RUN-TIME ANALYSIS AND SECURITY OF MULTI-LANGUAGE SYSTEMS

By

WEN LI

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2024

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of WEN LI find it satisfactory and recommend that it be accepted.

Haipeng Cai, Ph.D., Chair

Ananth Kalyanaraman, Ph.D.

Janardhan Rao Doppa, Ph.D.

Xiapu Luo, Ph.D.

ACKNOWLEDGMENT

I extend profound gratitude to all those who played a pivotal role in completing my Ph.D. My deepest appreciation is for my advisor, Professor Haipeng Cai, whose unwavering support, guidance, and encouragement were instrumental in the success of my research. Professor Cai's invaluable insights and constructive feedback were instrumental in molding the trajectory and focus of this dissertation.

I sincerely thank my dissertation committee members: Professor Ananth Kalyanaraman, Professor Janardhan Rao Doppa, and the external committee member, Professor Xiapu Luo from The Hong Kong Polytechnic University. Their insights, comments, and suggestions enriched this dissertation's depth and overall quality.

I express gratitude for the invaluable collaboration and guidance provided by Professor Xiapu Luo, Professor Long Cheng from Clemson University, Professor Ming Jiang from Tulane University, and Professor Li Li from Beihang University. Their support has been instrumental in advancing my research work.

I acknowledge the financial support from the National Science Foundation (NSF) (CCF-2146233) and Office of Naval Research (ONR) (N000142212111), facilitating the fieldwork and data collection underpinning this project.

Finally, sincere thanks go to all the co-authors who generously shared their time and insights for my research. Their contributions significantly enriched the depth and quality of this study in ways that I could not have achieved alone.

RUN-TIME ANALYSIS AND SECURITY OF MULTI-LANGUAGE SYSTEMS

Abstract

by Wen Li, Ph.D.
Washington State University
May 2024

Chair: Haipeng Cai

The contemporary software development landscape has witnessed a widespread integration of diverse programming languages, leveraging the specific advantages of each, such as the efficiency of C and the programmability of Python. This trend finds notable applications in prominent domains, including the Android operating system and advanced machine learning frameworks like PyTorch. However, adopting this multi-language approach has ushered in a series of great challenges for developers, necessitating the identification of robust solutions to tackle potential security vulnerabilities.

Traditional techniques such as program analysis and fuzzing, initially designed for single-language software, face limitations in effectively uncovering vulnerabilities in multi-language systems. Program analysis grapples with challenges in comprehending the intricate control and data flows across diverse languages, often resulting in incomplete vulnerability detection. Conversely, greybox fuzzing encounters difficulties adapting to the nuances of various languages, leading to incomplete code coverage and complications in reproducing identified vulnerabilities. The intricacies within runtime systems supporting multilingual software exacerbate the security clearance predicament, as these systems are often constructed using

multiple languages. This complexity adds an additional layer of difficulty for conventional security techniques, emphasizing the need for more adaptive and comprehensive approaches tailored to the unique challenges posed by the multifaceted nature of multi-language systems.

Within the scope of my dissertation, I endeavored to tackle the intricate challenges posed by vulnerabilities in multi-language software through a comprehensive and multifaceted approach. This approach entailed conducting extensive empirical investigations into vulnerability susceptibility, facilitating the development of dynamic cross-language information flow analysis. Recognizing the pivotal significance of comprehensive test input coverage, I devised an integrated greybox fuzzing methodology. This innovative approach integrates sensitivity analysis and comprehensive whole-system coverage measurements, significantly enhancing the efficiency of the fuzzing process and vulnerability identification. Furthermore, I focused on fortifying runtime security by proposing a novel two-level collaborative fuzzing framework tailored explicitly for Python language runtime. This contribution was pivotal in reinforcing the software system's foundational safeguards, ensuring a robust defense mechanism against potential security threats.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Dedication

This dissertation is dedicated to my family and friends, with special recognition extended to my wife, Wei Wei. Their unwavering love, encouragement, and profound understanding have been the cornerstone of my success throughout this challenging academic journey.

CHAPTER ONE

INTRODUCTION

In contemporary software systems, the incorporation of multiple programming languages has become commonplace, leading to developing a "multilingual" code base [102, 72, 103, 80]. Leveraging the unique advantages of different languages, such as the efficiency of C and the flexibility of Python, has been a standard practice in software development, sustaining its momentum for several decades. This approach has not only contributed to heightened productivity in the development process but has also improved the overall performance of the resulting software. However, this widespread adoption of multilingual software across various domains has also introduced additional security threats, making security concerns increasingly critical.

Despite the criticality and urgency, the support for ensuring the security of multilingual systems has remained largely inadequate, primarily due to the lack of comprehensive techniques and tools. Existing techniques like program analysis [88, 146] and fuzzing [99, 149, 95] all target single-language software. While operating on multi-language software, their functionalities get compromised by multiple challenges.

When considering single-language program analysis, the issue of comprehensive analysis that extends beyond language boundaries becomes a significant concern. Incomplete control and data flow analyses often lead to unsound analysis. Consequently, security applications, such as vulnerability detectors, built upon such unsound analyses, can produce lots of both false positives and false negatives. While several approaches are geared towards cross-language analysis [150, 83, 19, 160, 80, 53], yet many of them are narrowly focused on specific cases within the multilingual realm, particularly JNI programs, thus imposing certain limitations. Additionally, various other methodologies have practical challenges. For example, as a framework of dynamic taint analysis, Truffle relies on utilizing a customized JVM (GraalVM) to accommodate multi-language systems [74]. Nevertheless, implementing

a dedicated runtime for each non-Java language within this approach can be a complex and impractical undertaking. Moreover, the inherent resource-intensive nature of the VM-based technique renders it less scalable for real-world systems.

Moreover, in the realm of single-language fuzzing, the hurdle of incomplete coverage measurement presents a formidable challenge to the evolution of effective fuzzing techniques. Established fuzzing methods such as those focusing on C/C++ [46, 149, 31, 50, 48, 92] have recently expanded their reach to encompass various language units, yet their operations remain essentially confined within a single language unit. While these single-language fuzzers can be adapted for use with multilingual code, the approach treats other language units as enigmatic entities. The persistent issue of incomplete coverage measurement not only generates misleading outcomes but also significantly curtails the overall fuzzing effectiveness.

