PolyCruise: A Cross-Language Dynamic Information Flow Analysis

Wen Li, *Washington State University, Pullman;* Jiang Ming, *University of Texas at Arlington;* Xiapu Luo, *The Hong Kong Polytechnic University;* Haipeng Cai, *Washington State University, Pullman*

https://www.usenix.org/conference/usenixsecurity22/presentation/li-wen

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

POLYCRUISE: A Cross-Language Dynamic Information Flow Analysis

Wen Li
Washington State University, Pullman
li.wen@wsu.edu

Jiang Ming
University of Texas at Arlington
jiang.ming@uta.edu

Xiapu Luo
The Hong Kong Polytechnic University
csxluo@comp.polyu.edu.hk

Haipeng Cai ✉
Washington State University, Pullman
haipeng.cai@wsu.edu

Abstract

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 *cross-language* 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.

1 Introduction

Real-world software systems today are mostly *multilingual*—they 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.

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.

Recognizing this need, researchers have started developing cross-language analyses that can support security applications. Yet most existing relevant approaches [8, 22, 34, 36, 67, 69] are exclusively focused on and limited to a particular case in the multi-language world: JNI programs—programs written in C and Java which interact through an interfacing mechanism called the Java Native Interface (JNI). A few other approaches exist but face practicality barriers. For instance, Truffle [29], a dynamic taint analysis framework, leverages a custom Java virtual machine (VM) called GraalVM [52, 70] to support multilingual systems. For each none-Java language, the user has to implement a runtime, which is a daunting task hence not always practical. Also, this VM-based approach is heavyweight in nature hence not scalable to large, complex systems. Moreover, the reliance on customized runtime brings portability barriers, given the diverse and evolving language features that are hard to keep the customization up to.

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.

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).

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.

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 language-specific 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.

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.

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.

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 cross-language DIFA to the best of our knowledge (§3).
- An open-source implementation of PolyCruise for Python-C programs, which works with large-scale, real-world 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).

We have released our dataset and code to facilitate reproduction, replication, and reuse, as all found here.

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.

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 language-independent. 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.

At a more detailed level, the diversity of interfacing mechanisms is greater. For instance, within the same (FFI) category, Java interacts with C via native function calls, passing parameters via `jobject`, while the interfacing between Python and C is different and more complex [59]. Even in the same language combination, the mechanism also varies. For instance, in Python-C programs, one may use `ctypes` to load a C library and then search and invoke a C function. Alternatively, a C module can be built as a Python extension to be used as a Python module. In the other direction, the C unit can invoke a Python function through the APIs provided by the Python interpreter, after converting C-type parameters to `PyObjects`. Other mainstream languages provide similar yet still diverse mechanisms (e.g., Go interacts with C via the `cgo` command while Ruby interacts with C via a particular FFI).

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).

Illustrating/motivating examples. Figure 1 depicts three cases of cross-language vulnerabilities each with a different interfacing mechanism between Python and C. p*i* and c*i* 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). Cross-language 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.

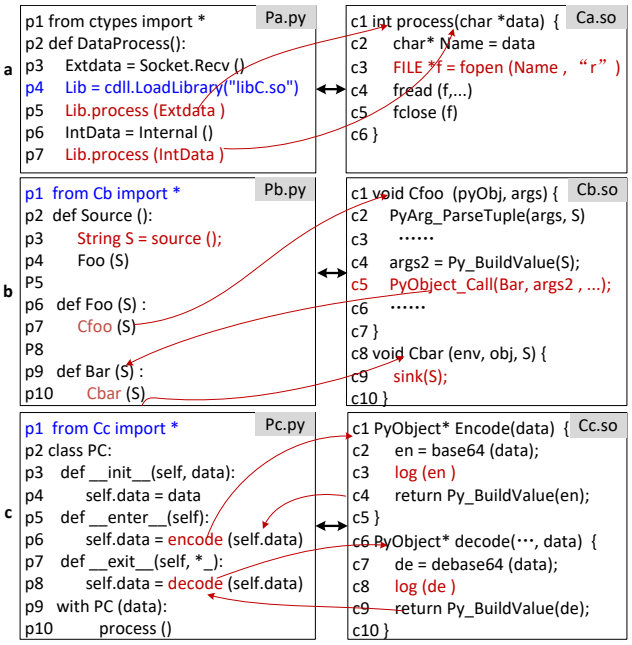

Figure 1: Example Python-C interfaces and vulnerabilities.

