind.
> Language-specific mechanism: via foreign function interface (FFI). Different language units interoperate through direct function invocations. Thus, this mechanism is strongly language-dependent. Current mainstream languages (e.g., Java, Python, PHP, Ruby [21]) all support FFI for C per their official documentations.

接口机制。在高层次上，不同语言单元的交互方式分为两大类：

- 统一机制：通过进程间通信（IPC）。这是普遍适用的并且完全独立于语言。例如，在中间件中广泛使用的远程过程调用（RPC）就是这种类型。
- 特定于语言的机制：通过外部函数接口（FFI）。不同的语言单元通过直接函数调用进行互操作。因此，这种机制强烈依赖于语言。当前的主流语言（例如，Java、Python、PHP、Ruby [21]）都根据其官方文档支持 C 的 FFI。

在更详细的层面上，接口机制的多样性更大。例如，在同一（FFI）类别中，Java 通过本机函数调用与 C 交互，通过 jobject 传递参数，而 Python 和 C 之间的接口则不同且更复杂 [59]。即使在同一种语言组合中，机制也各不相同。例如，在 Python-C 程序中，可以使用 ctypes 加载 C 库，然后搜索并调用 C 函数，或者可以将 C 模块构建为 Python 扩展以用作 Python 模块。另一方面，在将 C 类型参数转换为

PyObjects 之后，C 单元可以通过 Python 解释器提供的 API 调用 Python 函数。其他主流语言提供类似但仍然多样化的机制（例如，Go 通过 cgo 命令与 C 交互，而 Ruby 通过特定的 FFI 与 C 交互）。

> Due to these diversities at multiple levels, DIFA designs based on specific mechanisms would have limited applicability. Moreover, many modern languages (e.g., Python) are dynamic, for which static analysis is impeded and dynamic analysis is necessary [72]. Thus, designs that rely on substantial and deep (e.g., semantic) static analysis would have limited cost-effectiveness (low precision and/or low recall).

由于这些多层次的多样性，基于特定机制的 DIFA 设计的适用性有限。此外，许多现代语言（例如 Python）是动态的，因此静态分析受到阻碍，动态分析是必要的 [72]。因此，依赖于实质性和深度（例如语义）静态分析的设计将具有有限的成本效益（低精度和/或低召回率）。

Illustrating/motivating examples. Figure 1 depicts three cases of cross-language vulnerabilities each with a different interfacing mechanism between Python and C. pi and ci denotes line no. i in a Python and C unit, respectively.

In (a), the Python unit invokes a C function through ctypes. A least-privilege violation [47] happens at c3 when the external input (Extdata) retrieved at p3 reaches there via p5, allowing an attacker to gain unauthorized access to any file specified. Single-language analyses would either stop at the language (Python) boundary, or make a conservative assumption that the data retrieved at p6 causes a vulnerability as well (hence leading to excessive imprecision).

In (b), the interfacing is realized via Python extension (bidirectional). Sensitive data retrieved at p3 (source) may leak at c9 (sink) along the inter-language flow (red lines). Crosslanguage call graph construction hence a holistic flow analysis is necessary to find this vulnerability, which cannot be achieved by separately analyzing each language unit.

In (c), the interfacing mechanism is the same as in (b). Yet different from (b), here the two C function invocations are implicit—note that __enter__ and __exit__ are a result of the with construct at p9, and they are visible only to the Python interpreter but not to a static analyzer. Calls to these two functions can only be captured at runtime. Dynamic analysis is necessary for finding the information leak here.

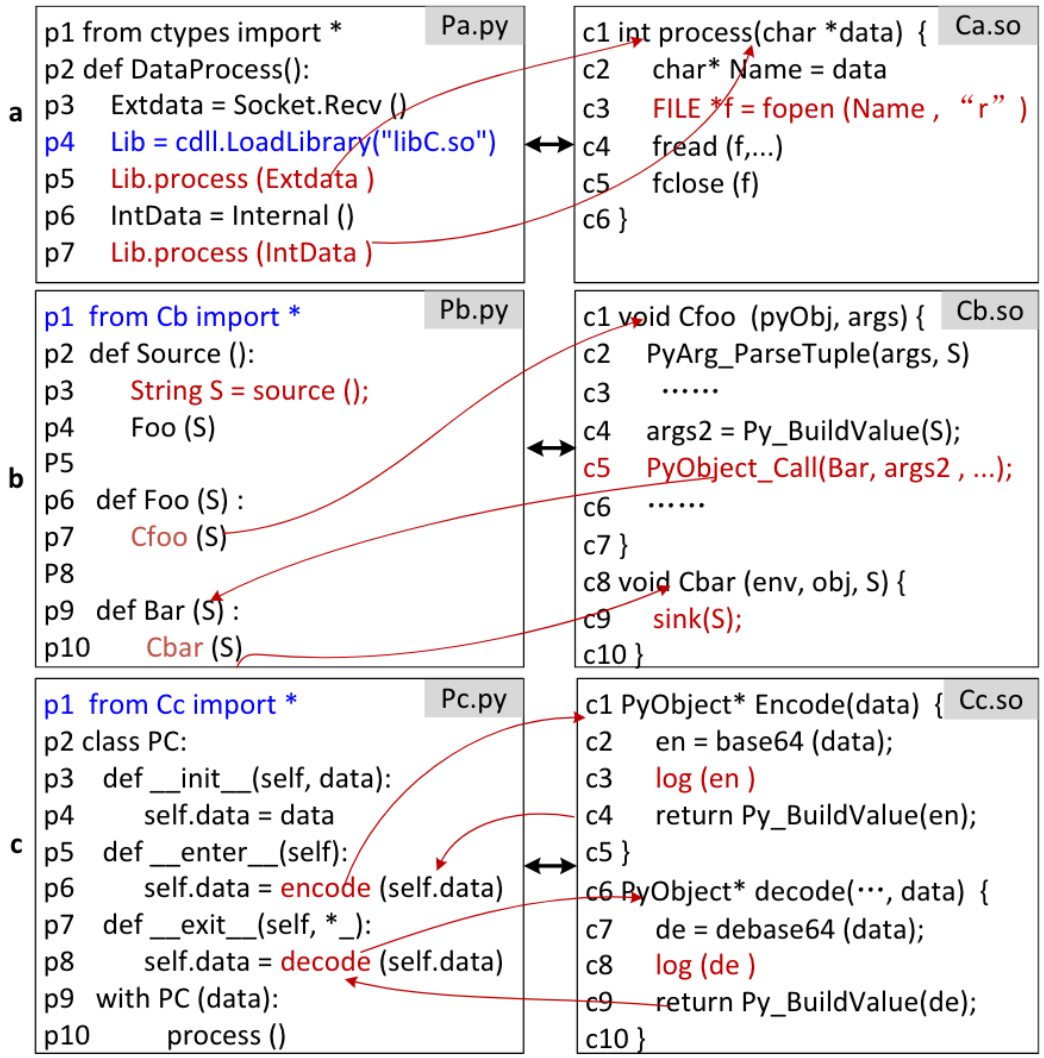

Figure 1: Example Python-C interfaces and vulnerabilities.

说明/激励的例子。图 1 描述了三个跨语言漏洞案例，每个案例在 Python 和 C 之间都有不同的接口机制。pi 和 ci 表示行号。 i 分别在 Python 和 C 单元中。

- 在 (a) 中，Python 单元通过 ctypes 调用 C 函数。当在 p3 检索到的外部输入（Extdata）通过 p5 到达 c3 时，在 c3 发生最小权限违规 [47]，从而允许攻击者未经授权访问指定的任何文件。单语言分析要么停止在语言（Python）边界，要么保守地假设在 p6 检索到的数据也会导致漏洞（因此导致过度不精确）。
- 在（b）中，接口是通过 Python 扩展（双向）实现的。在 p3（源）检索到的敏感数据可能会沿着跨语言流（红线）在 c9（接收器）泄漏。跨语言调用图构建因此需要进行整体流分析来发现此漏洞，而这无法通过单独分析每个语言单元来实现。
- 在（c）中，接口机制与（b）中的相同。然而与 (b) 不同的是，这里的两个 C 函数

调用是隐式的——注意 __enter__ 和 __exit__ 是 p9 处 with 构造的结果，它们仅对 Python 解释器可见，而对静态分析器不可见。只能在运行时捕获对这两个函数的调用。动态分析对于发现这里的信息泄漏是必要的。

```
    These examples not only illustrate how security
vulnerabilities may happen at and across language interfacing
hence the need for cross-language, holistic analyses, but also
justify our dynamic analysis based design for POLYCRUISE.
```

这些示例不仅说明了安全漏洞如何在语言接口处和跨语言接口发生，因此需要跨语言、整体分析，而且证明了我们基于动态分析的 POLYCRUISE 设计的合理性。

3. The POLYCRUISE Approach

This section describes our technical approach. We start with an overview (§3.1) of POLYCRUISE and then elaborate its two high-level phases: static analyses and instrumentation (§3.2) and online dynamic analysis (§3.3).

本节介绍我们的技术方法。我们从 POLYCRUISE 的概述（第 3.1 节）开始，然后详细阐述其两个高级阶段：静态分析和仪器（第 3.2 节）和在线动态分析（第 3.3 节）。

3.1 Approach Overview

An overview of POLYCRUISE is given in Figure 2. Three POLYCRUISE Inputs should be provided: (1) the multilingual program P under analysis, viewed as a collection of per-language code units, (2) user configuration X, including the lists of sources/sinks (given per language unit) as required by any DIFA and options for determining which security application plugins to apply, and (3) the set T of run-time inputs for P as required by any dynamic analysis.

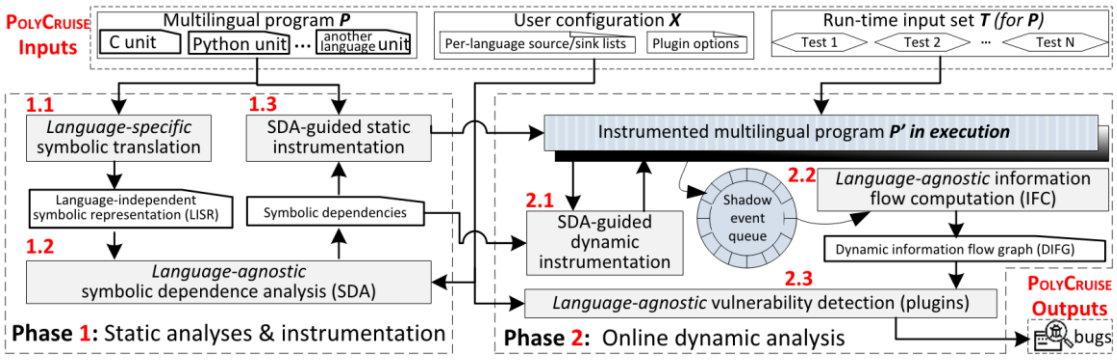

Figure 2: An overview of PolyCruise's architecture, including its inputs, major technical components, and outputs.

图 2 给出了 POLYCRUISE 的概述。应提供三个 POLYCRUISE 输入：(1) 正在分析的多语言程序 P，被视为每种语言代码单元的集合，(2) 用户配置 X，包括 source/sink 列表（按语言单位给出）任何 DIFA 的要求和用于确定应用哪些安全应用程序插件的选项，以及 (3) 任何动态分析所需的 P 的运行时输入集 T。

With these inputs, POLYCRUISE works in two main phases. In Phase 1, it first translates each language unit into a language-independent symbolic representation (LISR), from which a symbolic dependence analysis (SDA) computes symbolic dependencies in the unit by referring also to the source/sink list for the unit. The language independence of the LISRs enables the SDA to be language-agnostic. The resulting symbolic dependencies are then used to guide a static instrumentation step, informing it about the scope of necessary probing, for units written in a compiled language (e.g., C or Java). The main goal of this phase is to reduce the instrumentation (static or dynamic) scope hence the run-time overhead of the next phase. The rationale is that the symbolic dependencies (over)approximate all possible dynamic information flow between the given sources and sinks.

通过这些输入，POLYCRUISE 分为两个主要阶段。在第 1 阶段，它首先将每个语言单元转换为与语言无关的符号表示（LISR），符号依赖分析（SDA）通过引用该单元的源/汇列表从中计算单元中的符号依赖。 LISR 的语言独立性使 SDA 能够与语言无关。然后使用生成的符号依赖关系来指导静态检测步骤，告知它必要探测的范围，用于以编译语言（例如，C 或 Java）编写的单元。此阶段的主要目标是减少检测（静态或动态）范围，从而减少下一阶段的运行时开销。基本原理是符号依赖性（过度）近似于给定源和汇之间的所有可能的动态信息流。

> From the DIFG, a set of vulnerability detection plugins, each detecting one type of (e.g., data leak) vulnerabilities, continuously computes information flow paths between sources/sinks relevant to the vulnerability type and reports the vulnerabilities found as part of POLYCRUISE Outputs.

来自 DIFG，一组漏洞检测插件，每个检测一种类型的（例如，数据泄漏）漏洞，持续计算与漏洞类型相关的源/接收器之间的信息流路径，并将发现的漏洞作为 POLYCRUISE 输出的一部分报告。

3.2 Static Analyses/Instrumentation (Phase 1)

> To collect the run-time data required by its DIFA, POLYCRUISE needs to instrument the given program P. In accordance with the overarching principle (§1) of our design, the instrumentation should probe for harvesting the run-time data in a language-independent manner so as to enable languageagnostic dynamic analysis, while only relying on minimal language-specific analyses. Following this rationale, the first phase works in three major steps as elaborated below.