Furthermore, the inadequacy in security coverage for multi-language software at the application level poses a substantial risk, as potential vulnerabilities within the language runtimes hosting these applications could compromise the overall integrity of system security. While existing approaches have demonstrated efficacy in bug detection within specific language runtimes, a pivotal yet often overlooked facet is the role played by runtime libraries in the broader context of language runtime systems. Numerous fuzzing techniques, meticulously designed for compiler and language runtimes (examples include JSfunfuzz [143], TreeFuzz [118], Skyfire [158], and Fuzzil [55]), are primarily geared towards the generation of varied syntax-correct applications. However, they frequently neglect to account for the intricate interplay of application complexity and runtime libraries, thereby potentially limiting the effectiveness of security testing. This underscores the critical need for enhanced methodologies that consider the holistic dynamics of language runtime systems, addressing not only language runtimes but also the pivotal role of runtime libraries in fortifying the security posture of multi-language applications.

To tackle these challenges, this dissertation proposes a multifaceted and progressive approach. First and foremost, it undertakes systematic empirical studies on the security in-

vestigation of multi-language code, thereby enhancing the comprehension of multi-language systems and paving the way for further research directions. Building upon the empirical studies, a pioneering dynamic information flow analysis (DIFA) is conceived to facilitate fundamental cross-language program analysis. To overcome the limitations associated with input coverage in dynamic analysis, a comprehensive greybox fuzzing technique, incorporating sensitivity analysis and whole system coverage measurement, is introduced to target the testing of multi-language applications. Moreover, recognizing the significant impact stemming from identified security issues within language runtimes, a two-level collaborative fuzzing strategy is devised to thoroughly test the Python interpreter and runtime libraries, integrating a generation- and mutation-based fuzzing approach. These research endeavors have demonstrated the scalability and cost-effectiveness of detecting vulnerabilities within real-world multi-language applications and their respective language runtimes.

Through my PhD research journey, I have demonstrated a set of novel methodologies for practical security solutions for multilingual applications and their perspective language runtimes. In summary, I have made the following contributions.

- The empirical findings on the vulnerability proneness of multilingual code not only establish a robust statistical foundation affirming the susceptibility of multilingual code to vulnerabilities but also furnish comprehensive evidence elucidating the links between this susceptibility and actual instances of vulnerabilities. Leveraging these discoveries, I have formulated practical recommendations tailored for researchers, developers, and tool builders. These guidelines aim to enhance comprehension, analysis, and defense against vulnerabilities inherent in multilingual code. In addition to these insights, my contributions extend to an automated vulnerability classifier for commits, surpassing the accuracy of existing peer tools. I introduce the inaugural automated language interfacing mechanism detector, structured according to the first taxonomy of such mechanisms. Furthermore, I present a real-world vulnerability dataset that surpasses peer datasets in terms of size, accuracy, and diversity, encompassing a broader spec-

trum of projects. The introduction of these innovative tools and datasets provides immediate, tangible support for the implementation of my proposed recommendations in practical settings.

- I conceived PolyCruise, an extensible and scalable dynamic analysis framework designed for multilingual applications, specifically those developed in Python and C programming languages. This innovative framework leverages lightweight language-specific static analyses coupled with online language-agnostic dynamic analysis, marking a milestone as the first cross-language dynamic information flow analysis (DIFA). Additionally, I introduce PyCBench, an open DIFA test suite encompassing a diverse range of analysis features. This suite stands out as the first publicly available cross-language dynamic analysis benchmark.
- I conceptualized and created PolyFuzz, the pioneering holistic multi-language fuzzer. Its open-source and extensible design not only supports fuzzing across Java, C, and Python but also enables the extension of greybox fuzzing to other language combinations. Notably, the absence of multilingual fuzzing benchmarks posed a significant challenge to the technique evaluation. In response, I introduce the first benchmark suite for multilingual fuzzing, addressing a crucial gap in the community.
- I introduced PyRTFuzz, a pioneering two-level collaborative fuzzing technique designed specifically for Python runtime testing. Its open-source implementation and flexible design not only support greybox fuzzing within Python but also provide a foundation for extending this approach to other compiler and runtime systems. The methodology, which integrates generation-based compiler fuzzing with mutation-based application fuzzing, as demonstrated in PyRTFuzz, has broader applicability and can be extended to interpretation languages beyond Python.

1.1 Research Overview

In this dissertation, I address the challenges of vulnerabilities in multi-language software through a multifaceted approach. Beginning with empirical investigations into multi-language code [84, 87, 85], I progressed with ongoing research in runtime analysis and the security assurance of multi-language software, as illustrated in Figure 1.1. Specifically, my research encompasses a security analysis of application-level systems, including initiatives like PolyCruise and PolyFuzz. Additionally, I have delved into the domain of language runtimes with PyRTFuzz at system level.

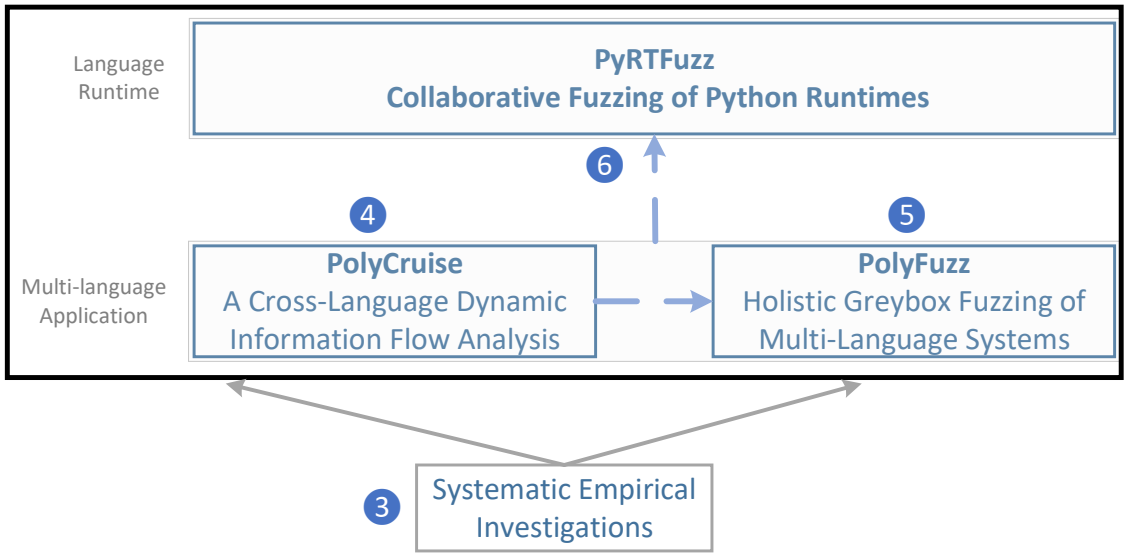

Figure 1.1 The research overview.