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.

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).

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.

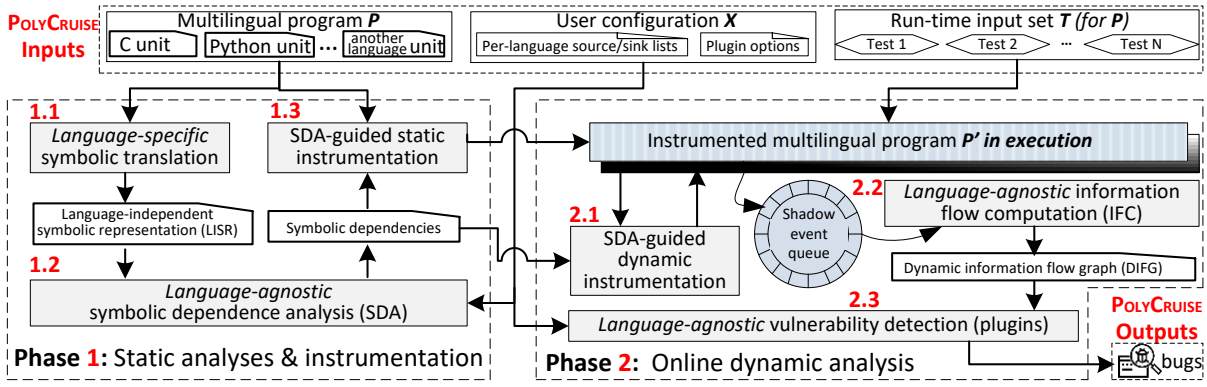

Figure 2: An overview of PolyCruise's architecture, including its inputs, major technical components, and outputs.

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.

For an interpreted language (e.g., Python), PolyCruise starts its **Phase 2** with a dynamic instrumentation while the statically instrumented program P' is running against the given input set T. This instrumentation is also guided by the symbolic dependencies resulting from the SDA in the previous phase. Notably, the dynamic analysis in this second phase is *online*; that is, the analysis is performed on the fly (in the memory). It uses a shadow memory based circular queue (i.e., *shadow event queue*) to buffer streaming execution events—no trace is serialized to any external storage. While these events are being collected, PolyCruise incrementally performs run-time information flow computation (IFC) between any source and sink (across all language units of P), with resulting flow facts being represented as (and continuously added to) a dynamic information flow graph (DIFG).

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**.

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 language-agnostic dynamic analysis, while only relying on minimal language-specific analyses. Following this rationale, the first phase works in three major steps as elaborated below.

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 language-independent symbolic representation (LISR) form.

Definition. The symbolic dependencies computed by SDA are intended to approximate *data dependencies* between statements of a language unit. Accordingly, our LISR representation considers three kinds of statements: *line* (i.e., assignment), *call*, and *return*, and two kinds of symbols: *global* and *function*. The formal syntax of LISR is hence given below:

$$
\begin{array}{lll}
P & ::= & G^*F^* \\
G & ::= & e \\
F & ::= & \tau f(x^*)S^* \\
S & ::= & [x=]e^* \mid [x=]f(e^*) \mid \text{return } e \\
e & ::= & x \mid C \mid \varepsilon \\
\tau & ::= & T \mid \varepsilon
\end{array}
$$

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

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
```

(defs) and uses, other common language syntax elements (e.g., operators) are dropped in LISR.

Translation. Following the LISR definition above, the symbolic translation of a language unit is simply done through light syntactic parsing of the unit. During the parsing, every statement in the unit's source code that belongs to the three kinds or includes one of the two types of data symbols considered in LISR is translated to a LISR statement or symbol. The main goal is to capture variable defs and uses. Since only such a simple syntactic analysis is needed and no any semantic (e.g., data/control flow) analysis is involved, translating the code of a language into its LISR is reasonably simple.

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).

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 $Output(arg)$, 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.