为了收集其 DIFA 所需的运行时数据，POLYCRUISE 需要对给定的程序 P 进行检测。根据我们设计的总体原则（第 1 节），检测应该探索以独立于语言的方式获取运行时数据，以便启用与语言无关的动态分析，同时仅依赖于最少的特定于语言的分析。根据这一基本原理，第一阶段分三个主要步骤进行，如下所述。

3.2.1 Symbolic Translation (Step 1.1)

```
    To minimize language-specific analyses, POLYCRUISE aims at a
language-agnostic analysis (i.e., SDA) that computes uniform
instrumentation-guiding information (i.e., symbolic
dependencies). This is achieved through a preprocessing step that
translates each language unit of P into its languageindependent
symbolic representation (LISR) form.
```

为了最大限度地减少特定于语言的分析，POLYCRUISE 旨在进行与语言无关的分析（即 SDA），该分析计算统一的仪器指导信息（即符号依赖性）。这是通过一个预处理步骤来实现的，该步骤将 P 的每个语言单元转换为其与语言无关的符号表示 (LISR) 形式。

```
    A program P is a sequence G* of global symbols, followed by a
sequence F* of function symbols—a symbol is simply an identifier.
A global symbol consists of an expression e. A function symbol F
has the return type τ, function name f , a sequence x* of
parameters, and a sequence of statements S*. The return tag τ
only indicates whether f returns a symbol, since the return type
does affect our symbolic dependence computation. A statement S is
of one of three kinds: a line statement [x =]e*, a function call
statement [x =] f (e*)—return value could be none thus is
optional, and a return statement return e. An expression e is of
one of three kinds: a symbol x, a constant C, and ε. A return tag
τ is a general type T or ε. ε denotes an empty string. As LISR
serves for data dependence approximation hence only needs to
capture variable definitions (defs) and uses, other common
language syntax elements (e.g., operators) are dropped in LISR.
```

程序 P 是一个全局符号序列 G*，后面跟着一个函数符号序列 F*——一个符号只是一个标识符。全局符号由表达式 e 组成。函数符号 F 具有返回类型 τ、函数名称 f、参数序列 x* 和语句序列 S*。返回标签 τ 仅指示 f 是否返回符号，因为返回类型确实会影响我们的符号依赖性计算。语句 S 是以下三种中的一种：行语句 [x =]e*，函数调用语句 [x =] f (e*)——返回值可以是无，因此是可选的，返回语句返回 e。表达式 e 是以下三种之一：符号 x、常数 C 和 ε。返回标签 τ 是一般类型 T 或 ε。 ε 表示一个空字符串。由于 LISR 用于数据依赖近似，因此只需要捕获变量定义（defs）和使用，其他常见的语言语法元素（例如，运算符）在 LISR 中被删除。

The detailed LISR translation process is given in Algorithm 1. As defined above, LISR considers three types of code entities: statement, global, and function. The algorithm translates each of these entities according to its type. Specifically, a global symbol is extracted from a global entity (line 5). For a function definition entity (line 7), the LISR translator symbolizes the definition as LISR expression. In the translation of a statement (line 8-21), if the statement's type is call (line 9), the return value is translated into the left symbol as a definition if it exists. Then, the function call is symbolized as the right symbol; if the statement's type is return (line 15), the symbol is extracted to formulate a LISR return statement; for other statements, the translator symbolizes the definitions as left symbols and the uses as right symbols (line 19-21).

Algorithm 1: Translate a given code entity to LISR

Input: $\mathbb{E}$: a given code entity of one of the three types: *global/function/statement*
Output: S_{lisr}: the LISR of $\mathbb{E}$

```
1  Function translate2LISR (𝔼)
2      S_lisr ← None;
3      𝕋_e ← getEntityType (𝔼);
4      if 𝕋_e == global then
5          S_lisr ← getGlobalSymbol (𝔼);
6      else if 𝕋_e == function then
7          S_lisr ← getFunctionSymbol (𝔼);          // symbolize the function type
8      else if 𝕋_e == statement then
9          if 𝔼 == call then
10             Ret ← getDef (𝔼);
11             if Ret ≠ None then
12                 S_lisr ← getSymbol (Ret) + "=";   // get return because def exists
13             Call = getCallSymbol (𝔼);            // symbolize the function call
14             S_lisr ← S_lisr + Call
15         else if 𝔼 == return then
16             Use ← getUse (𝔼);
17             S_lisr ← "Return " + getSymbol (Use);
18         else
19             Use ← getUse (𝔼);
20             Def ← getDef (𝔼);
21             S_lisr ← getSymbol (Def) + "=" + getSymbol (Use);
22     return S_lisr
```

详细的 LISR 翻译过程在算法 1 中给出。如上所述，LISR 考虑了三种类型的代码实体：语句、全局和函数。该算法根据其类型翻译这些实体中的每一个。具体来说，从全

局实体中提取全局符号（第 5 行）。对于函数定义实体（第 7 行），LISR 转换器将定义符号化为 LISR 表达式。在语句的翻译中（第 8-21 行），如果语句的类型是调用（第 9 行），则返回值被翻译为左符号作为定义（如果存在）。然后，函数调用被符号化为正确的符号；如果语句的类型是 return（第 15 行），则提取符号以制定 LISR return 语句；对于其他语句，翻译器将定义表示为左符号，将用途表示为右符号（第 19-21 行）。

```
1       Source Code          |      LISR
2   typeA gValue             |    gValue
3   Output(typeB& arg)       |    Output(arg)
4       print (arg)          |        print(arg)
5                            |
6   typeB Foo(typeB N)       |    T Foo(N)
7       typeB V := 1         |        V = C
8       typeB& S := V        |        S = V
9             V := N         |        V = N
10      while N != 0:        |             N
11          V := V * N       |        V = V,N
12          N := N - 1       |        N = N,C
13      Output (S)           |             Output(S)
14      return S             |        return S
```

Figure 3: An illustration of the symbolic translation.

```
    As an example, Figure 3 shows the resulting LISR (right
column) of the symbolic translation for a language unit (left
column). The global variable gValue of type typeA at Line 2 is
translated to a global symbol gValue. At Line 3, the original
statement is translated to a function symbol Out put(arg), and
the next to a call statement without a return value. The original
statement at Line 10 is translated to a line statement without
left value, so on and so forth.
```

例如，图 3 显示了语言单元（左列）符号翻译的结果 LISR（右列）。第 2 行的 typeA 类型的全局变量 gValue 被转换为全局符号 gValue。在第 3 行，原始语句被转换为函数符号 Out put(arg)，接下来是没有返回值的调用语句。第 10 行的原始语句被翻译为没有左值的行语句，依此类推。

3.2.2 Symbolic Dependence Analysis (SDA) (Step 1.2)

```
Once the LISR is obtained for a language unit, the next step is
to compute the symbolic dependencies in the unit according to the
(language-independent) def/use symbols in its LISR.
```

一旦获得语言单元的 LISR，下一步就是根据其 LISR 中的（与语言无关的）def/use 符号计算单元中的符号依赖关系。

Definition. Given two statements S_i and S_j, S_j is symbolically dependent on S_i iff $\{U(S_i) \cap D(S_j)\} \cup \{D(S_i) \cap U(S_j)\} \neq \varnothing$, where U($S$) and D($S$) is the use and def set of statement S.

定义。给定两个语句 Si 和 Sj，Sj 象征性地依赖于 Si 当且仅当 $\{U(S_i) \cap D(S_j)\} \cup \{D(S_i) \cap U(S_j)\} \neq \varnothing$，其中 U(S) D(S) 是语句 S 的使用和定义集。

```
1     Source Code          |     LISR
2  typeA gValue            |   gValue
3  Output(typeB& arg)      |   Output(arg)
4      print (arg)         |       print(arg)
5                          |
6  typeB Foo(typeB N)      |   T Foo(N)
7      typeB  V := 1       |       V = C
8      typeB& S := V       |       S = V
9             V := N       |       V = N
10     while  N != 0:      |           N
11         V := V * N      |       V = V,N
12         N := N - 1      |       N = N,C
13     Output (S)          |           Output(S)
14     return S            |       return S
```

Figure 3: An illustration of the symbolic translation.

```
1      LISR              |  symbolic def-use pairs
2  gValue                |
3  Output (arg)          |
4      print(arg)        |  D[4]={},U[4]={arg}
5                        |
6  T Foo(N)              |
7    V = C               |  D[7] ={V},U[7] ={C}
8    S = V               |  D[8] ={S},U[8] ={V}
9    V = N               |  D[9] ={V},U[9] ={N}
10       N               |  D[10]={ },U[10]={N}
11   V = V,N             |  D[11]={V},U[11]={V,N}
12   N = N,C             |  D[12]={N},U[12]={N,C}
13       Output(S)       |  D[13]={},U[13]={S}
14   return S            |  D[14]={},U[14]={S}
```

Figure 5: Symbolic defs/uses for the example of Figure 3.

The symbolic defs and uses for the example of Figure 3 are given in Figure 5. Let S_i denote Line i and suppose (S_9, V) is a source. As we consider true/flow dependencies, we have $D(S_9) \cap U(S_{11}) \neq \varnothing$; when we consider anti-dependencies, we also have $U(S_8) \cap D(S_9) \neq \varnothing$. Thus, the symbolic dependence set of S_9 is computed as $\{S_8, S_{11}\}$.

对于图 3 中的例子的符号定义和使用在图 5 中给出。设 Si 表示行 i，设（s9，V）是一个 source。当我们考虑 true/flow dependence 关系时，我们有 $D(S_9) \cap U(S_{11}) \neq \varnothing$，当我们考虑反向依赖，我们也有$U(S_8) \cap D(S_9) \neq \varnothing$。因此，S9 的符号依赖集计算为{S8，S11}。

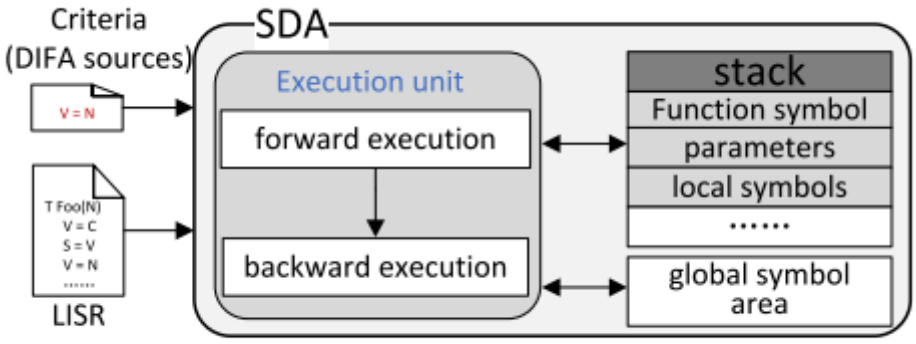

Figure 4: An overview of the symbolic dependence analysis.

Dependence computation. Based on the definition, symbolic dependencies in a language unit are computed as shown in Figure 4. The SDA takes a set of criteria (information sources) predefined for a language unit and the unit's LISR. It then symbolically executes the unit, both forward and backward, to capture true/flow and anti dependencies, respectively. To that end, it uses two memory regions: a stack including the local symbol area (LSA) to keep track of local symbols, and a global symbol area (GSA) to keep track of global symbols.

依赖计算。根据定义，如图 4 中展示的计算语言单元中的符号依赖关系。SDA 采用为一个语言单元和该单元的 LISR 预定义的一组标准（信息源）。然后它象征性地向前和向后执行该单元，分别捕获 true/flow dependence 和 anti-dependencies。为此，它使用两个存储区：一个堆栈包括本地符号区(LSA)来跟踪本地符号，另一个堆栈包括全局符号区(GSA)来跟踪全局符号。

Algorithm 2: Compute symbolic dependencies

Input: $\mathbb{E}_F$:set of entry functions, $\mathbb{C}$: predefined criteria (e.g., DIFA sources)
Output: $\mathbb{S}_s$:symbolic dependence set of $\mathbb{C}$

```
1  Function computeSD (E_F, C)
2      Q_F ← initQueue (E_F));                    // Q_F is a FIFO queue of functions
3      G_sym ← ∅;                                 // G_sym is the set of global symbols in the GSA
4      while Q_F ≠ ∅ do
5          F ← Q_F.pop();
6          Stack ← ∅;                             // (re)initialize the SDA stack for the current function F
7          SDS_F ← getSDS (F);                    // SDS_F is the symbolic dependence (SD) summary of F
8          computeSDoF (F, SDS_F, C, Stack, G_sym);   // compute SDs per function
9          if (gSym = getReachable (G_sym)) ≠ ∅ then
10             Ref = getRefer (gSym);             // get entry functions that use global symbols
11             Q_F.push(Ref);
12     S_s ← obtainStmt ();                       // get the statements symbolically dependent on C
13     return S_s
```

The main SDA algorithm is given in Algorithm 2 (as the computeSD routine). It takes the set of entry functions of target programs and the predefined criteria as inputs—an entry function is either the *main* function of an executable or a library interface exposed externally. Furthermore, the set of criteria is defined and customized by POLYCRUISE users. The algorithm starts with all entry functions being pushed into a FIFO queue $\mathbb{Q}_F$ (line 2); then, it (symbolically) executes these functions one by one with its symbolic dependence summary (*SDS*) through a subroutine computeSDoF.

SDA 的主要算法在算法 2 中给出（作为 computeSD 例程）。它以目标程序的一组入口函数和预定义的条件作为输入--入口函数要么是可执行文件的主函数，要么是外部公开的库接口。此外，一组标准由 PolyCruise 的用户定义和定制。算法以将所有入口函 数推入 FIFO 队列 QF（行 2）开始；然后，它（象征性地）通过子程序 ComputeSDof 用它的符号依赖摘要 (SDS) 逐个执行这些函数。

A function's *SDS* is a bitmap that indicates whether its return value or parameters are reachable from the given criteria through def-use-association (*DUA*) computation at a specific callsite of the function. For the example of Figure 3, if we define an 8-bit *SDS*, the *SDS* of the *Output* function at Line 13 will be computed as 01000000. The first bit 0 indicates that the return value is not reachable; the second bit 1 indicates the first parameter is reachable. The remaining bits are zero because this function has only one parameter.

函数的 SDS 是一个位图，它指出函数的返回值和参数在给定的标准下在特定函数调用点上的 def-use-Association(DUA)计算是否可达。如图 3 中示例所示，如果我们定义一个 8 位的 SDS，那么第 13 行的输出函数的 SDS 将被计算为 01000000。第一位 0 表示返回值不可达；第二位 1 标表示第一个参数是可达的。剩余位为零，因为此函数只有一个参数。

Algorithm 2: Compute symbolic dependencies

Input: $\mathbb{E}_F$:set of entry functions, $\mathbb{C}$: predefined criteria (e.g., DIFA sources)
Output: $\mathbb{S}_s$:symbolic dependence set of $\mathbb{C}$

```
1  Function computeSD (E_F, C)
2      Q_F ← initQueue (E_F));                    // Q_F is a FIFO queue of functions
3      G_sym ← ∅;                                 // G_sym is the set of global symbols in the GSA
4      while Q_F ≠ ∅ do
5          F ← Q_F.pop();
6          Stack ← ∅;                             // (re)initialize the SDA stack for the current function F
7          SDS_F ← getSDS (F);                    // SDS_F is the symbolic dependence (SD) summary of F
8          computeSDoF (F, SDS_F, C, Stack, G_sym);   // compute SDs per function
9          if (gSym = getReachable (G_sym)) ≠ ∅ then
10             Ref = getRefer (gSym);             // get entry functions that use global symbols
11             Q_F.push(Ref);
12     S_s ← obtainStmt ();                       // get the statements symbolically dependent on C
13     return S_s
```

For each entry function $\mathbb{F}$, the SDA stack is (re)initialized (line 6)—*GSA* is shared for all functions; then, its current *SDS* ($SDS_{\mathbb{F}}$) is retrieved via `getSDS` (line 7)—which can be initialized as -1 (by default) or a customized value given in the criteria. Once a global symbol in $\mathbb{F}$ is found (via `getReachable`) reachable from a criterion and inserted into *GSA* after one pass of symbolic execution of $\mathbb{F}$, all entry functions that reference the global symbol are pushed into $\mathbb{Q}_F$ again for recomputation to ensure consistency (line 11). When $\mathbb{Q}_F$ becomes empty, *SDA* obtains the set of all statements computed as symbolically dependent on the criteria as the algorithm's output.