Based on empirical findings, I introduced PolyCruise, a dynamic information flow analysis (DIFA) method that observes program behaviors in real-time, addressing varied semantics among languages and scalability issues [86]. Using a language-independent symbolic representation (LISR), PolyCruise successfully uncovered 14 previously unknown vulnerabilities in real-world multi-language systems, emphasizing the critical role of the approach in bolstering the security of multilingual software.

To overcome the weakness of PolyCruise due to the limited code coverage of available test inputs, I developed PolyFuzz, a holistic greybox fuzzing methodology, which effectively tested multi-language systems [89]. PolyFuzz consistently outperformed single-language fuzzers, achieving significant code coverage enhancements (10%-52.3%) and uncovering more bugs (1-10 vulnerabilities) across multi-language and single-language programs, highlighting its effectiveness.

Shifting the focus to language runtime security, I developed PyRTFuzz, a two-level collaborative fuzzing framework for Python runtime [90]. PyRTFuzz combines generation-based fuzzing at the compiler level and mutation-based fuzzing at the application-testing level, having discovered 61 new, demonstrably exploitable bugs within the interpreter and runtime libraries. The scalability, cost-effectiveness, and potential for continued bug discovery underlines the promise of PyRTFuzz's collaborative two-level fuzzing approach, for extending its application to other language runtimes, particularly those of interpreted languages.

1.2 Dissertation Organization

In the subsequent sections of this dissertation, I delve into the techniques and fundamental concepts that underpin my research in Chapter 2. Following this groundwork, Chapter 3 presents extensive empirical studies on multi-language software construction and the vulnerability proneness of code written in multiple programming languages, as documented in previous research [84, 87, 85]. This chapter addresses the lack of a systematic empirical assessment of multi-language software structure and the interaction mechanisms among various programming languages.

The ensuing three chapters showcase technical contributions, featuring PolyCruise [86] in Chapter 4, PolyFuzz [89] in Chapter 5, and PyRTFuzz [90] in Chapter 6. These chapters delve into specific technical aspects and implementations of the research.

Chapter 7 delineates my envisioned future directions for research, offering a glimpse into potential avenues for further exploration. These insights converge in the concluding remarks

of Chapter 8, particularly elucidating implications for researchers, industry professionals, developers, and security analysts within the expansive domain of software security and testing.

CHAPTER TWO

BACKGROUND

2.1 Language Interfacing Mechanisms

Language Interfacing Mechanisms are integral for enabling seamless communication and interaction among different programming languages. As software systems grow in complexity and diversity, the need for languages to effectively collaborate has become increasingly vital. These mechanisms allow developers to integrate code from one language with that of another, leveraging the unique strengths of each language within a single application.

The two primary categories of Language Interfacing Mechanisms exist: uniform and language-specific approaches. Uniform approaches typically involve interprocess communication (e.g., Remote Procedure Call), enabling different language components to communicate based on predefined agreements or protocols. These mechanisms are generally language-independent and widely utilized in various middleware technologies. Conversely, language-specific approaches, facilitated through Foreign Function Interfaces, enable direct function invocations between language components. These methodologies necessitate a more profound comprehension of the complexities inherent in each language's runtime environment.

Widely used languages like Python, PHP, Java, and Ruby [49] all provide support for Foreign Function Interfaces concerning C modules, with variations in the specific interfaces employed across languages. For instance, Java interacts with C functions using native function calls and 'jobject' for parameter passing, while Python's interaction with C is notably more complex and flexible [133]. Python can load a C library through 'ctypes' and call C functions, emulating the 'dlopen' usage in the C programming language. Alternatively, a common approach involves constructing the C module as a Python extension, allowing Python to import and use the C extension as a Python module. Conversely, using built-in APIs of Python runtime, C can call a Python function, necessitating the conversion of C-type parameters to PyObjects. Similar mechanisms are observed in other representative

languages, such as Go's communication with C through the 'cgo' command and Ruby's communication with C through FFI.

Understanding the intricacies of these Language Interfacing Mechanisms is crucial for ensuring smooth communication and seamless integration between different components of a software system. Developers must be aware of the varying interfaces and interaction patterns between different languages to design effectively, develop and maintain complex software applications. Meanwhile, for researchers, handling the intricate interactions among different language units is essential for sound technical design.

2.2 Multi-Language Program Analysis

As an evolving research domain, investigations on multilingual programs primarily focus on three key directions:

(1) Unified intermediate representation layer. In analyzing monolingual programs, a crucial step involves translating the source or binary code into an intermediate representation (IR). This language-independent IR contains essential information, including data types and instruction operations, facilitating effective analysis [76]. Frameworks such as Soot [75] for Java and LLVM [76] for C/C++ are equipped to perform this translation. Consequently, when considering the analysis of multilingual programs, the concept of translating all language components into a uniform IR with a consistent syntax structure naturally arises (as shown in Figure 2.1).

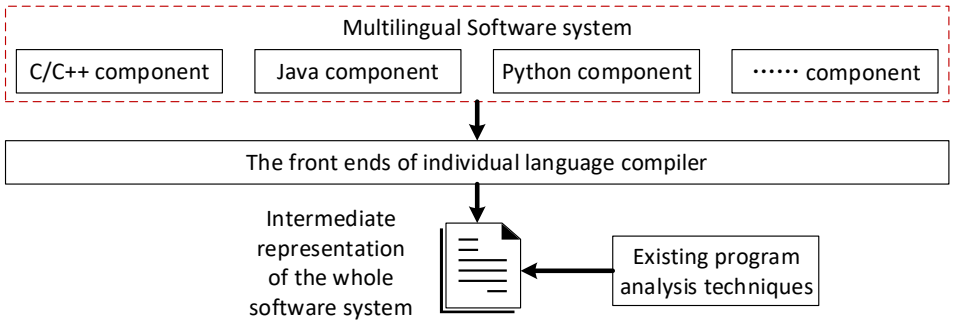

Figure 2.1 An intermediate representation layer for multilingual programs.