```
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.

Unlike an ordinary intermediate representation (IR), e.g., LLVM's bitcode, LISR is not meant to represent an entire program with respect to its full semantics. Instead, it only captures the most essential information (i.e.,def/use) needed for other analysis steps in PolyCruise. While different languages have different semantics, which constitutes a major barrier for cross-language analysis, the essential information can be represented in a language-agnostic manner, as is LISR. It is this nice property of LISR that essentially enables our DIFA approach to work across heterogeneous languages.

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.

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.

In addition to approximating true/flow dependencies [23] (i.e., when $D(S_i) \cap U(S_j) \neq \varnothing$), we also consider possible anti-dependencies [23] (i.e., when $U(S_i) \cap D(S_j) \neq \varnothing$) to ensure the soundness of the SDA. Without a pointer/reference analysis, the SDA has no access to aliasing information for the analyzed language unit. Yet ignoring aliases could lead to missing true dependencies induced by aliasing. For instance, in the example of Figure 3, suppose one source is variable V at Line 9 and S becomes an alias of V after Line 8. Considering the anti-dependencies here will lead Lines 13 and 14 to be included in the symbolic dependence set of Line 9 hence the dynamic information flow paths between Line 9 and Lines 13, 14 to be potentially captured—because only the statements symbolically dependent on Line 9 will be probed in the later instrumentation steps; or these flow paths would be missed.

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 $\{S_8, S_{11}\}$.

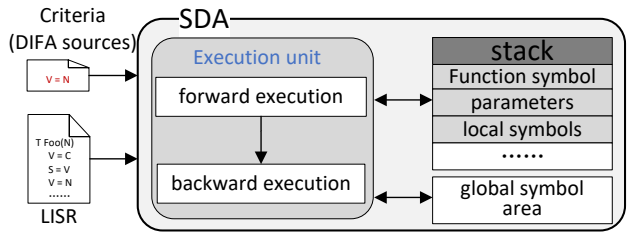

Figure 4: An overview of the symbolic dependence analysis.

```
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.

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.

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.

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.

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.

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
```

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.

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:

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.

Algorithm 4: The *Main* logic of SDA

```
   Input: L_sym:the set of local symbols for F; others are same as Algorithm 3
   Output: SDS_F:the (updated) SDS of 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_x.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
```

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.

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).

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.

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.

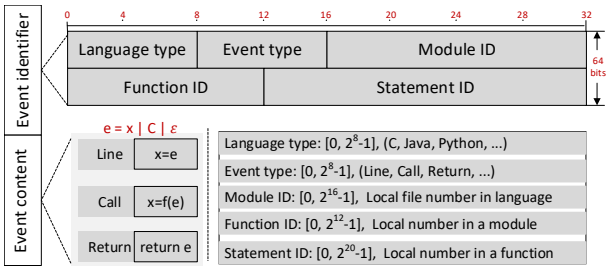

Figure 6: Definition of execution events monitored at runtime.

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.

3.3.1 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.

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.

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.

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.

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 $\vec{e}$.

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

- *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.
- *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 intra-procedural 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 inter-procedural (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.
- *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.
- *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.

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

Algorithm 5: Construct DIFG incrementally

```
   Input: e⃗_k: an execution event ∈ e⃗, φ: the no. of thread containing e⃗_k
   Output: 𝔾_Φ: 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
```

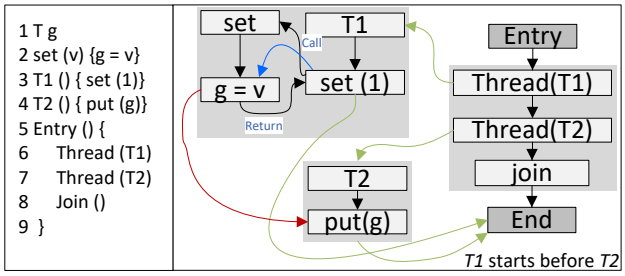

Figure 7: An example DIFG (right) for a program (left).