对于每一个进入的函数 F，SDA 栈（重新）初始化——GSA 由所有函数共享；然后它的实时 SDS（SDSF）通过 getSDS（行 7）搜索——能够被初始化为-1（默认情况下）或者一个在标准中指定的值。一旦一个 F 中的全局符号被发现（通过 getReachable）从一个标准中可达，并在 F 的符号执行一次之后被插入到 GSA 中，所有引用全局符号的入口函数都被再次推入 QF 中进行重新计算，以确保一致性（行 11）。当 QF 为空时，SDA 获得所有语句的集合，这些语句的计算依赖于算法的输出条件。

Algorithm 3: SDA for each function

Input: $\mathbb{F}$:an entry function, $SDS_{\mathbb{F}}$:the (current) SDS of $\mathbb{F}$, *Stack*:the stack (see Figure 4), G_{sym}:the set of global symbols, $\mathbb{C}$:predefined criteria
Output: $SDS_{\mathbb{F}}$:the (updated) SDS of $\mathbb{F}$

```
1  Function computeSDoF (𝔽, SDS_𝔽, ℂ, Stack, G_sym)
2      L_sym ← initLocalSymb (SDS_𝔽);          // initialize the local symbol area
3      while True do
4          computeMain (𝔽, L_sym, SDS_𝔽, ℂ, Stack, 𝔾_sym);   // forward execution
5          StmtNum_F ← getReachableStmt ();
6          𝔽_r = reverseFunc (𝔽);
7          computeMain (𝔽_r, L_sym, SDS_𝔽, ℂ, Stack, 𝔾_sym);   // backward execution
8          StmtNum_r ← getReachableStmt ();
9          if StmtNum_F == StmtNum_r then
10             break;                          // having reached the fixed point
11     updateSDS (𝔽, SDS_𝔽);
```

The algorithm for computing the symbolic dependencies for each function is given in Algorithm 3. *SDA* first initializes the local symbol area (*LSA*) for the current function in terms of its *SDS* that indicates which formal parameters should be pushed into the *LSA* (line 2). After that *SDA* repeats the procedure of forward (line 4) and backward (line 7) computation until the number of the reachable statements remains unchanged, which means the algorithm reaches a fixed point and all possible statements have been collected. As a reminder, the backward computation shares the same procedure with the forward but takes the reverse order of statements.

算法 3 中给出了计算各函数符号依赖关系的算法。SDA 首先初始化当前函数的本地符号区域，它的 SDS 指出那些形式参数应该被推入 LSA（行 2）。在此之后，SDA 重复进行前进的过程（行 4）和向后（行 7）计算，直到可达语句数保持不变，即算法达到一个固定点（不动点），并且所有可能的语句都已收集完毕，提醒一下，反向计算与正向计算共享相同的过程，但采用相反的语句顺序。

Algorithm 4: The *Main* logic of SDA

Input: $\mathbb{L}_{sym}$:the set of local symbols for $\mathbb{F}$; others are same as Algorithm 3
Output: $SDS_{\mathbb{F}}$:the (updated) SDS of $\mathbb{F}$

```
1  Function computeMain (𝔽, 𝕃_sym, SDS_𝔽, ℂ, Stack, 𝔾_sym)
2      if 𝔽 ∈ Stack then
3          return SDS_𝔽;
4      Stack.push(𝔽);
5      foreach S_i in 𝔽 do
6          D[S_i], U[S_i] ← getDefUse (S_i);
7          if getType (S_i) == Line then
8              if isCriteria (ℂ, S_i) or (U[S_i] ∈ {𝕃_sym ∪ 𝔾_sym}) then
9                  if isGlobalSym (D[S_i]) then
10                     𝔾_sym.push(D[S_i])
11                 else
12                     𝕃_sym.push(D[S_i])
13         else if getType (S_i) == Call then
14             Callee, Sds_c ← initCallee (S_i);   // initial SDS: actual->formal args
15             if Callee != NULL then
16                 Sds_c ← computeSDoF (Callee, Sds_c, ℂ, Stack, 𝔾_sym);
17             else
18                 Sds_c ← -1;
19             updateSym (𝕃_sym, Sds_c, S_i);   // update 𝔽's 𝕃_sym with the callee's SDS
20         else if getType (S_i) == Return then
21             if S_i ∈ {𝕃_sym ∪ 𝔾_sym} then
22                 setRetBit (SDS_𝔽);   // return to the callsite
23         if isReachable (S_i) == true then
24             𝕊_s.push(S_i);   // save the statements for instrumentation—will just probe there
25     Stack.pop(𝔽);
26     SDS_𝔽 ← SDS_𝔽 | summarize (𝕃_sym, 𝔽);   // update 𝔽's SDS with its local symbols
```

The details of computeMain are given in Algorithm 4, where *SDA* symbolically executes the input function. As initialization, *SDA* pushes the function into the stack if it is not there yet. This step helps avoid recursive invocations, which would be redundant here as the function's *SDS* will remain unchanged when the inputs are fixed. Then, *SDA* processes each statement differently per its type as follows:

computeMain 的细节在算法 4 中给出，其中 SDA 象征性的执行输入函数。作为初始化，如果函数不在栈中，SDA 将函数压进栈中。这一步有助于避免多余的递归调

用，因为当输入固定时函数的 SDS 将保持不变。然后 SDA 按其类型对每个语句进行不同的处理，如下所示：

Line. If the statement's *Use* is in the criteria set, *LSA*, or *GSA*, *SDA* pushes the associated *Def* into the *LSA* or *GSA* so that the symbolic dependence propagates along the execution flow.

行。如果语句的使用是在标准集、LSA 或者 GSA 中，SDA 将关联的 Def push 进 LSA 或 GSA 中，以便符号依赖关系沿执行流传播。

Call. *SDA* handles *Call* statements to compute interprocedural dependencies. Before executing a callee, *SDA* first obtains the callee's definition and constructs the invocation context for it; the context contains the callsite and an initial *SDS* of the callee computed with the actual parameters. The initial *SDS* indicates which symbols of the parameters would flow into the callee, and *SDA* utilizes this information to initialize callee's *LSA*. If *SDA* fails to get the definition of the callee (e.g., a library function) and its initial *SDS* is non-zero, *SDA* conservatively sets the callee's *SDS* as -1, which means the return value and all parameters of the function are reachable from criteria. Otherwise, *SDA* invokes computeSDoF for the callee. After the execution of the callee, SDA updates *LSA* of the current function with the callee's *SDS*, which indicates the symbols flowing from the callee back to the function.

调用。SDA 处理调用语句以计算程序间依赖关系。在执行被调用方之前，SDA 首先获取被调用方的定义并为其构造调用上下文；上下文包含调用点和用实际参数计算的被调用方的初始 SDS。初始 SDS 指出哪些参数将流入被调用方，SDA 利用这些信息初始化被调用方的 LSA。如果 SDA 未能获得被调用方的定义（例如，库函数），并且其初始 SDS 为非 0，则 SDA 保守的将被调用方的 SDS 设置为-1，这意味着函数的返回

值和所有参数都可以从标准中到达。否则，SDA 将为被调用方调用 ComputesDof，执行完被调用方

后，SDA 用被调用方的 SDS 更新当前函数的 LSA，该 SDS 指出从被调用方返回函数的符号。

> **Return.** *SDA* checks the *Use* at a *Return* statement. The presence of *Use* in the *LSA* or *GSA* indicates that the return value of the current function is reachable from the criteria, hence *SDA* sets the return-bit of *SDS* to 1 in this case (line 22).

返回。SDA 在 return 语句中检查 use。在 LSA 或者 GSA 中 Use 的存在表明当前函数的返回值从条件中可达，隐私 SDA 在本例中将 SDS 的返回位设置为 1（行 22）。

> When executing a statement, *SDA* inserts it into a set if it is reachable (from the input criteria $\mathbb{C}$). When *SDA* finishes executing all statements of a function, it updates the function's *SDS* with its *LSA* before returning; this conservative treatment is necessary because the function may have output parameters.

当执行一条语句时，如果它是可达的，SDA 将其插入到一个集合中。当 SDA 执行完一个函数的所有语句后，它在返回之前用它的 LSA 更新该函数的 SDS；这种保守处理是必要的，因为函数有可能有输出参数。

3.2.3 SDA-Guided Static Instrumentation (Step 1.3)

> In the last step of Phase 1, POLYCRUISE performs static instrumentation of each compiled-language unit to insert probes for harvesting the run-time data underlying its DIFA. More specifically, the static instrumenter takes the set of statements that are computed by the SDA as symbolically dependent on any given source in the unit. Then, it probes for the run-time data only at those statements. As these data are collected at

runtime, we elaborate them as part of Phase 2. Without loss of generality, if the given program P does not include any compiled-language unit, this step is skipped.

在第阶段 1 的最后一步，polycruise 对每个编译语言单元执行静态插入，插入探针以获取其 DIFA 基础上的运行时数据。更具体的说，静态检测器将 SDA 计算的语句集作为单元中任何给定源的符号依赖。然后，它只在这些语句处探测运行时数据。由于这些数据收集于运行时监测的执行事件的定义。运行时，我们将他们作为阶段 2 的一部分进行阐述，如果给定的程序 P 不包括任何编译语言单元则跳过此步骤。

3.3 Online Dynamic Analysis (Phase 2)

POLYCRUISE performs its dynamic analysis *while* the (statically instrumented, if necessary) program P' executes. Thus, this phase does not exit until P' exits. This *online* design is justified by the need for handling long/continuously-running programs (e.g., those working as services). For those programs, an offline analysis may not even be feasible [9, 11]. In particular, Phase 2 works in three steps as elaborated below.

Polycruise 当程序 P'运行时（如果需要的话使用静态插桩）才执行动态分析。因此该阶段直到程序 P'退出才退出。这个在线设计以处理长时间/连续运行的程序的需要证明是合理的（例如，作为服务运行）。对于那些程序，离线分析甚至可能不可行。特别是阶段 2 工作分为如下三个步骤。

3.2.1 SDA-Guided Dynamic Instrumentation (Step 2.1)

For units written in interpreted languages, especially those having rich dynamic constructs, static instrumentation cannot fully probe for harvesting necessary run-time data. In fact, the code of such units may not even be completely visible to a static analysis (§2). POLYCRUISE addresses this challenge by dynamically instrumenting these units. As in Step 1.3, only the statements symbolically dependent on the given sources are probed. Both instrumentation steps also consistently probe for the same kind of run-time data, as detailed as follows.

对于用解释型语言编写的单元，特别是那些有丰富动态构造的单元，静态插桩不能完全探测以获取必要的运行时数据。事实上这些单元的 diamond 甚至可能对静态分析并不完全可见（第 2 段）。polysruise 同动态地检测这些单元来解决这个挑战。如在步骤 3 中，只探测依赖给定源的符号语句。两个插桩步骤还一致的探测相同类型的运行时数据，详细信息如下。

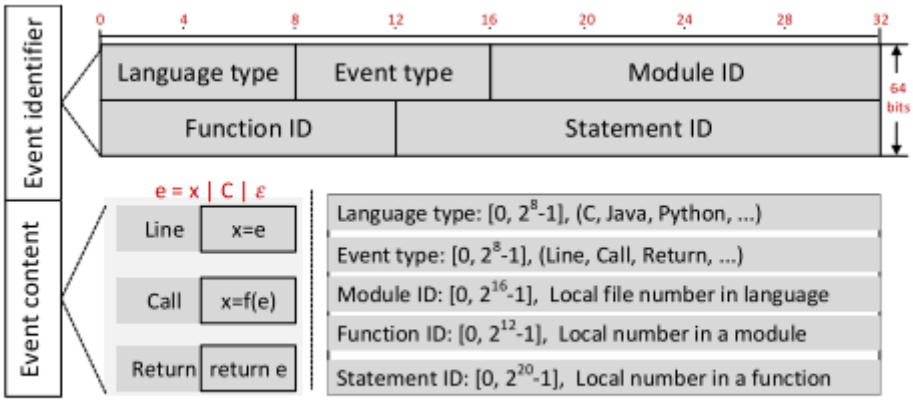

Figure 6: Definition of execution events monitored at runtime.

The run-time data to harvest are language-independent execution events that encode key (dynamic) information flow facts. Figure 6 gives the definition of these events. Each event starts with a 64-bit identifier followed by the event content. In particular, the event type within the identifier is used to guide the information flow computation (IFC, i.e., Step 2.2) to decode the event content. The other fields are not used in the IFC itself but enable mapping each event to the corresponding statement in the original program. This mapping is necessary for later producing information flow paths at source-code level that help understand vulnerability details.

要收集的运行时数据是编码关键（动态）信息流事实的独立于语言的执行事件。图 6 中给出了这些事件的定义。每一个事件以一个 64 位的标识符开始，后跟事件内容。特别的，标识符内的事件类型用于指导信息流计算(IFC ,即步骤 2.2)来解码事件内容。其他字段不在 IFC 本身使用，但允许将每个事件映射到原始程序中的相应语句。这种映射对于以后再源代码级别生成有助于理解漏洞细节的信息流路径是必要的。

> In accordance with the three types of LISR statements (§3.2.1), three types of execution events (i.e., *line*, *call*, and *return*) are probed for: $[x =]e^* \mid [x =]f(e^*) \mid return\ e$. An expression e can be a symbol x indicating a data type (e.g., `integer`), a constant C indicating the address of a reference/-pointer variable, or the ε. The event content stores the associated (LISR) statement itself. Importantly, for a variable of a non-primitive type, its address, rather than the symbol (literal), should be probed for. This address is used by the IFC to compute aliases hence information flow accurately.

根据三种类型的 LISR 语句（段 3.2.1），三种类型的执行事件（即行、调用和返回）被探测为：$[x =]e^*|[x =]f(e^*)|return\ e$。表达式 e 可以作为表示数据类型（例如，整数）的符号 x、表示引用/指针变量地址的常数 c 或 ε。事件内容存储被关联(LISR)语句本身。 重要的是，对于非原语类型的变量，应该探测其地址，而不是符号（literal）。这个地址被 IFC 用来计算别名，因此信息流准确。

3.3.2 Information Flow Computation (IFC) (Step 2.2)

> In this step, PolyCruise computes run-time information flow on the fly, by incrementally constructing a dynamic information flow graph (DIFG) from the execution events buffered in the shadow event queue (Figure 2). Given that real-world systems typically run in multiple threads, our design accounts for flow facts induced by multi-threading.