Various research endeavors are already in progress in this area. For instance, The pyLang [97] functions as a front-end tool for the compilation of Python programs into LLVM IR. Similarly, JLang [169] has succeeded in translating Java programs into LLVM IR, encompassing most of Java 7's language features, albeit excluding advanced reflection functionalities. In a separate effort, Arzt et al. [7] attempted to translate the Common Intermediate Language (CIL) of the Microsoft .NET framework into Jimple. However, this approach missed some crucial features, such as those pertaining to mixed-mode DLLs.

(2) Language-independent analysis techniques. From a technological perspective at the application level, existing language-independent techniques can be summarized into two aspects: (1) Techniques based on observation and (2) Techniques based on system emulators.

One representative of the former, ORBS [13], claims to analyze multilingual programs without the need for additional efforts. It accomplishes this by iteratively applying a process known as "delete-execute-observe" on the target system, obtaining a program slice until no further lines can be deleted. The resulting slice behaves the same as the original program, given a specific criterion. However, even its improved version [77] continues to face significant scalability issues, with obtaining a 5K program slice taking up to 10 hours.

The latter group has long been focused on dynamic analysis. Generally, such low-level technologies do not differentiate between language characteristics during runtime. Instead, they analyze the target system by tracking memory bytes with shadow memory [42, 37]. Despite their language-independent nature, these techniques have not been extensively reported for multilingual program analysis. One possible reason is their inability to capture details of vulnerabilities in software components developed using interpreted languages. For instance, they may report a JVM vulnerability caused by a specific application without recognizing any details about the application itself.

(3) Cross-language analysis techniques. Structurally, cross-language analysis (CLA) can be divided into three components: (1) Language-specific component (LSC), (2) Language-independent component (LIC), and (3) Analysis component (AC) as shown in Figure 2.2.

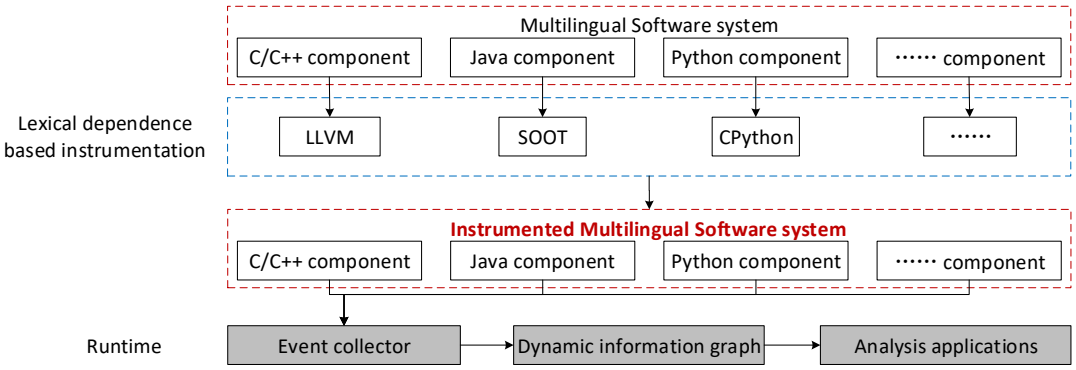

Figure 2.2 A general design of cross-language analysis framework.

The Cross-Language Analysis (CLA) approach leverages existing mature analysis frameworks (i.e., LSCs) tailored for specific languages, minimizing the effort required to obtain a language-independent intermediate representation, such as a forest of graphs in static analysis or an Abstract Syntax Tree (AST) in dynamic analysis, as seen in Truffle [74]. The Language Interconnection Component then links these intermediate representations based on the connecting semantics of each language component. Finally, the Analysis Component focuses on developing specific analysis algorithms for the entire software system. However, this technique still demands some engineering work for single-language analysis, and modeling the semantic connections between language components remains a challenge.

In this context, prominent work has largely focused on JNI analysis due to the extensive use of Java and Android. Previous efforts, such as those by Kondoh et al. [73] and Li et al. [83], detect bugs in native code by modeling semantic rules for JNI and native code interactions, employing data flow graphs and finite automated machines as intermediate representations. Similarly, Wei et al. [160] and Lee et al. [78] aim to construct comprehensive program data flow/call graphs with semantic summaries of all native functions, applying taint analysis or bug detection on these graphs. However, the current research is primarily limited to the static analysis of two languages, highlighting the ongoing challenge of implementing an effective and practical multilingual analytical framework.

2.3 Greybox Fuzzing

Greybox fuzzing [15, 100, 46] involves conducting lightweight program analyses on the targets and/or collecting execution feedback (such as coverage) to guide seed selection and/or mutation [98, 48]. The typical procedure of greybox fuzzing is outlined as below. (1) retains and operates a seed queue Q, (2) picks seeds from the queue Q based on a specific strategy; (3) mutates the seeds using diverse approaches (e.g., bit/byte flips, stacked tweaks, and splicing [98]); (4) executes the program using the mutated seeds, detecting vulnerabilities and, if necessary, updating the queue Q. and (5) repeats the cycle from step (2).

Greybox fuzzing techniques, including AFL [99], VUzzer [141], and AFL++ [46], leverage lightweight program analysis on the target programs, using execution feedback such as coverage data to guide seed selection and mutation strategies. The general iterative process of greybox fuzzing entails maintaining a seed queue that can be dynamically updated. Seeds are selected from the queue based on specific policies and then subjected to diverse mutations such as bit flips, arithmetic operations, and splicing. The fuzzer subsequently executes the target program with mutated seeds, reporting vulnerabilities and updating the seed queue as necessary. The process continues cyclically, refining the fuzzing strategy over time.

Serving distinct purposes, greybox fuzzing can be broadly categorized into two main types: coverage-based greybox fuzzing (CGF) and directed greybox fuzzing (DGF). CGF primarily concentrates on attaining extensive coverage of the target program by generating seeds through input mutation. The aim is to traverse previously undiscovered program statements, thereby increasing the overall code coverage rate. Prominent examples of CGF techniques include AFL [99] and its enhanced version, AFL++ [46]. On the other hand, DGF techniques are specifically designed to generate seeds capable of reaching specified or potentially vulnerable code. This may involve triggering known vulnerabilities with identified locations or assisting in the creation of proof-of-concept (PoC) scenarios for vulnerabilities [16, 23, 167]. The distinction between these two categories underscores the varied objectives

and strategies employed by greybox fuzzing techniques in the pursuit of effective vulnerability discovery and exploitation.