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 U($\vec{e}_k$) contains references to shared or global variables, global data dependence would be computed and added (Line 23).

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.

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.

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.

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 $class.M$.

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.

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.

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.

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.

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 $\boldsymbol{X}$, as shown in Figure 2).

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.

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.

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.

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.

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.

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.

6 Evaluation

Using our PolyCruise implementation (§4), the evaluation of our approach was guided by four questions below:

RQ1 *How effective is* PolyCruise *in terms of its precision?*
RQ2 *How efficient is* PolyCruise *in terms of its costs?*
RQ3 *Can* PolyCruise *find real-world vulnerabilities?*
RQ4 *How does* PolyCruise *compare to peer tools?*

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).

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

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

(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.

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).

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.

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.

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.

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.

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.

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.

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.

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.

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

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

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

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%

Python) is presently object-insensitive. These false positives could be eliminated by tuning the implementation.

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.

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%.

> 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.*

6.4 RQ2: Efficiency 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.

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%

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).

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.

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.

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.

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

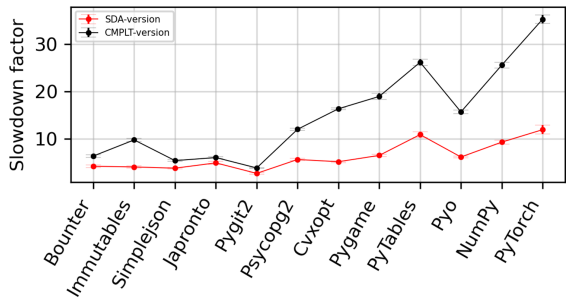

Figure 8: Run-time slowdown (*y* axis) on the SDA-version versus the CMPL-version of real-world benchmarks (*x* axis).

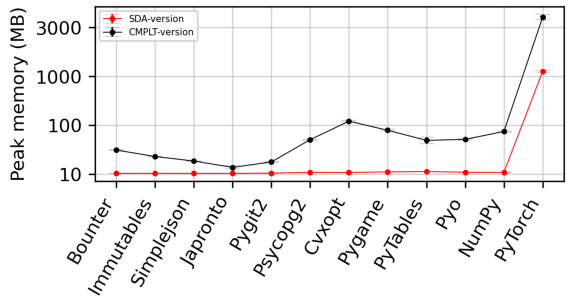

Figure 9: Peak memory usage (*y* axis) on the SDA-version versus the CMPL-version of real-world benchmarks (*x* axis).

were from the pure-versions. The error bars indicate the standard errors associated with the means.

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.

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.

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.

Table 6: Three types of new vulnerabilities discovered by PolyCruise: Integer-overflow (IntOf), Buffer-overflow (BufOf), and Incomplete-comparison (InCc).

Benchmark	#IntOf	#BufOf	#InCc	#Fixed	#Confirmed	#Pending	#CVEs
Bounter	0	1	0	1	0	0	1
Immutables	0	1	0	0	0	1	0
Japronto	0	1	0	0	0	1	0
Cvxopt	0	0	4	4	0	0	1
Pyo	0	2	0	2	0	0	2
NumPy	1	3	1	1	3	1	4
Summary	1	8	5	8	3	3	8

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.*

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.

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.

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").