在这个步骤中，polycruise 通过从缓存在影子事件队列中的执行事件增量的构造动态信息流图来动态计算运行时信息流（图 2）。鉴于在真实世界的系统通常在多个线程中运

行，我们的设计考虑了由多线程引起的流事实。

> **Definition.** Suppose the P' execution consists of a set of threads $\Phi = \{\varphi_1, ..., \varphi_n\}$, which may share code and/or data (via global/shared variables). The DIFG for the execution is a directed graph $\mathbb{G}_\Phi = \langle G_\varphi{}^*, s^*, t^* \rangle$, $\varphi \in \Phi$, consisting of a set of thread graph G_φ each starting from an entry s and an exit t. A thread graph G_φ consists of a set of function graph G_f connected via interprocedural (call or return) edges, where each G_f consists of a set of nodes mapped one-to-one to the underlying event sequence $\vec{e}$. $\vec{e}_k$ denotes the k-th event in $\vec{e}$.

定义。假设 P'执行由一组线程$\Phi = \{\varphi_1, \ldots, \varphi_n\}$组成，这些线程共享代码和/或数据（通过全局/共享变量）。执行的 DIFG 是一个有向图$G_\Phi =< G_\varphi{}^*, s^*, t^* >, \varphi \in \Phi$，由一组线程图$G_\varphi$组成，每个$G_\varphi$从一个 entry s 开始和 t 结束。一个线程图G_φ由一个通过过程间（调用或返回）边连接的方法图G_f组成，每个G_f由一组一对一映射到底层事件序列$\vec{e}$的节点组成。$\vec{e}_k$表示$\vec{e}$中的第 k 个事件。

> Each DIFG node represents one of the execution events. Each edge, of one of four types below, represents a flow fact.

每个 DIFG 节点表示执行事件中的一个。以下 4 种类型之一的每条边表示一个流事实。

> - *Interthread control flow edge.* An edge of this type connects two nodes that represent two events $\vec{e}_i \in \varphi_m$ and $\vec{e}_j \in \varphi_n$, where $\vec{e}_i$ is a *call* event for thread creation, $\vec{e}_j$ is the first event in φ_n, and φ_m is the predecessor (creator) of φ_n.

- 线程间控制流边。一条这种类型的边连接两个节点，这两个节点表示两个事件$\vec{e}_i \in \varphi_m$和$\vec{e}_j \in \varphi_n$，其中$\vec{e}_i$是线程创建的调用事件，$\vec{e}_j$是φ_n的第一个事件，并且φ_m是φ_n的前驱（创建者）。

- *Intra-thread control flow edge.* This includes intra- and interprocedural (function graph) control flow edges. Given two events $\vec{e}_i$ and $\vec{e}_j$ in the same thread, $\vec{e}_i \rightarrow \vec{e}_j$ is an intraprocedural control flow edge if $\vec{e}_i$ and $\vec{e}_j$ happend in order in the same function. If $\vec{e}_i$ is a *call* event and $\vec{e}_j$ indicates the entry of the corresponding callee, $\vec{e}_i \rightarrow \vec{e}_j$ is an interprocedural (call) edge. If $\vec{e}_i$ is a *return* event and $\vec{e}_j$ is the *call* event corresponding to the callsite associated with that return, $\vec{e}_i \rightarrow \vec{e}_j$ is an interprocedural (return) edge.

- 线程内控制流边。这个包含过程内和过程间（方法图）控制流边。给定在同一线程中的两个事件$\vec{e}_i$和$\vec{e}_j$，如果$\vec{e}_i$和$\vec{e}_j$在同一个方法中按顺序发生$\vec{e}_i \rightarrow \vec{e}_j$是一个过程内调用控制流边。如果$\vec{e}_i$是一个调用事件并且$\vec{e}_j$表示对应被调用方的事件，$\vec{e}_i \rightarrow \vec{e}_j$是一个过程间（调用）边。如果$\vec{e}_i$是一个返回事件并且$\vec{e}_j$是对应与返回关联的调用点的调用事件，$\vec{e}_i \rightarrow \vec{e}_j$是一个过程间（返回）边。

  - *Interthread data flow edge.* Given two events $\vec{e}_i \in \varphi_i$ and $\vec{e}_j \in \varphi_j$ that happened sequentially in time, $\vec{e}_i \rightarrow \vec{e}_j$ is an interthread data dependence edge if $U(e_j) \subseteq D(e_i)$ while $U(e_j)$ includes a shared or global variable use and $D(e_i)$ includes the latest def of the variable. $U(\vec{e}_k)$ and $D(\vec{e}_k)$ denotes the use and def set of the statement where $\vec{e}_k$ happened.

- 线程间数据流边。给定在时间上连续发生的两个事件$\vec{e}_i \in \varphi_i$和$\vec{e}_j \in \varphi_j$，如果当$U(e_j)$包含一个共享或全局变量使用并且$D(e_i)$包含此变量的最新定义时$U(e_j) \subseteq D(e_i)$，$\vec{e}_i \rightarrow \vec{e}_j$是一个线程间数据依赖边。$U(\vec{e}_k)$和$D(\vec{e}_k)$表示$\vec{e}_k$发生的位置使用和定义语句集合。

- *Intra-thread data flow edge.* This includes intra- and interprocedural (function graph) data flow edges. Given two events $\vec{e}_i$ and $\vec{e}_j$ that happened in the same thread while satisfying $U(e_j) \subseteq D(e_i)$, $\vec{e}_i \rightarrow \vec{e}_j$ is an intra-procedural data flow edge if $\vec{e}_i$ and $\vec{e}_j$ happened sequentially in the same function. If $\vec{e}_i$ is a *call* event and $\vec{e}_j$ happened within the corresponding callee, $\vec{e}_i \rightarrow \vec{e}_j$ is an interprocedural (call) edge. If $\vec{e}_i$ is a *return* event and $\vec{e}_j$ is the *call* event corresponding to the callsite associated with that return, $\vec{e}_i \rightarrow e_j$ is an interprocedural (return) edge.

- 线程间数据流边。这个包含过程内和过程间（方法图）数据流边。给定两个当满足 $U(e_j) \subseteq D(e_i)$时在同一线程内发生的事件 $\vec{e}_i$和 $\vec{e}_j$，如果 $\vec{e}_i$和 $\vec{e}_j$在同一个方法中按顺序发生，$\vec{e}_i \rightarrow \vec{e}_j$时一个过程内数据流边。如果 $\vec{e}_i$是一个 call 事件并且 $\vec{e}_j$ 包含对应的被调用者，$\vec{e}_i \rightarrow \vec{e}_j$是一个过程间(call)边。如果 $\vec{e}_i$是一个 return 事件并且 $\vec{e}_j$是对应到与 return 关联的调用点 call 事件，$\vec{e}_i \rightarrow \vec{e}_j$是一个过程间（return）边。

Algorithm 5: Construct DIFG incrementally

Input: $\vec{e}_k$: an execution event $\in \vec{e}$, φ: the no. of thread containing $\vec{e}_k$
Output: $\mathbb{G}_\Phi$: the updated DIFG

```
1  Function constructDIFG (e⃗_k, φ)
2      event ← decodeEvent (e⃗_k);
3      G_φ ← getOrAddGraph (event, φ);
4      G_f ← getFuncDIFG (G_φ, event);
5      CurNode ← newNode (G_φ, event);
6      PreNode ← getPreNode (G_φ);
7      if G_f == NULL then
8          G_f ← addFuncDIFG (G_φ, event);
9          if PreNode == NULL then
10             createItcfgEdge (𝔾_Φ, G_φ)          // compute interthread control flow
11     else
12         TailNode_f ← getTail (G_f);
13         createCfgEdge (TailNode_f, CurNode);// add intra-procedural control flow
14         while TailNode_f != NULL do
15             if U(CurNode) ⊆ D(TailNode) then
16                 createDDEdge (TailNode_f, CurNode);
17                 break;                          // done computing intra-procedural data flow
18             TailNode_f ← TailNode_f->previous
19         if IsCallEvent(e⃗vent) then
20             G_callee ← getFuncDIFG (G_φ, event);
21             createCfgEdge (G_f, G_callee);      // compute interprocedural control flow
22             createDDEdge (G_f, G_callee);       // compute interprocedural data flow
23         computeGlobalDD (𝔾_Φ, CurNode);// data flow due to global/shared variables
```

DIFG construction. Algorithm 5 outlines the major steps for constructing the DIFG incrementally—updating it once per event. Upon receiving an event $\vec{e}_k$, the algorithm first decodes the event content, gets/updates the enclosing thread graph G_φ (i.e., for the thread in which $\vec{e}_k$ occurred), and retrieves the enclosing function graph (Lines 2–4). It then allocates a new

node *CurNode* for this event and gets its predecessor in G_φ (Lines 5–6). If G_f for $\vec{e}_k$ does not exist, meaning $\vec{e}_k$ is the first event of the current function, a new G_f would be created; this also indicate $\vec{e}_k$ is the first event of the current thread, thus an interthread control flow edge would be created also (Lines 7–10). If G_f already exists, the intra-procedural control and data flow edges would be first added if any (Lines 12–17); if moreover $\mathbb{E}$ is a *call* event, the DIFG of the callee G_{callee} would be obtained, and interprocedural control and data flow edges would be added (Lines 19–22). In the end, if $\mathrm{U}(\vec{e}_k)$ contains references to shared or global variables, global data dependence would be computed and added (Line 23).

DIFG 构造。算法 5 概述了以增量方式构造 DIFG 的主要步骤——每个事件更新一次。在接收到事件$\vec{e}_k$时，算法首先解码事件内容，获取/更新封闭线程图G_φ（即对于发生$\vec{e}_k$的线程），并检索封闭函数图（行 2-4）。然后它给这个事件分配一个新的节点 CurNode 并且在G_φ中获取它的前驱（行 5-6）。如果$\vec{e}_k$的G_f不存在则意味着$\vec{e}_k$是当前函数的第一个事件，一个新的G_f将会被创建；这也表明$\vec{e}_k$是当前线程的第一个事件，因此一个线程间控制流边也将会被创建并且数据流边将会被创建（行 7-10）。如果G_f已经存在，过程内控制和数据流边将会被首先添加（如果有）（行 12-17）；此外，如果 E 是 call 事件，被调用者的 DIFG G_{callee}将会被获得并且过程间控制和数据流边将会被添加（行 19-22）。最后，如果$U(\vec{e}_k)$包含对共享或全局变量的引用，全局数据依赖将会被计算和添加（行 23）

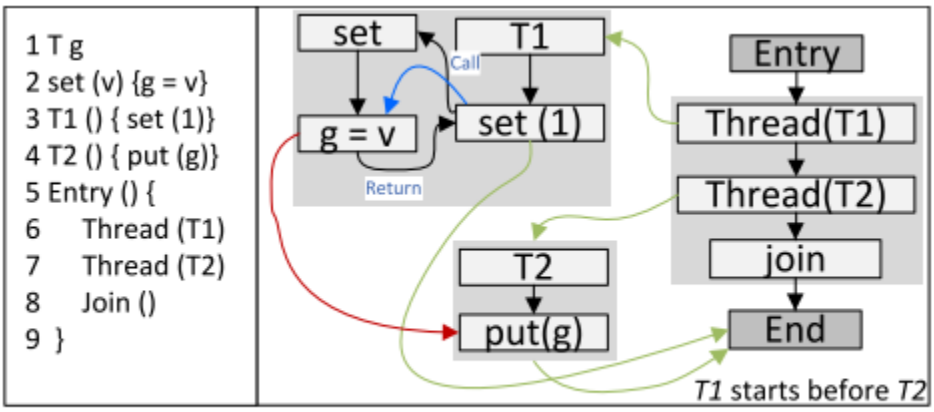

Figure 7: An example DIFG (right) for a program (left).

Figure 7 shows the DIFG (right column) of an example program (left column). The example code contains a global variable g, a main thread, and two sub-threads where we assume thread $T1$ starts before $T2$. The *DIFG* includes three sub-graphs corresponding to the three threads. The green (arrowed) lines represent the interthread control flow edges, while the red one is the interthread data flow edge because $T2$ reads g after the write/def of g in $T1$. The blue edge indicates an interprocedural data flow edge between the main procedure of $T1$ and the function *set*. The remaining black lines represent control flow edges in individual threads.

图 7 展示了示例程序（左列）的 DIFG（右列）。示例代码包含一个全局变量 g，一个主线程和两个子线程，我们假设 T1 在 T2 之前开。DIFG 包含 3 个子图对应三个线程。绿色（箭头）线表示线程间控制流边，红色线是线程间数据流边，因为 T2 读取 g 在 T1 写/定义 g 之后。蓝色边表示 T1 主进程和 set 方法之间的过程间数据流边。剩余的黑色线表示各个线程中的控制流边。

3.3.3 Vulnerability Detection (plugins) (Step 2.3)

> The dynamic information flow computed in Step 2.2 can support the general source-sink problem. To show the practical usefulness of our technique, we focus on vulnerability detection based on the DIFA in PolyCruise. According to the various sources and sinks relevant to various information flow security vulnerabilities, different detector plugin works by computing the information flow paths between those sources and sinks through a traversal (i.e., a reachability analysis) on the (most recently updated) DIFG. As the DIFG is incrementally constructed/updated, vulnerabilities are also reported as (security) bugs (Figure 2) in an incremental manner.

步骤 2.2 中的动态信息流计算能够支持一般的 source/sink 问题。为了展示该技术的实用性，我们重点研究了 polycruise 中基于 DIFA 的漏洞检测。根据各种信息流安全漏洞相关的 source 和 sink，不同的检测器插件通过在（最新的）DIFG 上遍历（即可达性分析）计算这些 source 和 sink 之间的信息流路径来工作。由于 DIFG 是增量构造/更新的，漏洞也被以增量的方式报告为（安全）bug（图 2）。

4 Implementations and Limitations

> We have implemented PolyCruise for Python-C programs and seven plugins each detecting one type of information flow vulnerabilities. More details are in Appendix A.

我们已经为 Python-C 程序实现了 polycruise 和 7 个插件，每个插件检测一种类型的信息流漏洞。更多的细节见附录 A。

LISR translations. To support Python-C programs, we implemented the LISR translator for C on top of LLVM [30], and that for Python by reusing the code normalization module of PyPredictor [72] but with significant enhancement. Specifically, the translator for C walks through the LLVM intermediate representation (IR) produced by the compiler frontend (Clang) while translating the IR code to LISR, following the syntax described in §3.2.1. In the LISR translator for Python, we reused the code normalization module of PyPredictor, which translates Python sources into their static single assignment (SSA) form under Python 2.7 and generates a corresponding abstract syntax tree (AST). We upgraded the tool to support higher versions of Python (3.6+) and translated the source with SSA form into LISR on top of AST. For Python code that uses object-oriented programming, we translated each class member *M* in the form of *class.M*.