As an evolution of greybox fuzzing, data-flow-sensitive (DFS) fuzzing aims to leverage data flow features such as data dependence effectively to explore hard-to-reach code and uncover vulnerabilities while balancing efficiency and effectiveness. For instance, Greyone [48] introduced a fuzzing-driven taint inference (FTI) technique to infer semantic dependence between seeds and branch variables. Leveraging these dependence facts, it builds an input prioritization model to guide mutation and implements conformance-guided evolution to enhance seed updating and selection. Despite utilizing minimal data flow facts, Greyone has demonstrated a strong capability in detecting vulnerabilities.

2.4 Compiler Testing

Compiler testing is a complex task involving generating a diverse set of valid test programs, ensuring that the generated programs exercise various compiler parts for effective bug detection [24]. The generation of invalid programs can lead to their exclusion during the initial phases of the compilation process, making them less useful for comprehensive testing. To address these challenges, two primary strategies are commonly used: grammar-guided approaches and program mutation-based approaches [24].

Generation-based fuzzing techniques, including JSfunfuzz [143], TreeFuzz [118], and Skyfire [158], leverage existing samples to learn grammatical features and rules, enabling the generation of valid applications. These techniques use formal grammar of programming languages to ensure the syntactic correctness of the generated programs. On the other hand, mutation-based fuzzing techniques, such as Superion [159] and Fuzzil [55], modify existing programs based on their Abstract Syntax Trees (ASTs) or intermediate representations while considering the underlying grammar. By intelligently mutating existing programs, these techniques aim to create diverse and valid variations for testing purposes.

LangFuzz [63] is an example of a grammar-driven approach that generates random pro-

grams by assembling code snippets according to the rules defined by the grammar. Moreover, specific grammar-aware methodologies, such as those proposed by Chen et al. [28, 27], emphasize the use of generative models based on machine-learning techniques for effective compiler testing, particularly in the context of Java Virtual Machine (JVM) testing. Despite the effectiveness of these techniques in testing compilers and interpreters, they often overlook the critical role played by runtime libraries, which are integral components of the language runtime system. The selection of an effective strategy is determined by various factors, including the programming language, the type of compiler, and the defined testing objectives, with each approach having unique advantages and limitations.

CHAPTER THREE

SYSTEMATIC EMPIRICAL INVESTIGATION ON MULTILINGUAL CODE

Utilizing multiple languages in contemporary software development has become a common practice. To delve into the intricacies of how different languages are used and chosen in tandem, this research work presents an updated overview of the language profiles in multi-language software, through analyzing 7,113 active projects on GitHub over the past 5 years. The research identifies an increasing trend of incorporating 3 to 5 languages within a single project, along with a consistent preference for specific languages over time. Furthermore, the study reveals the considerable impact of language selection, project age, and team size on the susceptibility of multi-language software to code vulnerabilities, with a particularly notable effect on specific vulnerability categories. The exploration of various cross-language interfacing mechanisms serves as a key explanatory factor for these effects. In light of previous research, the results emphasize the significance of holistic language profile selections, rather than individual languages, in influencing the vulnerability proneness of software, highlighting the critical need for heightened attention to security threats across language interfaces. Moreover, These findings provide valuable insights for developers and language designers, aiding them in making informed decisions for language selection and design strategies.

3.1 Motivation

Numerous studies have thoroughly explored the panorama of programming languages within the realm of software development, shedding light on critical factors influencing language success [25] and popularity [14, 104, 142]. These studies have explored language interactions [14, 155], relationships, and evolutionary trends [25], offering valuable insights into the dynamics of programming languages over time. While extensive research has explored the impact of language usage on defect-proneness and maintenance in software systems, the majority

has focused on single-language systems. Insufficient attention has been dedicated to comprehending the intricacies of these dynamics within the realm of multi-language software, thereby underscoring an untapped potential for extended investigation and exploration.

Although some studies have investigated multilingual software, these have primarily centered on the prevalence of such projects and developer practices [1], often relying on surveys rather than empirical analysis of project artifacts [103, 151]. Limited research has explored the rationale behind language selections and combinations in multilingual projects, leaving a gap in understanding the underlying decision-making process. While a few studies have examined associations among chosen languages in multilingual software [102, 39], these works have not comprehensively delved into the justifications for language combinations with respect to project characteristics or functionalities.

Moreover, a substantial portion of existing literature has focused on individual programming languages [14, 104, 142, 25, 155, 170], with only a few studies addressing the challenges and implications of multilingual software development [103, 151]. Some recent research has hinted at potential security risks associated with multilingual code, but these findings have been limited to Java Native Interface (JNI) programs [160, 80, 64], leaving the broader context of vulnerability proneness in multilingual software unexplored. Overall, there is a need for more comprehensive and detailed studies to unravel the complexities and implications of employing multiple programming languages in contemporary software development, especially in the context of security vulnerabilities.

3.2 Approach

To enable the systematic empirical investigation of the open-source projects on GitHub, I developed a toolkit, PolyFax [85], which facilitates a comprehensive and systematic empirical investigation of open-source projects on GitHub. This toolkit is designed to streamline the process of data collection and processing, enabling efficiently gathering relevant information for analysis. The architecture of PolyFax is illustrated in Figure 3.1, providing an insightful

overview of its internal structure and functionality.

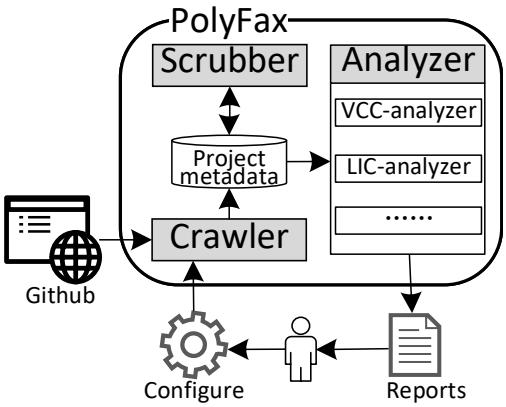

Figure 3.1 Overall design of PolyFax.