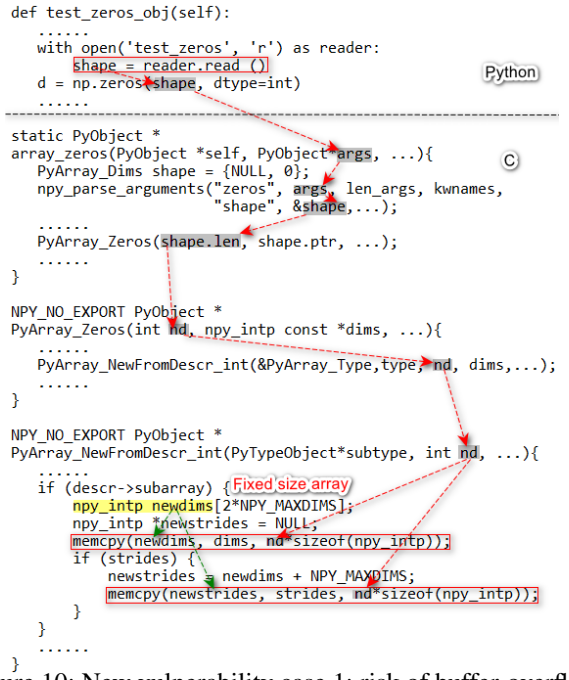

Figure 10: New vulnerability case 1: risk of buffer-overflow.

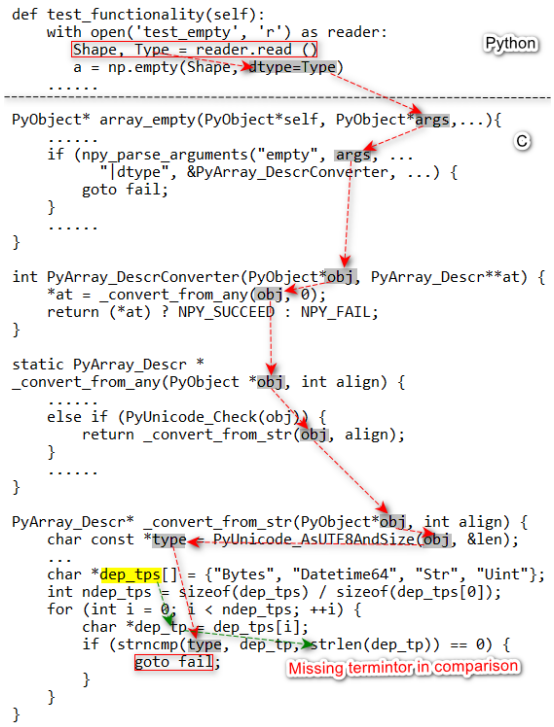

Figure 11: New vulnerability case 2: incomplete comparison.

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

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.

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

6.6 RQ4: Comparison with peer tools

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.

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).

> *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.*

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

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.

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.

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.

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.

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.

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.

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.

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).

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.

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 mini-

mal 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.

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.

Acknowledgments

We thank our shepherd Yan Shoshitaishvili and the anonymous reviewers for their effective guidance and constructive comments. This work was enabled by funding support from ARO (grant W911NF-21-1-0027) and NSF (grants CCF-1936522 and CCF-2146233). Jiang Ming was supported by NSF (grants CNS-1850434 and CNS-2128703).

References

[1] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *NDSS*, pages 1–15, 2016.

[2] Ajax. Pyo. https://github.com/belangeo/pyo, 2020.

[3] Steven Arzt, Tobias Kussmaul, and Eric Bodden. Towards cross-platform cross-language analysis with Soot. In *State Of the Art in Program Analysis (SOAP)*, pages 1–6, 2016.

[4] Sora Bae, Sungho Lee, and Sukyoung Ryu. Towards understanding and reasoning about Android interoperations. In *ICSE*, pages 223–233, 2019.

[5] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS: Language-independent program slicing. In *FSE*, pages 109–120, 2014.

[6] Bounter. Web. https://tinyurl.com/4667pnkv, 2017.

[7] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in javascript bindings. In *S&P*, pages 559–578, 2017.

[8] Achim D Brucker and Michael Herzberg. On the static analysis of hybrid mobile apps. In *Engineering Secure Software and Systems*, pages 72–88, 2016.

[9] Haipeng Cai. Hybrid Program Dependence Approximation for Effective Dynamic Impact Prediction. *TSE*, 2017.

[10] Haipeng Cai and Xiaoqin Fu. D^2ABS: A framework for dynamic dependence analysis of distributed programs. *TSE*, 2021.