LISR 翻译。为了支持 Python-C 程序我们在 LLVM 上实现了 LISR 翻译器，并通过重用 PyPredictor 的代码规范化模块来实现 Python 的代码规范化，但有显著增强。具体来说，C 语言的编译器将 IR 代码翻译成遵循段 3.2.1 中描述的语法的 LISR 时会遍历编译器前端（clang）生成的 LLVM 中间表示（IR）。在对 Python 的 LISR 翻译中，我们重用 PyPredictor 代码规范化模块，该模块将 Python 源代码翻译成 Python 2.7 下的静态单赋值（SSA）形式，并生成了对应的抽象语法树（AST）。我们升级了该工具以支持更高版本的 Python（3.6+），并将源代码与 SSA 表单转换为 AST 之上的 LISR。对于使用面向对象编程的 Python 代码，我们将每个类成员 M 转换为 class.M 的形式。

Symbolic dependence analysis. We implemented SDA as a simplified symbolic execution engine with C++ since no branches exist in the LISR of the two language units except for function invocations. SDA aims to compute all possible statements reachable from the criteria in each language unit. Hence, to ensure full analysis of data flow across different language units without compromising the overall scalability of PolyCruise, we did a two-level SDS implementation for entry functions in different language units.

符号依赖分析。我们使用 C++实现了 SDA 作为一个简化的符号执行引擎因为除了函数调用这两种语言单元的 LISR 中不存在分支。SDA 的目标是计算从每个语言单元中的标

准所有可能可达的语句。因此为了确保在不损害 PolyCruise 总体可伸缩性的情况下对跨不同语言单元的数据流进行全面分析，我们为不同语言单元的入口函数做了两级 SDS 实现。

> At the first level, we adopt a conservative computation that assumes all parameters of entry functions are reachable from the criteria. Specifically, for each language unit, SDA computes invocation relationship among functions; then SDA considers the functions invoked by no other functions as entry functions and initializes their SDSs as -1. Such an implementation would result in redundant instrumentation but cover all possible data flow paths across languages. The POLYCRUISE users can define the second level SDS; they can customize the SDS for each language interface through configuration. Only the data flow paths that users are interested in will be covered. Guided by these SDSs and user-defined criteria, the results of SDA can cover all possible statements.

在第一级，我们采用保守计算，假设入口函数的所有参数都可以从标准中可达。具体来说，对于每个语言单元，SDA 计算函数间的调用关系； 然后 SDA 将没有其他函数调用的函数视为入口函数，并将它们的 SDS 初始化为-1。这样的实现将导致冗余的检测，但涵盖了跨语言的所有可能的数据流路径。PolyCruise 的用户可以定义第二级 SDS；他们可以通过配置为每种语言接口定制 SDS。只有用户感兴趣的数据流路径才会被覆盖。在这些 SDS 和用户定义的标准的指导下，SDA 的结果可以涵盖所有可能的语句。

Instrumentation. With the SDA results, we implemented an LLVM pass [30] to instrument the C sources statically, partly reusing our whole-program C analysis tool PCA [37]. The execution event is constructed following the format defined in Figure 6. For Python components, we utilized a Python built-in tracing API (`sys.settrace`) for dynamic instrumentation based on the SDA results. Combining the AST generated earlier and dynamic stack information, we obtain details of statements during runtime and construct the events following the syntax. For simplicity, we implemented the tracing module as a C library such that the two kinds of language units can integrate directly.

插桩。利用 SDA 的结果，我们实现了一个 LLVM pass[30]对 C source 进行静态测试，部分重用我们的整个程序 C 分析工具 PCA[37]。执行事件按照图 6 中定义的格式构造。对于 Python 组件，我们使用 Python 内置跟踪 API（sys.settrace）进行基于 SDA 结果的动态检测。结合早期生成的 AST 和动态堆栈信息，我们在运行时获取语句的详细信息，并按照语法构造事件。为了简单起见，我们将跟踪模块实现为一个 C 库，这样两种语言单元就可以直接集成。

Run-time monitoring. The run-time monitors, for both in-memory tracing and event buffering, are implemented as a C library that can be linked to foreign components of mainstream languages (e.g., Python, Java, Ruby). Accordingly, the probes for run-time events are instrumented as simple function calls, a common approach to instrumentation in dynamic analysis [12, 19]. The shadow event queue is implemented as a bidirectional circulation queue initialized in shared memory.

运行时监控。用于内存跟踪和事件缓冲的运行时监视器被实现为一个 C 库，可以链接到主流语言的外部组件（例如，Python、Java、Ruby)。因此，对运行时事件的探测被检测为简单的功能调用，这是动态分析中检测的一种常见方法[12, 19]。影子事件队列实现为在共享内存中初始化的双向循环队列。

Online dynamic analysis. To support multi-threaded executions, we tagged each execution event with the corresponding thread id. Also, the shadow event queue is implemented in a thread- and process-safe manner to avoid undesirable interference with the original execution of real-world systems. To minimize the run-time overhead of the online analysis, we implemented a high-performance memory database using hash algorithms and scalable memory pools.

To enable evaluation and practical security applications of POLYCRUISE, we have implemented seven plugins for detecting seven types of vulnerabilities as defined in the CWE catalog [48] (i.e., sensitive data leak, control flow integrity, partial comparison, buffer overflow, integer overflow, and divide by zero). We also curated a set of sources/sinks for each plugin according to respective SDKs and libraries of Python and C. Our implementation allows users to flexibly customize the source/sink lists and plugin usage options (through the user configuration X, as shown in Figure 2).

在线动态分析。为了支持多线程执行，我们用相应的线程 ID 标记每个执行事件。此外，影子事件队列是以线程和进程安全的方式实现的，以避免与实际系统的原始执行发生不希望的冲突。为了最小化在线分析的运行时开销，我们使用哈希算法和可伸缩内存池实现了一个高性能的内存数据库。

为了使 PolyCruise 的评估和实际安全应用成为可能，我们实现了七个插件，用于检测 CWE 目录中定义的七种类型的漏洞[48]（即敏感数据泄漏、控制流完整性、部分比较、缓冲区溢出、整数溢出和除零）。我们还根据 Python 和 C 各自的 SDK 和库为每个插件策划了一组 source/sink。我们的实现允许用户灵活地定制 source/sink 列表和插件使用选项（通过用户配置 X，如图 2 所示)。

Supporting other languages. Real-world multilingual systems may use a great variety of combinations of different languages. Developing an analysis specific to each combination (e.g., Java-C) is not a scalable or desirable solution. Our design facilitates adding support for other languages (and according new combinations) into the current implementation.

Specifically, to support another language $\mathcal{L}$, only two parts need to be added: (1) a LISR translator for $\mathcal{L}$ and (2) an instrumenter that probes an $\mathcal{L}$ program for the kinds of run-time data needed (as described in §3.3.1) according to results of the (language-agnostic) SDA on the program's LISR. Implementing both parts can be eased by a basic analysis utility for $\mathcal{L}$ that supports syntactic parsing and instrumentation. As discussed earlier, if $\mathcal{L}$ is an interpreted/dynamic language, part (2) may need to be done at runtime.

支持其他语言。现实世界中的多语言系统可能使用多种不同语言的组合。开发特定于每个组合的分析（例如，Java-C）不是一个可伸缩的或理想的解决方案。我们的设计有助于在当前实现中添加对其他语言的支持（以及新的组合）。具体地说，要支持另一种语言 L，只需要添加两个部分：（1）用于 L 的 LISR 翻译器和（2）用于探测 L 程序所需运行时数据种类的插桩器（如段 3.3.1 所述）根据程序的 LISR 上的（语言不可知的）SDA 的结果。对于 L 来说，支持句法解析和插装的基本分析实用程序可以简化这两个部分。如前面讨论过的，如果 L 是解释/动态语言，那么第（2）部分可能需要在运行时完成。

Other VR support roles. As a cross-language DIFA, POLYCRUISE can work in other roles for vulnerability response (VR). For example, it may be enhanced by a test-input generator (e.g., fuzzer), which would produce additional, diverse run-time inputs to help POLYCRUISE potentially discover more vulnerabilities, as a result of increased coverage of the executions it analyzes.

On the other hand, POLYCRUISE can be leveraged to enhance a test-input generator (e.g., a greybox fuzzer) to generate test cases more effectively. For instance, POLYCRUISE can compute accurate data flow information between the source/sink pairs that are relevant to a taint-guided fuzzer. Then, the fuzzer may utilize the flow information to tune its evolution direction, informing seed scheduling and where to mutate and how. In particular, one may use POLYCRUISE to capture the dependency between seed inputs (sources) and branch variables or dangerous functions (sinks); the seeds on which more such sinks are dependent would be prioritized during seed selection and/or assigned with greater power.

其他漏洞响应支持规则。作为一个跨语言的 DIFA，PolyCruise 可以在其他漏洞响应角色中工作。例如，它可以通过测试输入生成器（例如 Fuzzer）得到增强，该生成器将产生额外的、双向的运行时输入，以帮助 PolyCruise 潜在地消除更多的漏洞，因为它分析的执行覆盖率增加了。

另一方面，可以利用 PolyCruise 生成测试输入生成器（例如 Greybox Fuzzer）来更有效地生成测试用例。例如，PolyCruise 可以计算与污点引导模糊器相关的 source/sink 对之间的准确数据流信息。然后，模糊器可以利用流信息来调整其进化方向，通知种子调度以及在哪里变异和如何变异。特别是可以使用 PolyCruise 捕捉种子输入（sink）和分支变量或危险函数（sink）之间的依赖关系；在种子选择期间将优先考虑更多这样的 sink 所依赖的种子和/或分配更大的权力。

Limitations. To deal with practical challenges with the online DIFA due to the complex and diverse interoperations (e.g., data encapsulation and conversion) between languages, the current implementation is field-insensitive. As a result, the DIFA results are not always precise and, accordingly, the vulnerabilities detected can be false positives.

In addition, while we managed to support most if not all language features of Python 3.x and validated so for Python 3.7, there might still be other features that are not well supported at this point. The evolution of such features may cause undesirable behaviors of POLYCRUISE. Also, explicit support for multi-process executions [10, 20] is not implemented yet.

Like a typical DIFA, POLYCRUISE only computes information flow exercised in the particular program executions considered. Thus, its ability to discover vulnerabilities is limited to those that are covered in the analyzed executions. This ability is additionally subject to the coverage of the sources and sinks considered and covered in the executions. Thus, from a general vulnerability detection's perspective, POLYCRUISE also suffers false negatives.

The current design of LISR focuses on capturing data dependence information (def/use). As a result, POLYCRUISE only computes explicit information flow hence would miss vulnerabilities solely induced by implicit information flow.

限制。由于语言之间复杂多样的互操作（如数据封装和转换），在线 DIFA 的实际应用面临着挑战，目前的实现是字段不敏感的。 因此，DIFA 结果并不总是精确的，因此，检测到的漏洞可能是假阳性。

此外，虽然我们设法支持了 Python 3.x 的大部分（如果不是全部的话）语言特性，并且对 Python 3.7 进行了验证，但目前可能还有其他特性没有得到很好的支持。这些特征的演化可能导致 Polycruise 的不良行为。此外，对多进程执行的显式支持[10, 20]尚未实现。

像典型的 DIFA 一样，PolyCruise 只计算特定程序执行中执行的信息流。因此，它

发现漏洞的能力仅限于分析的执行中包含的漏洞。 这一点还受制于执行中所考虑和涵盖的 source 和 sink 的范围。因此，从一般漏洞检测的角度来看，PolyCruise 也有假阴性的问题。

当前 LISR 的设计主要集中在捕获数据依赖信息（def/use）。因此，PolyCruise 只计算显式信息流，因此会遗漏完全由隐式信息流引起的漏洞。

5 PyCBench: A Multilingual Microbench

> Evaluating the precision and recall of a multilingual code analysis (e.g., DIFA) needs a multilingual benchmark suite that comes with ground truth for the analysis. Yet such a benchmark suite is not available, while curating the ground truth for large/complex, real-world programs may not be feasible.

评估多语言分析代码（例如 DIFA）的精确度和召回需要一个多语言基准测试套件，该套件附带用于分析的基本事实。然而，这样一个基准套件是不可用的，而为大型/复杂的现实世界程序策划基本事实可能是不可行的。

Table 1: Distribution of PyCBench by analysis features: general flow (GenF), global flow (GF), field sensitivity (FieldSen), ObjectSensitivity (ObjSen), dynamic invocation (DynInv).

Vulnerability Type	GenF	GF	FieldSen	ObjSen	DynInv	Total
Sensitive data leak	7	5	4	2	2	20
Code injection	1	1	0	0	0	2
Buffer overflow	1	2	2	1	1	7
Division by zero	1	0	0	1	0	2
Integer overflow	1	0	2	5	1	9
Incomplete comparison	3	1	0	0	0	4
Control-flow integrity	1	0	0	1	0	2
Total	15	9	8	10	4	46

We thus took the first step to manually create **PyCBench**, a microbench for multilingual program analysis, including but not limited to DIFA. As shown in Table 1, PyCBench consists of 46 benchmarks, covering seven common types of vulnerabilities (shown in the first column). We created the benchmarks for each vulnerability type by summarizing the patterns of those vulnerabilities in reference to the corresponding CWE descriptions [48]. These benchmarks were also selected purposely to cover five analysis features (listed in the first row) that we believe a multilingual code (static or dynamic) analyzer should consider handling.

因此，我们迈出了手动创建 PycBench 的第一步，这是一个用于多语言程序分析的微平台，包括但不限于 DIFA。如表 1 所示 Pycbench 由 46 个基准测试组成，涵盖七种常见的漏洞类型（如第一列所示）。 我们通过参考相应的 CWE 描述总结这些漏洞的模式，为每种漏洞类型创建了基准[48]。 这些基准测试也是有意选择的，以涵盖我们认为多语言代码（静态或动态）分析器应该考虑处理的五个分析特性（列在第一行）。

Currently, PyCBench only includes Python-C programs and one test for each. We will maintain and augment it by including more benchmarks and test cases while covering other language combinations. The current version of PyCBench is included in our open-source package for PolyCruise.

目前，Pycbench 只包括 Python-C 程序和每个程序的一个测试。我们将通过包含更多的基准和测试用例来维护和增强它，同时覆盖其他语言组合。Pycbench 的当前版本包含在我们的 PolyCruise 开源包中。

6 Evaluation

使用我们的 PolyCruise 实现（段 4），我们评价的方法是由以下四个问题指导的：

问题 1。就精度而言，Polycruise 有多有效？

问题 2。就其成本而言，PolyCruise 的效率如何？

问题 3。PolyCruise 能发现现实世界的漏洞吗？

问题 4。PolyCruise 与同行工具相比如何？

6.1 Experiment Setup

In addition to PyCBench (Table 1), we also evaluated PolyCruise against 12 real-world multilingual systems written in Python and C as primary languages and the original test cases. Table 2 summarizes these systems as our subjects (1st column), including the total code size (2nd column), percentage of code written in each language (3rd and 4th columns), and the number of tests used (last column).