PolyFax operates in three main stages: Crawler, Scrubber, and Analyzer. Initially, the Crawler fetches repository profiles, clones projects, and retrieves historical commits to local storage. Subsequently, the Scrubber preprocesses textual information from all project metadata, such as project descriptions and commit logs. Finally, the Analyzer executes specific analyses for predefined purposes in the study. This includes vulnerability-fixing commit categorization, language interfacing mechanism categorization, functionality domain identification, and association analysis, all designed to achieve the study's investigative goals.

3.2.1 Vulnerability-fixing commit categorization

Vulnerability-fixing commit categorization (VCC) is based on the principle that commits aimed at addressing specific vulnerabilities often include corresponding keywords or phrases in their logs. This methodology is akin to the approach adopted in earlier studies, such as that by Ray et al. [142], where bug-fixing commits were identified using keyword analysis in commit logs. The VCC process involves two primary phases: (1) Vulnerability keyword summarization: This phase condenses the top 25 critical Common Weakness Enumerations (CWEs) [22] into three overarching categories: Porous defenses, Risky resource management, and Insecure interaction. It generates a set of security-related keywords or phrases associated

with each category. (2) Vulnerability keyword matching: In this phase, the FuzzyWuzzy technique [32] is employed to classify commit logs based on the identified keywords.

The process outlined in Algorithm 1 begins by retrieving the predefined categories (line 2) and cleaning the input commit using the pre-processText function (line 3). Subsequently, the algorithm calculates a match score between the commit and each category (lines 5-21). It retrieves and iterates through the keywords or phrases in each category (lines 8-20), dividing the commit log into n-grams for a given phrase or keyword length n (lines 9-17) and matching them using the FuzzyWuzzy algorithm (line 18). To ensure accuracy, the algorithm employs a minimum threshold score of 90 (line 6) and selects the highest score from all the phrases in a category as the final score for that category (lines 19-20). Finally, the algorithm returns the most relevant category as the vulnerability category for the given commit (lines 22-23).

Following the prescribed methodology, a comprehensive analysis revealed 141.38K commits designated as vulnerability-fixing out of the total 20.37 million commits. These were further categorized into three distinct classes, with 36%, 48%, and 16% assigned to each category, as illustrated in the detailed breakdown provided in Table 3.1.

To assess the efficacy of the approach, among randomly sampled 50 projects, I examined 500 commits of each project. I measured precision and recall based on manually established ground truth, as shown in Table 3.1. The ground truth was established through independent labeling by the authors, involving a thorough examination of the commit log, associated code snippets, and issue comments, if available. Disagreements were resolved through discussion to arrive at a final decision for each commit. Moreover, each identified vulnerability-fixing commit corresponds to a vulnerability that has been confirmed, going beyond matches of simple keywords or phrases.

3.2.2 Language interfacing mechanism categorization

In order to obtain a comprehensive grasp of language selection's security implications, it was imperative to ascertain the specific language interfacing mechanisms employed in every

Algorithm 1: Procedure of vulnerability-fixing commit categorization

Input: $Cmmt$: a commit including its log and code snippet

Output: $vCat$: the vulnerability category of $Cmmt$

```
1  Function classifyCommit (Cmmt)
2      VC ← initVulCategory () /* Categories with keywords/phrases */
3      Cmmt ← pre-processText (Cmmt) /* Tokenize, stemmatize, etc. */
4      CatScore ← φ
5      foreach Cat in VC do
6          Score ← 90 /* The minimum match score as the threshold */
7          PhraseList ← Cat.phrases /* Keywords/phrases of category Cat */
8          foreach Phrase in PhraseList do
9              N_p ← getWordNum (Phrase) /* 1 if Phrase is a keyword */
10             N_c ← getWordNum (Cmmt) /* Number of tokens */
11             xGramSet ← φ /* The set of n-grams in Cmmt; n=N_p */
12             Index ← 0
13             while Index < N_c do
14                 End ← Index + N_p /* Split Cmm into n-grams */
15                 xGramStr ← Cmmt[Index:End]
16                 xGramSet.append (xGramStr)
17                 Index++
               /* Match Phrase against Cmm's n-grams with FuzzyWuzzy */
18             Result = FuzzyWuzzy.extractOne (Phrase, xGramSet)
19             if Result.score > Score then
20                 Score ← Result.score
21         CatScore[Cat] = Score /* Keep the best match score with Cat */
22     vCat ← maxScoreCat (CatScore) /* Take the best-matched category */
23     return vCat
```

project. To accomplish this, I undertook an extensive manual investigation, examining codebases and relevant documentation thoroughly. This meticulous process enabled the development of a comprehensive taxonomy categorizing the various interfacing mechanisms. Leveraging insights derived from this taxonomy, I subsequently created an automated tool tailored for interfacing classification.

Table 3.1 Vulnerability categorization over the millions of commits.

Category	Security Vulnerability Description	Signatures (Keywords/Phrases)	%Cts	Prec	Rec
Porous defenses	vulnerabilities associated with defensive techniques that are either misused, abused, or simply ignored.	missing authorization, missing authentication, broken cryptographic, missing encryption, hard-coded credential, unnecessary privilege, excessive authentication, authorization bypass, user-controlled key, privilege escalation, etc.	36%	79%	81%
Risky resource management	the creation, usage, transfer, or destruction of important system resources is not properly managed.	data race, deadlock, buffer overflow, data leak, memory leak, exposed danger, stack overflow, memory corruption, integer overflow, untrusted control, etc.	48%	83%	86%
Insecure interaction	data is sent and received between separate components, modules, programs, processes, threads, or systems in insecure ways.	command injection, request forgery, SQL injection, reflected XSS, CSRF, unintended proxy, unrestricted upload, origin validation error, unintended intermediary, incomplete blacklist, etc.	16%	81%	88%

3.2.2.1 Definition of language interfacing mechanism

During the initial exploration aimed at understanding the fundamental mechanisms governing the interactions between the top languages, I categorized the language interactions into four overarching categories:

(1) Foreign function invocation (FFI). Foreign function invocation enables the seamless interaction between different programming languages. In this context, the host language provides an interface for ease of use by the guest language. For example, Java utilizes the Java Native Interface (JNI) to establish communication through native code programmed in languages such as C. Notably, these cross-language interfaces strictly adhere to standardized definitions outlined in various documentation sources, including JNI and Python extension [135]. Furthermore, these interactions are executed through direct function invocations, characterizing the distinctive features of this category.