[11] Haipeng Cai and Raul Santelices. TracerJD: Generic trace-based dynamic dependence analysis with fine-grained logging. In *SANER*, pages 489–493, 2015.

[12] Haipeng Cai, Raul Santelices, and Douglas Thain. DiaPro: Unifying dynamic impact analyses for improved and variable cost-effectiveness. *TOSEM*, 25(2):1–50, 2016.

[13] CurlSe. Web. https://curl.se, 2020.

[14] Cvxopt. Web. https://tinyurl.com/4k4v9646, 2016.

[15] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577, 2011.

[16] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases. In *NDSS*, 2021.

[17] Fabric. Web. https://github.com/fabric/fabric, 2020.

[18] Xiaoqin Fu and Haipeng Cai. FlowDist: Multi-staged refinement-based dynamic information flow analysis for distributed software systems. In *USENIX Security*, 2021.

[19] Xiaoqin Fu, Haipeng Cai, and Li Li. Dads: Dynamic Slicing Continuously-Running Distributed Programs With Budget Constraints. In *ESEC/FSE*, pages 1566–1570, 2020.

[20] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li. Seads: Scalable and cost-effective dynamic dependence analysis of distributed systems via reinforcement learning. *TOSEM*, 30(1):1–45, 2020.

[21] GitHub. The 2020 state of the OCTOVERSE. https://octoverse.github.com/, 2020.

[22] Manel Grichi, Mouna Abidi, Fehmi Jaafar, Ellis E Eghan, and Bram Adams. On the impact of inter-language dependencies in multi-language systems. In *QRS*, pages 509–509, 2020.

[23] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. 2011.

[24] Immutables. Web. http://immutables.github.io/.

[25] Intel. PIN. https://tinyurl.com/4yuhkd9r, 2020.

[26] Bob Ippolito. Simplejson a JSON encoder/decoder for Python. https://tinyurl.com/2s4y5hhr, 2020.

[27] Vasileios Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *VEE*, pages 121–132, 2012.

[28] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. A large scale study of multiple programming languages and code quality. In *SANER*, pages 563–573, 2016.

[29] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. Multi-language dynamic taint analysis in a polyglot virtual machine. In *MPLR*, pages 15–29, 2020.

[30] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004.

[31] Seongmin Lee, David Binkley, Robert Feldt, Nicolas Gold, and Shin Yoo. Observation-based approximate dependency modeling and its use for program slicing. *JSS*, 2021.

[32] Seongmin Lee, David Binkley, Nicolas Gold, Syed Islam, Jens Krinke, and Shin Yoo. Evaluating lexical approximation of program dependence. *JSS*, 160:110459, 2020.

[33] Sungho Lee, Julian Dolby, and Sukyoung Ryu. Hybridroid: static analysis framework for android hybrid applications. In *ASE*, pages 250–261, 2016.

[34] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis. In *ASE*, pages 127–137, 2020.

[35] Penghui Li, Wei Meng, Kangjie Lu, and Changhua Luo. On the feasibility of automated built-in function modeling for php symbolic execution. In *WWW*, pages 58–69, 2021.

[36] Siliang Li and Gang Tan. Finding bugs in exceptional situations of JNI programs. In *CCS*, pages 442–452, 2009.

[37] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. Pca: memory leak detection using partial call-path analysis. In *ESEC/FSE*, pages 1621–1625, 2020.

[38] Wen Li, Na Meng, Li Li, and Haipeng Cai. Understanding language selection in multi-language software projects on GitHub. In *ICSE-Companion*, pages 256–257, 2021.

[39] Libarchive. Library. http://www.libarchive.org, 2020.

[40] Libgit2.org. pygit. https://tinyurl.com/2hpw3pwt, 2014.

[41] Libtiff.org. lib&utilities. http://www.libtiff.org, 2020.

[42] Pedro Daniel Rogeiro Lopes. Discovering vulnerabilities in WebAssembly with code property graphs.