Table 2: Real-world multilingual systems used as our subjects

Benchmark	Size (KLOC)	C/C++%	Python%	#Tests
Bounter [6]	3.5	48.2%	50.9%	190
Immutables [24]	5.9	55.0%	44.3%	152
Simplejson [26]	6.4	37.6%	59.8%	31
Japronto [53]	9.4	50.4%	48.2%	15
Pygit2 [40]	17.0	57.4%	44.6%	241
Psycopg2 [54]	27.5	50.8%	48.2%	198
Cvxopt [14]	56.0	60.8%	39.0%	78
Pygame [55]	207.0	54.3%	44.7%	324
PyTables [57]	219.8	52.1%	46.6%	6,355
Pyo [2]	259.1	50.8%	48.8%	51
NumPy [49]	919.7	36.1%	63.7%	16,002
PyTorch [61]	6,419.2	56.2%	35.2%	4,146

除了 PyCbench（表 1），我们还针对 12 个以 Python 和 C 作为主要语言编写的真实世界多语言系统和原始测试用例对 PolyCruise 进行了评估。表 2 将这些系统总结为我们的科目（第 1 列），包括总代码大小（第 2 列）、用每种语言编写的代码的百分比（第 3 列和第 4 列）以及使用的测试数量（最后一列）。

All of these 12 systems were downloaded from GitHub. For better benchmark representativeness, we developed a crawler to select multilingual projects that (1) has 1,000+ stars, which indicates popularity—a criterion used in prior work [62], (2) is developed mainly in Python and C (or C++), with each language's unit size accounting for 30%+ of the total project size, (3) is frequently updated and maintained, (4) has a rich set of test cases, indicating potentially good code quality, and

(5) comes with detailed documentation for quick installation, use, and testing. More details on each benchmark can be found via the link (1st column Table 2) to its repository.

这 12 个系统都是从 GitHub 下载的。为了更好的基准代表性，我们开发了一个爬虫来选择（1）有 1000+星的多语言项目，这表明流行度--这是以前工作中使用的标准

[62]，(2)主要用 Python 和 C（或 C++)开发，每种语言的单元规模占项目总规模的 30%+，(3)经常更新和维护，(4)有丰富的测试用例集，表明潜在的良好代码质量，以及（5）附带详细的文档，用于快速安装、使用和测试。关于每个基准测试的更多详细信息可以通过链接（表 2 第 1 列）找到其存储库。

> For a reasonable run-time coverage of each subject, we focused on integration tests provided with it. To run these tests under POLYCRUISE, we customized the Pytest [58] and Unittest [60] frameworks such that we can automate the test execution while performing the analyses in POLYCRUISE. We also developed a tool to automatically extract sources and sinks used as the default source/sink configuration. These additional utilities help increase the usability of POLYCRUISE and its capabilities for vulnerability discovery with respect to the run-time inputs available (i.e., without augmenting them).

为了对每个主题进行合理的运行时覆盖，我们将重点放在与之一起提供的集成测试上。为了在 PolyCruise 下运行这些测试，我们定制了 Pytest[58]和 Unittest[60]框架，这样我们可以在 PolyCruise 中执行分析的同时自动执行测试。我们还开发了一个工具来自动提取用作默认 source/sink 配置的 sink 和 source。这些额外的实用程序有助于提高 PolyCruise 的可用性及其针对可用运行时输入的漏洞发现能力（即，在不增加它们的情况下）。

6.2 Experimental Methodology

> We evaluated the effectiveness of POLYCRUISE on both PyCBench and the five least complex real-world benchmarks against all their test cases. For each benchmark and test, we traced the kinds of run-time data described earlier at every program statement. Then, by inspecting the trace while referring to the benchmark source code, we identified all possible paths between each of the predefined source/sink pairs. Using these paths as ground truth, we computed the precision and recall of POLYCRUISE for that benchmark and test. This precision is purely based on source-sink reachability. We further computed *precision under security context* by only considering exploitable paths as true positives.

我们在 PYCBENCH 和五个最不复杂的真实世界基准上对 PolyCruise 的所有测试用例进行了有效性评估。对于每个基准测试和测试，我们跟踪了前面在每个程序语句中描述的运行时数据的种类。然后，通过在参考基准源码的同时检查跟踪，我们确定了每个预定义 source/sink 对之间的所有可能路径。利用这些路径作为基准值，我们计算了 PolyCruise 对该基准和测试的精度和召回率。这种精度纯粹基于 source-sink 可达性。通过只考虑可利用路径为真，进一步提高了安全上下文下的精度。

> For each case in both the manual precision/recall evaluations, three of the authors each obtained results independently, followed by a confirmation process based on cross-validation. We confirmed a result only when all the three agreed on it.

对于两个手工精确/召回评估中的每一个案例，三个作者各自独立获得结果，然后基于交叉验证进行确认过程。只有当三个人都同意时，我们才确认一个结果。

> We assessed the efficiency and scalability of POLYCRUISE for its static and dynamic analysis part separately. In particular, for Phase 1, we measured the time and peak memory cost of the SDA, which overwhelmingly dominated the total cost of this phase—the costs of the other two steps are comparatively negligible, as both technically anticipated and empirically validated. Thus, for this phase, we only report SDA costs. For Phase 2, we compared the run-time slowdown on the same two cost measures (i.e., time and peak memory) among three versions of each benchmark against ten randomly selected tests: the original (*pure-version*), the instrumented with SDA guidance (*SDA-version*), and the entirely (every-statement) instrumented (*CMPL-version*). This procedure allowed for an in-depth evaluation of the efficiency impact of the SDA (Step 1.2), by comparing run-time slowdown among the three versions per benchmark.

我们分别对 Polycruise 的静态和动态分析部分的效率和可扩展性进行了评估。特别的，对于阶段 1 我们测量了 SDA 的时间和峰值内存成本，这在这一阶段的总成本中占据了压倒性的主导地位--正如技术预期和经验验证的那样，其他两个步骤的成本相对来说可以忽略不计。因此对于这一阶段，我们只报告 SDA 成本。阶段 2 我们比较了每

个基准测试的三个版本与随机选择的十个版本在相同的两个成本度量（即时间和峰值内存）上的运行时减慢测试：原始的（纯版本）、用 SDA 指南检测的（SDA 版本）和完全（每个语句）插桩的（CMPL 版本）。该程序允许对 SDA 的效率影响进行深入评估（步骤 1.2），通过比较每个基准的三个版本之间的运行时放缓。

> We are not aware of a DIFA working with Python-C programs. Thus, for peer comparison, we used the closest, state-of-the-art baselines we can find: PyPredictor [72], a Python analyzer, and libdft [27], a C/C++ dynamic taint analyzer. Neither is originally comparable, thus we did extra development/setup as detailed below. We did try to compare with several other tools that claimed to support cross-language analysis. Unfortunately, few of them are publicly available online (e.g., Truffle [29]) and actually usable.

我们不知道 DIFA 使用 Python-C 程序。因此，为了进行同行比较，我们使用了我们所能找到的最接近、最先进的基线：PyPredictor [72]，一个 Python 分析器，以及 libdft [27]，一个 C/C++动态污点分析器。两者都没有原生的可比性，因此我们做了额外的开发/设置，详见下文。我们确实尝试与其他几个声称支持跨语言分析的工具进行比较。 不幸的是，它们中很少有在网上公开获得（例如，Truffle[29])并且实际可用。

> **PyPredictor**. This is an analyzer of Python programs combining dynamic tracing and static symbolic execution to predict potential bugs (e.g., *AttributeError, TypeError*). It runs the given program against its given tests to collect modules involved in the execution. Then, it normalizes the code of these modules into their SSA form and executes the normalized program again to collect run-time traces. With these traces as inputs, it conducts predictive analysis based on symbolic execution to explore bugs on all possible execution paths.

PyPredictor。这是一个 Python 程序的分析器，结合动态跟踪和静态符号执行来预测潜在的 bug（例如 AttributeError，TypeError)。 它根据给定的测试运行给定的程序，以收集在执行过程中插入的模块。然后，它将这些模块的代码规范化为它们的 SSA 形式，并再次执行规范化的程序以收集运行时跟踪。以这些跟踪作为输入，它进行基于符号执

行的预测分析，以探索所有可能的执行路径上的 bug。

To enable comparison with it, we developed and set up a modified version of PyPredictor that is compatible with Python 3.7. Moreover, we improved its normalizing module to support some standard Python features (e.g., *GeneratorExp, BoolOp, Call*) and a few others (e.g., *AnnAssign, AsyncFunctionDef, Try* [59]). We then developed a plug-in based on POLYCRUISE to detect *TypeError* bugs for Python programs.

为了与之进行比较，我们开发并设置了 PyPredictor 使它与 Python 3.7 兼容。此外，我们改进了它的规范化模块以支持一些标准的 Python 特性（例如，GeneratorExp、BoolOp、Call）和其他一些特性（例如，AnnAssign、AsyncFunctionDef、Try[59])。然后，我们开发了一个基于 PolyCruise 的插件来检测 Python 程序的 TypeError bug。

libdft. This is designed for dynamic data flow tracking based on the Intel PIN framework [25]. It dynamically instruments the target binary and tracks the taint flow for every executed instruction with a set of predefined taint propagation rules (i.e., external API calls). This design brings huge run-time overhead that hurts its scalability in real-world applications.

libdft.这是为基于 Intel PIN 框架的动态数据流跟踪而设计的[25]。它动态地检测目标二进制文件，并用一组预定义的污点传播规则（即外部 API 调用）跟踪每个执行的指令的污点流。这种设计带来了巨大的运行时开销，从而损害了它在实际应用程序中的可伸缩性。

To enable comparison, we built a dynamic taint analyzer on libdft based on PIN 3.7. We added APIs for arithmetic instructions (e.g., *ADD, SUB, DIV*) to detect integer-overflow and divide-by-zero vulnerabilities. For CALL instructions, we added rules to check the taint tags and sinks, targeting vulnerabilities such as buffer overflow and data leakage.

为了进行比较，我们在基于 PIN 3.7 的 libdft 上构建了一个动态污点分析器。我们为算术指令（例如，ADD、SUB、DIV）添加了 API，以检测整数溢出和被零除的漏洞。 对于调用指令，我们添加了检查 taint 标记和接收器的规则，目标是缓冲区溢出

和数据泄漏等漏洞。

We ran all of our experiments on an Ubuntu 18.04 workstation with an Intel i7-10875H CPU and 16GB RAM.

6.3 RQ1: Effectiveness of POLYCRUISE

Table 3: Effectiveness results of POLYCRUISE on PyCBench including #inter-language paths (INT-LP), #Intra-language paths (ITR-LP), #false negatives (FN), #false positives (FP)

Group	#INT-LP	#ITR-LP	#FN	#FP
General flow	10	4	0	0
Global flow	9	0	0	0
Field sensitivity	8	0	0	2
Object sensitivity	9	2	0	1
Dynamic invocation	4	0	0	0
Total	40	6	0	3

Results on PyCBench. As Table 3 shows, POLYCRUISE reported all the true-positive paths. It additionally reported three false positives, including two in the *field sensitivity* group and one in the *object sensitivity* group. This was anticipated as our current implementation drops field-insensitivity for better language independence (§4), and the dynamic instrumenter (for Python) is presently object-insensitive. These false positives could be eliminated by tuning the implementation.

Pycbench 上的结果。作为表格 3 显示，PolyCruise 报告了所有的真正的阳性路径。它还报告了三个假阳性，包括两个在 field 敏感组和一个在对象敏感组。这是预期的，因为我们当前的实现为了更好的语言独立性（段 4）而放弃了 field 不敏感，以及动态插桩器（用于 Python）目前是对象不敏感的。这些误报可以通过调优实现来消除。

As a result, POLYCRUISE achieved 93.5% precision and 100% recall on PyCBench. Although the micro-benchmarks can not represent all real-world application scenarios, common program analysis features have been considered in the design of PyCBench to help assess the analysis soundness and accuracy. Thus, these numbers are still encouraging for the merits of POLYCRUISE for cross-language analysis.

结果，PolyCruise 在 PyCBench 上实现了 93.5%的准确率和 100%的召回率。虽然微基准不能代表所有的实际应用场景，但 PyCbench 的设计中考虑了一般的程序分

析的特点，以帮助评估分析的正确性和准确性。因此，对于 PolyCruise 在跨语言分析方面的优点，这些数字仍然令人鼓舞。

Table 4: Effectiveness results of POLYCRUISE on real-world projects. P_g: #ground-truth paths, P_p: #paths found by POLYCRUISE, TP: true positive, TP_{sc}: true positive in security context, FN: false negative, RC: recall, PI: precision, PI_{sc}: precision under security context.

Benchmark	P_g	P_p	#TP	$\#TP_{sc}$	#FN	RC	PI	PI_{sc}
Bounter	3	3	3	2	0	100%	100%	66.7%
Immutables	2	2	2	1	0	100%	100%	50%
Japronto	1	1	1	1	0	100%	100%	100%
Cvxopt	5	7	5	4	0	100%	71.4%	57.5%
Pyo	4	4	4	2	0	100%	100%	50%
Summary	15	17	15	10	0	100%	88.2%	58.8%

Results on real-world benchmarks. Table 4 shows the effectiveness results of POLYCRUISE on the five real-world projects. For the total of 486 tests, we obtained 15 paths as ground truth by manual validation. Among the 17 paths generated by POLYCRUISE, 15 were validated to be true positives, leading to a precision of 88.2% and a recall of 100% overall. The two false positives in Cvxopt were caused by the field-insensitivity of our current POLYCRUISE implementation as illustrated in Figure 12. Moreover, we validated that 10 of the potential vulnerabilities induced by the 15 paths were exploitable, leading to a security-context precision of 58.8%.

真实世界基准的结果。表 4 给出了 Polycruise 在五个实际项目上的效果。在总共 486 次测试中，我们通过人工验证获得了 15 条路径作为基准真值。在 Polycruise 确定的 17 条路径中，15 条为真阳性，正确率为 88.2%，召回率为 100%。在 Cvxopt 中的两个假阳性是由我们当前的 PolyCruise 实现的 field 不敏感引起的，如图 12 所示。此外，我们还验证了 15 条路径所引发的潜在漏洞中有 10 条是可利用的，安全上下文精度为 58.8%。

POLYCRUISE achieved 93.5% and 88.2% precision on our microbench and real-world systems, respectively, with a perfect recall for both, hence promising effectiveness.

PolyCruise 在微平台系统和真实世界系统的精度分别为 93.5%和 88.2%，分别具有完

美的召回，因此是希望的有效性。

6.4 RQ2: Effi ciency of POLYCRUISE

> Since the PyCBench programs are small and their executions (against the tests we curated) are simple, we gauged the efficiency of POLYCRUISE just on the 12 real-world benchmarks. As seen from Table 2, these benchmarks include very-large scale systems like PyTorch and NumPy. The basis of our efficiency study is *8.1 million* lines of code.