(2) Implicit invocation (IMI). Implicit invocation serves as a cross-language interaction mechanism, relying on inter-process communications to facilitate communication among diverse components. One of the commonly adopted techniques in this category is the Remote

Procedure Call, which is often utilized to enable seamless communication among various language components.

(3) Embodiment (EBD). Certain languages might not exhibit any visible interface interactions within the software, yet they are interdependent and coexist to form a comprehensive system. We categorize this type of relationship between languages as interdependence. For example, in a web application, there might not be any FFI or IMI interactions between languages like CSS, HTML, and JavaScript. However, their collaboration is essential to ensure the functionality of the application.

(4) Hidden interaction (HIT). This category encompasses language selections where there is no apparent explicit interaction between the languages. I refer to it as "hidden interaction" as there might be an indirect data connection between various language components. For example, a crawler developed in Python could download GitHub projects to serve as inputs for an analyzer developed in C. While the two components are language-independent but rely on the interactive data flows.

3.2.2.2 Language interfacing classification

Based on pattern matching and finite state machine techniques, we constructed a rule-based classification model $\mathbf{C}$ in the following manner:

$$\mathbf{C} = (s_0, \mathrm{F}, \delta, \mathrm{S}, \mathrm{R}, \Phi), \quad s_0, \mathrm{F} \in \mathrm{S}, \quad \delta^* : \mathrm{S} \times \mathrm{R}^* \to \mathrm{S}$$

In the model, s_0 and F represent the initial and end state, respectively; S is the set of states; R is the set of patterns; δ is the function for state transition and Φ is a regular expression engine. With a sequence of inputs $\mathbf{I} = \{I_0, I_1, ..., I_n\}$, $\mathbf{C}$ obtains a set of matched rules $\mathbf{R} = \Phi(\mathbf{I})$, iff $\delta^*(s_0, \mathbf{R}) = \mathrm{F}$ then we say $\mathbf{C}$ accepts the input $\mathbf{I}$ or $\mathbf{I}$ is classified by $\mathbf{C}$.

Built upon the model, I constructed a language interaction classification engine (LICE), illustrated in Figure 3.2. Meanwhile, LICE focuses on the top twelve languages in the study's language selections with pre-defined classifiers. The engine operates through two interdependent components, including classifier configuration and classifier matching.

Algorithm 2: Classify a project based on language interfacing mechanism

```
Input: P: a multilingual program, C: the classifier set
Output: P_C: the classifier set matched
1  Function classifyProject (P)
2      C ← initClassifier ();                                  /* Initiate classifier set.  */
3      R ← compileRegex (C);                                   /* Compile all regex in classifiers.  */
4      R_C ← initMap (C);                                      /* Initiate a map from regex to classifiers.  */
5      foreach file in P do
6          R_M ← scanRegex (R, file);                          /* Obtain matched regexs.  */
           /* Obtain possible classifiers by matched regex.                                  */
7          P_Clf ← selectClassifier (R_C, R_M);
8          foreach Clf in P_Clf do
9              if classifyMatch (Clf, R_M) then
10                 P_C.insert (Clf);
11     return P_C;
12 Function classifyMatch (Clf, R_M)
       /* Initiate state queue by the initial state of Clf .                                 */
13     S_Q ← initStateQueue (Clf);
14     foreach r_m in R_M do
15         foreach S in S_Q do
               /* Try to move to the next state with input r_m.                              */
16             N_S ← nextState (S, r_m);
17             if N_S == NULL then
18                 continue
19             if isFin (Clf, N_S) then
20                 return TRUE;                                /* Reach the Fin state.  */
21             else
22                 S_Q.push (N_S);
23     return FALSE
```

(1) Classifier configuration (CC). CC is designed to establish a classifier set chain, comprising four categories: FFI, IMI, EBD, and HIT as defined above. For the FFI classifier set, I manually extracted programming patterns from the official language interface documents for all sixty-six pairs of the top twelve languages, resulting in the creation of twenty classifiers such as c_java, c_python, and java_python. In the case of the IMI classifier set, I examined

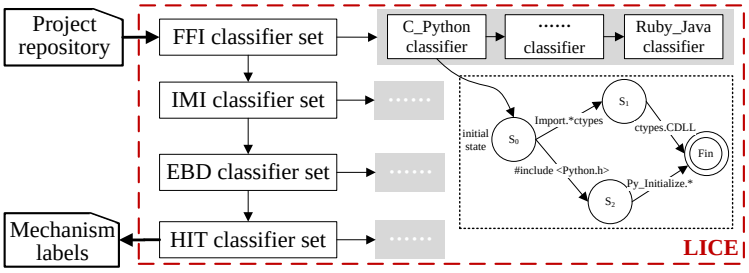

Figure 3.2 The overview of interfacing classification technique.

programming patterns on standard components facilitating remote calls like D-bus [121] and gRPC [56], leading to the development of seven classifiers. The EBD classifier set includes a single classifier for the interdependent existence of javascript, css, html in the top language selections. Additionally, in cases where the three classifier sets mentioned above cannot categorize a specific scenario, CC introduces a HIT category to account for instances where explicit programming patterns cannot be detected. Although LICE ensures interaction classification for the top twelve languages in the study, the implementation can be extended to support additional languages with relative ease.

(2) Classifier match (CM). Upon receiving a repository as input, CM conducts a comprehensive scan of the source files, making every effort to identify all types of language interactions utilized within the project. Algorithm 2 illustrates the process through which CM classifies a given project. During the initialization phase (lines 2-4), CM sets up all classifiers, compiles regular expressions, and establishes a mapping from each regex to a corresponding set of classifiers employing the regex. Subsequently, CM initiates the file scanning procedure. For every file, CM first employs a regex engine to identify all matched regex patterns. Through a process called dynamic selection (line 7), CM can narrow down the number of classifiers to be examined by utilizing the pre-defined regex-classifiers mapping. Thereafter, CM iterates through each classifier using the matched regex as input and adds any matched classifiers to the results. During the classifier matching process (lines 12-23), CM sequentially passes the matched regex into the state machine of the classifier, maintaining a state queue to store the matching context, hence capturing all accepted regex.