[43] lukarao. PyLLVM. https://tinyurl.com/yckr397m, 2020.

[44] Philip Mayer and Alexander Bauer. An empirical analysis of the utilization of multiple programming languages in open source projects. In *EASE*, pages 1–10, 2015.

[45] Philip Mayer, Michael Kirsch, and Minh Anh Le. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *JSERD*, 5(1):1–33, 2017.

[46] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: Pipelined symbolic taint analysis. In *USENIX Security*, pages 65–80, 2015.

[47] Mitre. CWE-272. https://tinyurl.com/4xwzakyf, 2020.

[48] Mitre. CWEs. http://cwe.mitre.org/, 2020.

[49] NumPy.org. NumPy. https://github.com/numpy/, 2018.

[50] Openjpeg.org. JPEG. https://www.openjpeg.org, 2015.

[51] Oracle. JNI 6.0. https://tinyurl.com/2p94tvhp, 2009.

[52] Oracle. GraalVM. https://www.graalvm.org/, 2021.

[53] Pawel Piotr Przeradowski. Japronto a Python 3.5+ HTTP toolkit. https://tinyurl.com/v37799mp, 2017.

[54] Psycopg. Web. https://tinyurl.com/27buex7n, 2018.

[55] Pygame. Web. https://tinyurl.com/nhhdtj4f, 2020.

[56] PYPL Index. Web. https://tinyurl.com/2d6tdhs5, 2022.

[57] PyTables. Web. https://tinyurl.com/ywbr9jm3, 2019.

[58] Pytest. Web. https://tinyurl.com/45sajhuw, 2021.

[59] Python. 3.9.2 docs. https://docs.python.org/3.9, 2020.

[60] Python. Unit test. https://tinyurl.com/2fewd6ca, 2021.

[61] PyTorch. Web. https://tinyurl.com/5n8pfv9m, 2016.

[62] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *FSE*, pages 155–165, 2014.

[63] Micha Reiser and Luc Bläser. Accelerate javascript applications by cross-compiling to webassembly. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pages 10–17, 2017.

[64] Requests. Web. https://tinyurl.com/3b32x54k, 2020.

[65] Salt. Web. https://tinyurl.com/2p8wnmkr, 2020.

[66] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P*, pages 317–331, 2010.

[67] Gang Tan and Greg Morrisett. ILEA: Inter-language analysis across Java and C. In *OOPSLA*, pages 39–56, 2007.

[68] Web2py. Web. https://tinyurl.com/up2cw2rb, 2020.

[69] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. JN-SAF: Precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code. In *CCS*, pages 1137–1150, 2018.

[70] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Onward!*, pages 187–204, 2013.

[71] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, 2006.

[72] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. Python predictive analysis for bug detection. In *FSE*, pages 121–132, 2016.

[73] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, pages 569–584, 2012.

[74] Drew Zagieboylo and Andrew C. Myers. JLang. https://tinyurl.com/2tsrbkdt, 2020.

A More on POLYCRUISE Implementation

We implemented the DIFA engine for POLYCRUISE in C, consisting of five components: event decoder, memory database, dynamic information flow analyzer, and plugin support.

Following the event definition in §3.3.1, the decoder parses the events as a character stream to obtain definition-use pairs and event type for each event and associated statement.

The memory database is implemented to empower low-overhead online DIFA. Its high efficiency is achieved by two strategies: (1) lightweight implementation—we only implemented the necessary functionalities (e.g., memory pool, hash, data management, and data add/query/delete); (2) high memory utilization—for an average data size of 64bytes, the database can hold 10+ million pieces with 1GB memory; and (3) scalable memory pool—we used a scalable memory pool to avoid frequent memory allocations and releases.

The online dynamic information flow analyzer is implemented on top of the memory database while working incrementally—each execution event triggers an updating pass of DIFG. For the implementation of Algorithm 5, we overcame two major challenges, with the thread graph and due to language boundary, respectively.