由于 PyCbench 程序很小，它们的执行（与我们策划的测试相比）也很简单，所以我们仅在 12 个真实世界的基准上衡量了 PolyCruise 的有效性。如表 2 所示，这些基准包括非常大的像 Pytorch 和 Numpy 这样的大型系统。我们效率研究的基础是 810 万行代码。

> A key metric of efficiency in our study is the run-time slowdown incurred by the dynamic analysis in POLYCRUISE. Due to the complexity of multilingual system executions, we used a calibrated method to compute this metric, as detailed in Appendix B. Next, we present our results on this and other efficiency metrics (as laid out in §6.2).

在我们的研究中，效率的一个关键度量标准是由 PolyCruise 中的动态分析引起的运行时放缓。由于多语言系统执行的复杂性，我们使用了一个校准的方法来计算这个度量，详见附录 B。接下来，我们将介绍我们在这个和其他效率度量上的结果（段 6.2 制定)。

Table 5: Efficiency of SDA in terms of time cost (SDA-T), peak memory (SDA-M), and instrumentation rate (Instm%).

Benchmark	SDA-T (seconds)	SDA-M (MB)	Instm%
Bounter	0.02	2.97	52%
Immutables	0.04	4.68	50%
Simplejson	0.03	4.47	56%
Japronto	0.02	3.89	47%
Pygit2	0.13	14.54	43%
Psycopg2	0.14	15.32	57%
Cvxopt	1.21	35.52	52%
Pygame	2.27	85.32	44%
PyTables	2.45	101.11	51%
Pyo	20.21	258.73	62%
NumPy	10.99	557.95	48%
PyTorch	175.19	7,414.95	51%

Efficiency of SDA. Table 5 shows the efficiency results of the SDA step which dominates the total costs of Phase 1. Overall, although its time and memory cost increased with greater subject size, the SDA finished in three minutes at most, for our largest subject system PyTorch (of 6 million lines of code). For most of the other (smaller) subjects, the SDA finished in just a few seconds or milliseconds.

SDA 效率。表 5 显示控制阶段 1 总成本的 SDA 步骤的效率结果。总的来说，虽然它的时间和内存成本随着项目大小的增加而增加，但对于我们最大的主题系统来说，SDA 最多在三分钟内完成 PyTorch（指 600 万行代码）。对于大多数其他（较小的）主题，SDA 仅在几秒钟或毫秒内完成。

For systems of 220KLOC or smaller, the SDA only took <100MB memory at peak, which is almost negligible on modern computing platforms. For larger systems, the peak memory was 260MB or more. The highest memory cost was seen by PyTorch, which was over 7.4GB hence may or may not be acceptable on a modestly-configured computer. However, for a system at this scale, the peak memory is still reasonable.

对于 220KLOC 或更小的系统，SDA 峰值仅占用略 100MB 内存，这在现代计算平台上几乎可以忽略不计。对于较大的系统，峰值内存为 260MB 或更多。内存开销最高的是 PyTorch 它超过 7.4GB，因此在配置较低的计算机上可能被接受，也可能无法接受。 但是，对于这种规模的系统，峰值内存仍然是合理的。

Recall that the inclusion of the SDA step in Phase 1 was mainly justified by its efficiency merits in reducing the static/-dynamic instrumentation scope. Now, to quantify these merits, we computed another efficiency metric, *instrumentation rate*, which is the percentage of code lines instrumented as guided by symbolic dependencies (i.e., the SDA results). As shown in Table 5, the SDA reduced the instrumentation by 57% in the best case (Pygit2). Even in the worst case (Pyo), the reduction was still substantial (38%). The main reason that the reduction was not even greater was that we chose to be conservative with the SDA to ensure the resulting symbolic dependencies to be a safe approximation of the dynamic information flow against any possible execution of the system under analysis.

回想一下，在阶段 1 中包含的 SDA 步骤主要是由于它在减少静态/动态检测范围方面的效率优势。现在，为了量化这些优点，我们计算了另一个效率度量，检测率，它是由符号依赖（即 SDA 结果）指导检测的代码行的百分比。 如表 5 所示，在最好的情况下，SDA 将插桩减少了 57%（Pygit2）。即使在最坏的情况下（Pyo），减少幅度仍然很大（38%）。减少不是更大的主要原因是我们选择了对 SDA 的保守性，以确保所产生的符号依赖关系是动态信息流的安全近似值，而不是分析中系统的任何可能执行。

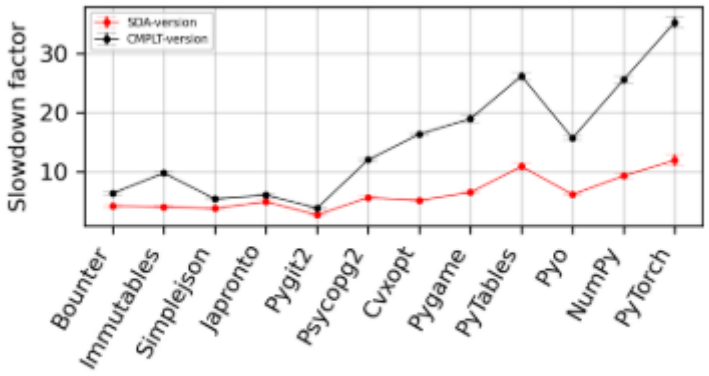

Figure 8: Run-time slowdown (y axis) on the SDA-version versus the CMPL-version of real-world benchmarks (x axis).

Run-time slowdown and memory usage. Figure 8 compares the run-time slowdown (for the ten executions) incurred by POLYCRUISE as seen by the SDA- versus CMPL-version of each of the 12 real-world benchmarks. The baseline costs were from the pure-versions. The error bars indicate the standard errors associated with the means.

运行时减慢和内存使用。 图 8 对比了由 polycruise 引起的运行时减速（对于 10 个执

行），如 12 个真实世界基准中的每一个的 SDA 与 CMPL 版本所见。基线成本来自纯净版本。错误条指出与原因相关联的标准错误。

Compared to the pure-versions, the SDA-versions were 2.71x (for Pygit2) to 11.96x (for PyTorch) slower. From Table 5 we saw that POLYCRUISE achieved a lower instrumentation rate on Pygit2 (43%) than it did on PyTorch (51%), which partly explains the smaller slowdown factor POLYCRUISE had with Pygit2 at runtime. Another reason lies in the complexity of the executions. Indeed, the PyTorch executions were much more computationally complex hence expensive than the executions of other benchmarks like Pygit2. One straightforward measure of this complexity for an execution is the number of events in the execution. We found that 55 million events were generated on average across the ten PyTorch executions, significantly more than the numbers of events in other benchmarks' executions.

与纯净版本相比，SDA 版本为 2.71 倍（对于 Pygit2）至 11.96 倍（对于 PyTorch）慢。从表 5 我们看到 Polycruise 在 Pygit2 上（43%）比在 PyTorch 上(51%)实现了更低的插桩率，这部分解释了 Polycruise 较小的减速因子 Pygit2 在运行时。另一个原因在于执行的复杂性。事实上，PyTorch 的执行在计算上要复杂得多，因此比执行其他基准测试（如 Pygit2）要昂贵得多。对执行的复杂性的一个直接度量是执行中的事件数。我们发现在 10 个 PyTorch 执行中平均产生了 55 百万个事件，大大超过其他基准执行中的事件数量。

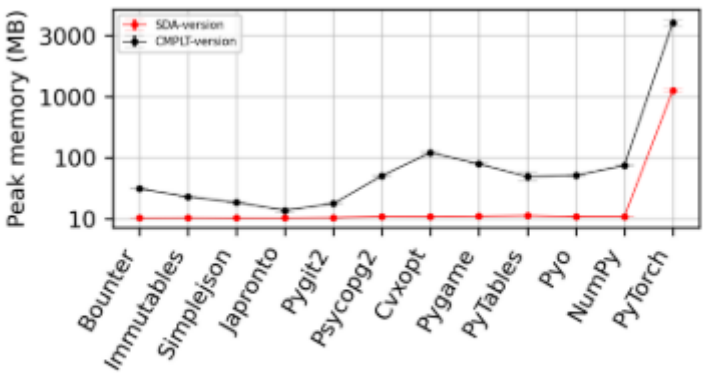

Figure 9: Peak memory usage (y axis) on the SDA-version versus the CMPL-version of real-world benchmarks (x axis).

In terms of (peak) memory usage, as shown in Figure 9, the overall online analysis in POLYCRUISE only used a small amount of memory in absolute terms. For all system executions except for those of the largest system PyTorch, the memory use was no more than 30MB. Even in the worst case (with PyTorch), the peak memory was 1GB, which is still acceptable on modern computers.

在（峰值）内存使用方面，如图 9 所示，在 PolyCruise 中的整体在线分析只使用了少量的绝对内存。除最大系统 PyTorch 外的所有系统执行的内存使用不超过 30MB。即使在最坏的情况下（PyTorch），峰值内存为 1GB，这在现代计算机上仍然可以接受。

On the other hand, both the run-time slowdown and peak memory usage as seen by the SDA-versions were much smaller than those seen by the CMPL-versions, as contrasted in Figures 8 and 9. Specifically, the SDA improved the reduction of slowdown factor from 18.3% (in Japronto) to 66.2% (in PyTorch), and reduced the memory usage by 16.2% (in Japronto) to 67.1% (in Cvxopt), compared to whole-system instrumentation with any scope reduction.

另一方面，SDA 版本的运行时减慢和内存使用峰值都比 CMPL 版本小得多，如图 8 和 9 所示。特别是与任何范围缩小的全系统插桩相比，SDA 将放缓因素的减少幅度由 18.3%（在 Japronto 中）提高至 66.2%（在 PyTorch 中），并将内存使用量从 16.2%（在 Japronto 中）减少到 67.1%（在 Cvxopt 中）。

POLYCRUISE took mostly a few seconds (and 3 mins at worst) for its static analyses, while its online dynamic analysis incurred 2.71∼11.96x slowdown and used a moderate amount of memory. The static analyses helped improve the efficiency of the dynamic analysis significantly.

Polycruise 的静态分析大多需要几秒钟（最坏情况下需要 3 分钟），而它的在线动态分析导致 2.71-11.96 倍的放缓并且使用了一个中等的内存量。静态分析有意义的帮助提高了动态分析的效率。

6.5 RQ3: Vulnerability Discovery

> From the subject executions considered in RQ1 through RQ3, POLYCRUISE identified 14 new vulnerabilities related to five of the real-world benchmarks, as listed in Table 6. We filed issues for these findings on corresponding GitHub repositories. At the time of writing, the respective developers have confirmed 11 and fixed 8 of them. We illustrate with one of the fixed cases and one of the other confirmed cases.

从 RQ1 到 RQ3 中考虑的主题执行中，PolyCruise 确定了 14 个与五个现实世界基准相关的新漏洞，如表 6 中的列表。我们在相应的 GitHub Repositories 上为这些发现提出了问题。在撰写本报告时，相关开发人员已经确认了 11 个，并修复了其中的 8 个。 我们用一例修复案例和一例其他确认案例来说明。

Figure 10: New vulnerability case 1: risk of buffer-overflow.

Case 1: Buffer overflow. In the integration-test execution of NumPy, this fixed case is a risk point of buffer overflow, as depicted in Figure 10. The user input is the shape of an array, which can be arbitrary. This input is taken from a file as the source then passed through the API `numpy.zero`. Python interpreter wraps the data as a `PyObject` and takes it as an argument to the C function `array_zeros`. The function decodes the arguments, gets the value of *shape*, and passes dimension (*shape.len*) to another function `PyArray_Zeros` as argument *nd*. Eventually in the function `PyArray_NewFromDescr_int`, the variable *nd* is used in `memcpy` to specify the number of bytes to copy. Since the target *newstrides* is a fixed-size stack buffer, buffer overflow will happen when the shape length propagated from Python is larger than expected here.

案例 1：缓冲区溢出。在 Numpy 的集成测试执行中，此修复案例是缓冲区溢出的风险点，如图 10 所示。用户输入是数组的 shape，可以是任意的。该输入取自作为源的文件，然后通过 API numpy.zero 传递。 Python 解释器将数据包装为 PyObject，并将其作为 C 函数 array_zeros 的参数。函数对参数进行解码，获取 shape 的值，并将 dimension (shape.len) 作为参数 nd 传递给另一个函数 PyArray_Zeros。最后，在函数 PyArray_NewFromDescr_int 中，在 memcpy 中使用变量 nd 来指定要复制的字节数。由于目标 newstrides 是一个固定大小的堆栈缓冲区，当从 Python 传播的 shape 长度大于此处的预期时，就会发生缓冲区溢出。

Figure 11: New vulnerability case 2: incomplete comparison.

Case 2: Incomplete string comparison. During a NumPy integration-test execution, a vulnerability of incomplete string comparison was found as depicted in Figure 11. At the source, the user input was read into the variable `Shape` and `Type` in the Python function `test_functionality`, passed through the NumPy API `numpy.empty`, and flowed forward into the C function `array_empty` where the `dtype` object is passed into `PyArray_DescrConverter` for parsing. As the object propagated into `_convert_from_str`, the type description is extracted and used as the first argument in `strncmp`. Since the terminator of the string is not considered, the comparison result may not be the same as expected. More seriously, it can be exploited to cause the API to change its control flow (i.e., causing it to deny services by "goto fail").

案例 2：字符串比较不完整。 在 Numpy 集成测试执行时，发现了不完整字符串比较的漏洞，如图 11 所示。在 source 中，用户输入被读入变量 shape 和 Type 到 Python 函数 test_functionality 中，通过 Numpy API numpy.empty，并向前流入 C 函数 array_empty，在该函数中 dType 对象被传递到 PyArray_DescrConverter 中进行解

析。当对象传播到_convert_from_str 时，将提取类型描述并将其用作 strncmp 中的第一个参数。由于没有考虑字符串的结束符，比较结果可能与预期的不一样。更严重的是，可以利用它来导致 API 更改其控制流（即，通过“goto fail”来导致 API 拒绝服务）

Exploitability. To demonstrate that the vulnerabilities discovered by PolyCruise are exploitable, we have developed a proof-of-the-concept (PoC) exploit for each of the (11) vulnerabilities that have been confirmed by developers so far, found here. For each PoC, a script that triggers the vulnerability and the corresponding run-time output are provided.

可利用性。为了证明 PolyCruise 发现的漏洞是可利用的，我们为开发人员迄今确认的（11）个漏洞中的每一个开发了概念验证（poc）利用。对于每个 PoC 提供触发漏洞的脚本和相应的运行时输出。

PolyCruise discovered 14 new vulnerabilities in six real-world multilingual systems, with 11 confirmed and 8 fixed; all of these were cross-language vulnerabilites.