To handle the large volume of events generated during multi-threaded executions, we implemented DIFG as a set of thread-level information flow graphs, such that (1) the events within a thread can be analyzed sequentially and (2) the DIFAE can compute intra- and interprocedural data dependencies in a specific thread graph upon receiving an event. To compute interthread data flow, we maintained a global cache shared by all thread graphs—for each memory write, the DIFAE updates the cache to keep the location of the latest write to the memory address, while for memory read the DIFAE obtains the location of the address's latest write and inserts a data flow edge from the write to the read. This heavy global search hurts the DIFAE's efficiency. To avoid this kind of excessive searching, we adopt a local-first strategy: when a local dependence is found, it will abandon the global search.

The other challenge is induced by language interoperations. The SDA ensures the static/dynamic instrumentation to cover all possible data flow paths across language boundaries. However, different data encapsulation and conversion between languages become barriers for dynamic data dependence computation during runtime. For instance, a process of converting parameters from Python to C through C extension is {Python type → Pyobject → C type} [59], while parameters flow from Java to C through Java native invocation is in the form of {Java type → jobject → C type} [51]. Instead of modeling all interface APIs at language boundaries to guide how to relate parameters together in different language components, we adopted a field-insensitive approach for parameter passing in foreign function invocations.

To illustrate, consider the example of Figure 12. Line 4 of the Python unit is defined as a criterion (source). Variable V1 flows along edge 1 to the callsite at line 6 and continues flowing along edge 2 into a PyObject args encoded from V1 and V2. Without knowing how to decode args (language-dependently), we conservatively add two data flow edges {3, 6} from the definition site of args to line 4 of the Python-C

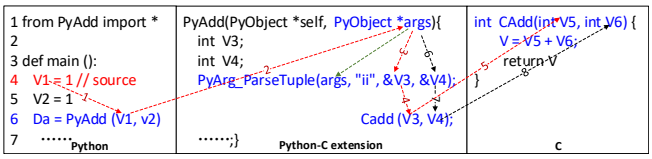

Figure 12: An example illustrating our field-insensitive approach to parameter passing in the DIFA of PolyCruise.

extension; edge 3 represents V1 flows into V3 while edge 6 indicates V1 flowing into V4 (which is a spurious data flow). Although this conservative computation causes imprecision, it ensures soundness and language independence hence the extensibility of our framework to support additional languages.

The DIFAE provides a set of interfaces for flexible application plugin development in C. Specifically, a user of PolyCruise can quickly implement a specific vulnerability detection plugin with about 100 lines of C code. Then, the user can easily integrate the plugin into our framework by specifying the location and entry point of, as well as the lists of sources/sinks relevant to, the plugin.

B Run-Time Slowdown Computation

Typically, we would compute the run-time slowdown of our dynamic analysis against a benchmark by dividing the observed/recorded execution time (T_{SDA}) of the *SDA-version* of the benchmark by the observed/recorded execution time (T_{pure}) of the *pure-version* (§6.2). However, in the course of our efficiency evaluation, we noticed that the Python interpreter has a nearly constant-time overhead in each execution at its initialization stage. The major reason is that the customized test frameworks (i.e., `pytest` and `unittest`) bring apparent side-effects. The test frameworks need to initialize the whole test set at each start and then execute the specified case. The initialization even took longer than the pure test case execution time. Moreover, as the test set grew, the extra overhead became even greater. However, in the expected use scenario of PolyCruise, the user would manually run it on a subject system against individual executions/tests, rather than having to automatically run an entire test set using the test framework. We used the test frameworks only to facilitate our evaluation experiments. Therefore, to accurately compute the slowdown (i.e., with respect to net execution times), we should leave out the initialization times I_{SDA} and I_{pure} from the observed ones. That is, the slowdown factor should be computed as $(T_{SDA} - I_{SDA})/(T_{pure} - I_{pure})$.

To obtain the initialization time costs, we took a simple approximation approach by considering that the shortest observed test execution mainly consists of the initialization stage only. Accordingly, for a given benchmark, we took the observed execution time of its fastest-running test as the initialization time cost in computing the slowdown factor for that benchmark (using the formula above).