POLYCRUISE 在 6 个真实世界多语言系统中发现 14 个新的漏洞，其中 11 确认 11 个修复 8 个；所有的这些均为跨语言漏洞。

6.6 RQ4: Comparison with peer tools

Table 7: PolyCruise vs PyPredictor: PolyCruise can find the same defects (type errors) for Python programs. Sizes are in KLoC; T/M denotes Time (seconds)/Memory (MB).

Benchmark	Size	Issue#	PolyCruise		PyPredictor	
			Identify	T/M	Identify	T/M
request [64]	42.7	2638	✓	1.2/15.1	✓	41.6/20.3
		2639	✓	1.3/14.9	✓	35.1/19.5
		2267	✓	0.6/14.8	✓	13.2/15.1
		2613	✓	0.8/14.8	✓	28.4/21.7
fabric [17]	4.3	1303	✓	0.6/11.6	✓	1.4/6.1
		1906	✓	0.7/11.6	✓	1.1/4.9
		1191	✓	0.7/11.6	✓	2.5/6.5
salt [65]	70.9	24820	✓	2.3/18.3	✓	83.6/33.3
		25006	✓	2.4/18.5	✓	89.8/33.5
web2py [68]	93.1	968	✓	0.8/12.1	✓	4.2/10.3

Comparison with PyPredictor. We compared our fixed version of PyPredictor with POLYCRUISE against four popular Python benchmarks (with 10 type error defects as ground truth) used in the original evaluation of PyPredictor, in terms of efficiency and effectiveness. As shown in Table 7,

POLYCRUISE detected all the 10 bugs as the fixed PyPredictor did. In terms of efficiency, PyPredictor took significantly more time and memory in most cases due to the symbolic execution. For the benchmark fabric [17], POLYCRUISE allocated more memory as it created a fixed-size event queue during initialization, although only a tiny portion of the allocated space was actually used.

与 PyPredictor 比较。我们将固定版本的 PyPredictor 与 PolyCruise 在 4 个用于 PyPredictor 的原始评估，包括效率和有效性的流行的 Python 基准测试上（以 10 种类型的错误缺陷作为基本事实）进行了比较。如表 7 所示，PolyCruise 检测到所有 10 个 bug 像固定的 PyPredictor 做的一样。在效率方面，PyPredictor 在大多数情况下，由于符号执行，占用了大量的时间和内存。对于基准结构[17]，PolyCruise 分配了更多的内存，在其初始化期间创建了一个固定大小的事件队列，尽管实际上只使用了一小部分分配的空间。

Table 8: POLYCRUISE vs libdft: POLYCRUISE can find the same vulnerabilities and be more efficient for C programs.

Benchmark	Size (KLoC)	Type	CVE ID	POLYCRUISE			libdft		
				Identify	SD-factor	Memory (MB)	Identify	SD-factor	Memory (MB)
openjpeg-2.1.2 [50]	168.8	Division-by-zero	CVE-2016-9112	✓	3.5	21.4	✓	40.3	580.6
libarchive-3.2.2 [39]	233.2	Buffer overflow	CVE-2016-1541	✓	4.3	33.6	✓	17.9	375.2
curl-7.50.1 [13]	127.9	Integer overflow	CVE-2016-8620	✓	3.1	25.9	✓	65.7	51.8
libtiff-4.0.7 [41]	127.1	Integer overflow	CVE-2016-10093	✓	2.7	13.6	✓	11.3	55.3

Comparison with libdft. We carefully selected four widely used C programs as the benchmarks in the comparison as shown in Table 8. In all four of these programs, there are defects that have been reported as CVEs, which are regarded as ground truth to compare these two tools. In terms of effectiveness, POLYCRUISE succeeded in identifying all known vulnerabilities when corresponding source/sink pairs are configured. For instance, the reason for *CVE-2016-1541* in `libarchive` is that an external value (defined in the archive file) reaches `memcpy` as a buffer size indicator, and no boundary check exists along the data flow path; hence we configured source/sink pair as (`fead`, `memcpy`) and succeeded in detecting this vulnerability. In terms of efficiency, the slowdown of libdft was 4 to 13 times that of POLYCRUISE while using 10 times more memory by average. The main reason is that libdft needs to track taint information for most instructions; this design incurs a significant runtime overhead due to the corresponding API calls (i.e., predefined propagation rules).

与 libdft 比较。我们精心挑选了四个广泛使用的 C 程序作为比较的基准，如表 8 所示。在所有四个程序中，都有缺陷被报告为 CVE，这些缺陷被视为对这两个工具进行比较的基本事实。就有效性而言，PolyCruise 在配置相应的 source/sink 对时成功地识别了所有已知的漏洞。例如，Libarchive 中 CVE-2016-1541 的原因是，外部值（在归档文件中定义）作为缓冲区大小指示符到达 memcpy，并且沿着数据流路径不存在边界检查；因此，我们将、source/sink 收器对配置为(fead，memcpy)，并成功地检测到了该漏洞。在效率方面，libdft 是 Polycruise 的 4 到 13 倍，而平均使用多 10 倍的内存。主要原因是 libdft 需要跟踪大多数指令的污点信息；由于相应的 API 调用（即，预定义的传播规则），这种设计导致了显著的运行时开销。

While mainly targeting cross-language DIFA, POLYCRUISE also outperformed (in cost-effectiveness) state-of-the-art peer tools for single-language analysis on C and Python.

虽然主要针对跨语言 DIFA, POLYCRUISE 也比在 C 和 Python 上最先进的单语言分析对等工具更好（在成本-效益方面）。

6.7 Regarding the Vulnerabilities Discovered

The previously known vulnerabilities discovered by PolyCruise have been documented in detail on respective CVE pages as listed in Table 8. The documentation describes how the vulnerabilities were disclosed and addressed. Regarding each of the 14 new vulnerabilities discovered by PolyCruise, we have contacted the respective developers. By the time of this paper submission, all of these have been reported to the system vendors, although some of them have not been confirmed yet, possibly because the developers have not been active recently. Others have all been confirmed, among which three have been fixed. The details on each of these 14 vulnerabilities are documented in our artifact package here.

Polycruise 发现的先前已知的漏洞已详细记录在各自的 CVE 页面上 ,如表 8 中的列表。文档描述了如何披露和解决漏洞。针对 PolyCruise 发现的 14 个新漏洞，我们已经联系了相应的开发人员。在提交本文时，所有这些都已报告到系统供应商，尽管其中一些尚未得到确认，可能是因为开发人员最近没有活动。其他的都已经确认，其中三个已经修复。这 14 个漏洞的详细信息都记录在我们的工件包中。

6.8 Effort for Vulnerability Confirmation

Based on our current implementation, additional analysis/effort is expected in order to confirm whether a dynamic information flow path reported by PolyCruise actually represents a true, non-trivial vulnerability.

基于我们当前的实现，需要额外的分析/努力来确认 PolyCruise 报告的动态信息流路径是否真的代表了一个真实的、非平凡的漏洞。

One reason for requiring such effort is that no sanitization is currently implemented in the vulnerability detection plugins (i.e., security application modules). As a result, false alarms may arise. For example, a reported data flow path between a source/sink pair may be a false alarm when a boundary check exists on the control flow from the source to the sink.

需要这样努力的一个原因是，目前在漏洞检测插件（即安全应用程序模块）中没有实现消毒。 因此，可能会出现假警报。 例如，当从源到接收器的控制流上存在边界检查时，

source/sink 对之间报告的数据流路径可能是假警报。

> Another reason is that a reported path may not be concerning to the user because the source is not sensitive or the sink is not critical in particular use scenarios. For instance, a data flow path from a source to a sink indicates a possible sensitive data leakage; however, the data retrieved at the source may not be sensitive to the user in the specific application scenario. Such security context factors usually vary across different application scenarios, making it hard for PolyCruise to discern whether the source is actually sensitive or the sink actually critical. Thus, post-DIFA analysis and confirmation is generally a necessary additional step.

另一个原因是，报告的路径可能与用户无关，因为在特定的使用场景中，source 不敏感，或者 sink 不是决定性的。例如，从 source 到 sink 的数据流路径指示可能的敏感数据泄漏；但是，在 source 处检索的数据可能对特定应用程序场景中的用户不敏感。这种安全上下文因素通常在不同的应用场景中有所不同，这使得 PolyCruise 很难区分 source 实际上是敏感的还是接收器实际上是关键的。因此，后 DIFA 分析和确认通常是一个必要的额外步骤。

7 Related Work

> **Language-independent technique.** ORBS [5] claimed support for analyzing multilingual programs without extra efforts. It computes a program slice through repeated action called "delete-execute-observe" on the target system until no more lines can be deleted. Although the language-independent feature is appealing, serious scalability issues prevent its application in real-world programs; even its improved version [32] can not scale well yet.

语言无关技术。ORBS[5]声称无需额外努力就能分析多语言程序。它通过在目标系统上重复的“删除-执行-观察”操作来计算程序片，直到不能删除更多的行。尽管独立于语言的特性很有吸引力，但严重的可伸缩性问题阻碍了它在现实世界程序中的应用；甚至它的改进版本[32]还不能很好地缩放。

Semantic summarization. Debuting for heap analysis in C/C++ programs [15], semantic summarization is being applied to static cross-language analysis with a formally defined syntax [34, 69]. Unfortunately, the summarization causes profound information loss, leading to low recall. Moreover, complex language semantics is a barrier to the expansion and application of such techniques.

语义摘要。C/C++程序中堆分析的调试[15]语义摘要正被应用于具有正式定义的语法的静态跨语言分析[34，69]。遗憾的是，摘要会造成原发现信息的丢失，导致低召回率。此外，复杂的语言语义也阻碍了这些技术的推广和应用。

Unified intermediate representation. For multilingual program analysis, pyLang [43] compiles Python programs into LLVM IR; JLang [74] also succeeds in translating Java code into LLVM IR. Arzt et al. [3] translated the common intermediate language of the Microsoft .net framework into Jimple. Lopes et al. [42] constructed code property graphs from WebAssembly code to detect vulnerabilities in it. With the support of LLVM, several languages (e.g., C/C++/Rust [42], JavaScript/TypeScript [63]) were compiled into WebAssembly, which then enabled analyses across components of those languages. However, much engineering work is required to develop and maintain the compiler for each language, and many of the prior frameworks have been shown impractical in our empirical studies due to complex language features.

统一中间表示。对于多语言程序分析，pyLang[43]将 Python 程序编译成 LLVM IR；JLang[74]还成功地将 Java 代码转换为 LLVM IR。Arzt 等人[3]将 Microsoft .NET Framework 的通用中介语言翻译成 Jimple。Lopes 等人[42]从 WebAssembly 代码构造代码属性图以检测其中的漏洞。在 LLVM 的支持下，几种语言（如 C/C++/Rust[42],JavaScript/TypeScript [63])被编译成 Webassembly ,然后可以跨这些语言的组件进行分析。然而，开发和维护每种语言的编译器需要大量的工程工作，而且由于语言的复杂特性，许多现有的框架在我们的实证研究中已经证明是不切实际的。

Dynamic techniques based on virtual machine. Truffle [29] proposed a multi-language dynamic taint analysis framework supporting the languages JavaScript, Python, and C/C++ on top of a polyglot virtual machine (GraalVM). However, three significant issues limit its practicability: (1) implementing a runtime component for each specific language is laborious and error-prone work. (2) the AST-based runtime environment can lead to different program behaviors as in a real environment. (3) efficiency is a severe challenge to this virtual machine-based technique. DroidScope [73] analyzes Android malware across Java and native code, which however relies on runtime customization (via virtualization).

基于虚拟机的动态技术。特鲁夫[29]提出了一个在多语言虚拟机（GraalVM）之上支持JavaScript、Python 和 C/C++语言的多语言动态污点分析框架。无论如何，有三个重要的问题限制了它的实用性：（1）为每种特定语言添加运行时组件是一项费力且容易出错的工作。（2）基于 AST 的运行时环境可以导致不同于真实环境的程序行为。（3）效率是这种基于虚拟机的技术面临的严峻挑战。Droidcope[73]通过 Java 和本机代码分析 Android 恶意软件，但这依赖于运行时定制（通过虚拟化）。

Techniques targeting specific language combinations. Several prior works [1, 36, 67] addressed defects between Java and C components, while [4, 8, 33] focused on bug detection between Java and JavaScript. Brown et al. [7] proposed to detect vulnerabilities in JavaScript binding code (in C/C++) via static analysis, while Favocado [16] achieved that detection via fuzzing. In [35], symbolic execution across PHP code and its built-in C functions was realized by converting the execution results of the PHP code into C code.

针对特定语言组合的技术。几项先前的工程[1，36，67]解决了 Java 和 C 组件之间的缺陷，而[4，8，33]重点介绍了 Java 和 JavaScript 之间的 bug 检测。Brown 等人[7]建议通过静态分析来检测 JavaScript 绑定代码（在 C/C++中）中的漏洞，而 Favocado[16]通过模糊来实现检测。在[35]，通过将 PHP 代码的执行结果转换为 C 代码，实现了跨 PHP 代码及其内置的 C 函数的符号执行。

In contrast, our approach is significantly different. (1) We proposed SDA, a language-independent approach that is effective and efficient for large-scale programs and helps greatly reduce the runtime slowdown. (2) Our technique uses minimal language-specific analysis hence should be transferable to support new language combinations. (3) Our technique provides the first practical DIFA across Python and C languages.

相比之下，我们的做法明显不同。（1）我们提出了一种独立于语言的 SDA 方法，它对大型程序来说是有效的，并有助于大大减少运行时的速度减慢。（2）我们的技术使用最小的语言特性分析，因此应该对于支持新的语言组合是可转移的（3）我们的技术提供了第一个跨 Python 和 C 语言的实用 DIFA。

8 Conclusion

We presented PolyCruise, a novel dynamic information flow analysis (DIFA) for multilingual systems. Unlike prior approaches, PolyCruise utilizes an efficient and effective static analysis algorithm—language-independent symbolic dependence analysis to guide instrumentation hence enabling language-agnostic DIFA. Moreover, PolyCruise takes advantage of existing single-language analysis techniques and minimizes the language-specific engineering work. At runtime, PolyCruise adopts online and incremental analysis. It constructs the whole-program dynamic information flow graph based on which applications can be developed for vulnerability detection and beyond. We empirically demonstrated PolyCruise's efficiency with practical effectiveness against micro benchmarks and 12 real-world multilingual systems.

提出了一种新的多语言系统动态信息流分析(DIFA)方法 PolyCruise。与以前的方法不同，PolyCruise 使用了一种高效的静态分析算法--独立于语言的符号依赖分析来指导插桩，从而实现了与语言无关的 DIFA。此外，PolyCruise 充分利用了现有的单语言分析技术的优点，减少了特定语言的工程工作量。在运行时，PolyCruise 采用在线和增量分析。构建了整个程序的动态信息流图，并在此基础上开发了漏洞检测等应用程序。我们用微基准和 12 个真实世界的多语言系统的实际效果证明了 PolyCruise 的效率。