3.2.3 Functionality domain identification

To analyze the functional context of language utilization within multilingual software, I conducted an automatic identification of the functional domain for each project. For this purpose, I categorized the projects under investigation by employing inductive and axial coding analysis techniques to assess their functional attributes. [34, 105]. During the inductive coding step, I labeled randomly selected projects based on their associated non-code artifacts, such as project descriptions and README files, thereby compiling a codebook of identified labels. Subsequently, in the axial coding phase, I systematically grouped the projects into well-defined functional domains, assigning each project a specific code that most accurately reflects its primary functionality domain, as outlined in the developed codebook.

In particular, the manual analysis for categorizing project functionality involves two key phases: codebook creation and project categorization, detailed below.

(1) Codebook creation. To establish the codebook, I initiated a random selection of 1,500 projects, ensuring a statistically significant sample at a 95% confidence level (CL) and a 5% margin of error (ME). Three collaborators meticulously analyzed the documents of the selected projects, continuously refining the codebook and resolving any disagreements through collaborative discussions. Each project underwent a thorough evaluation, involving a comprehensive review of the project documents, assessment for compatibility with existing categories, and creation of new categories when needed.

The process of introducing a new category involved the initial definition of a label by the collaborators, followed by the creation of a comprehensive description for that category. To facilitate the labeling of future projects, the collaborators provided a concise summary of the characteristics exhibited by projects belonging to that particular category. As a result, the analysis yielded a codebook comprising 20 distinct codes, each accompanied by a summary description, as illustrated in Table 3.2. The codebook adopts a hierarchical categorization approach, where *level 0* encompasses codes representing different layers within the standard software stack, spanning from drivers to end-user applications. On the other hand, *level 1*

Table 3.2 Codebook for categorizing project' functionality domains.

ID	Level	Category	Description
1	0	driver	a software connecting OS and hardware devices
2	0	system	the interface between hardware and user applications
3	0	programming	tools for software development
4	0	middleware	software providing services to applications beyond OS
5	0	application library	libraries providing supports to other client applications
6	0	end-user application	applications providing functionalities to end users
7	1	word process	software for manipulating and designing text
8	1	database	software for operating database files and records
9	1	spreadsheet	software for operating data arranged in rows/columns
10	1	multimedia	software for operating audio/video files
11	1	presentation	software for creating a presentation of ideas via medias
12	1	enterprise	information system for organizations
13	1	information worker	software for users to create and manage information
14	1	communication	software for passing information from between entities
15	1	education	software for educational purposes
16	1	simulation	software modeling a real phenomenon
17	1	content access	software for accessing content without editing
18	1	application suite	a group of different but inter-related applications
19	1	email	software for using electronic mail
20	1	engineering	integrated software systems supporting development

accounts for the varied types of application software present within the system.

Table 3.3 Distribution of projects across functional domains.

Software functionality domain	Percentage of projects
end-user application	28.67%
application libraries	14.38%
middleware	13.34%
content access	8.75%
engineering/development	6.11%
education	5.16%
database	4.30%
programming	3.82%
multimedia	2.94%
word process	2.82%
system	2.10%
communication	1.60%
email	1.57%
presentation	1.13%
application suites	1.07%

(2) Project categorization. The entire set of projects was analyzed and coded by the five collaborators, in adherence to the codebook's guidelines. Notably, certain projects received multiple labels, reflecting their involvement in various functionality domains. The collaborators employed a negotiated agreement approach to ensure coding reliability, resolving discrepancies through discussion and consensus. The results of this comprehensive categorization process, detailing the distribution of projects in the SPC dataset across various functionality domains (Table 3.3).

3.2.4 Statistical methods

The study employed two key statistical techniques: association analysis and regression analysis using the NBR method.

Association analysis. In line with the established per-project functionality categories, a comprehensive examination was conducted to explore the relationship between these categories and the language combinations present in the dataset. The approach involved employing association rule mining techniques, aimed at identifying common if-then associations that comprised a software category as the antecedent and a language combination as the consequent. This involved the implementation of the Apriori algorithm [122] within the Mlxtend library [140], following a meticulous process consisting of three specific steps.

(1) Data formatting. The inputs are structured as a data frame, with one column containing project functionality categories and the other column containing language combinations.

(2) 1-hot encoding. The data frame is transformed in two steps: first, all unique data items (i.e., words) in the data frame are formed into a set of size n; then, each cell of the data frame is encoded as n bits by setting each item of the cell as 1 if it is in that set, followed by zero padding.

(3) Association computation. Using the encoded data frame, I initially calculate the support for each row (with min_support=0.01). Subsequently, I derive the association rules, representing the if-then associations, for the specified project set (with min_threshold=0.6).

In parallel, I analyze the association between project functionality categories and language combinations across the projects under study. This exploration aims to provide insights into the decision-making process and rationale behind language selection in developing multilingual software, focusing on the functionality domain.

NBR modeling. The NBR model was utilized to establish the relationship between vulnerability-fixing commits and various factors associated with multi-language software projects. This analysis was inspired by previous research in the field [142] and was particu-

larly well-suited for handling datasets exhibiting over-dispersion [5, 142].

In the study, each pairing of (language selection, project) was treated as an individual sample representing the overall landscape of multi-language programs. The analysis focused on several predictors, including project age (#days since creation), language selection size (#languages selected), and the language selection itself. To account for the unbalanced nature of the factors under consideration, weighted effects coding [139] was employed. This method assessed the relative impact of using a specific language selection to the response, relative to the dependent variable's weighted mean on all samples. Furthermore, to validate the results, a Chi-Square test [33] was conducted to assess the two-factor variables' dependences. In cases where dependence was identified, the effect size was computed using an r $\times$ c equivalent of the phi coefficient, as adopted in [142].

3.3 Empirical Results

3.3.1 Association between functionality & language selection

I investigated the correlation between developers' language choices in the studied software projects and their preferences within functionality domains, particularly emphasizing project topics. This analysis entailed calculating associations between functionality domains and language selections.

The outcomes of the association analysis, considering level-0 and level-1 functionality domains, are presented in Tables 3.4 and 3.5. Each entry in these tables represents a pair of (software domain, language selection), where support indicates the frequency of the pair in the dataset, and confidence signifies the conditional probability of the language selection given the domain. The inclusion of pairs is based on the criteria support $\geq 1\%$ and confidence $\geq 50\%$. These thresholds were chosen empirically. Lowering them would not yield additional pairs with at least weak association [62] (i.e., lift ≥ 1). The level of association is indicated by the lift factor: lift < 1 denotes mutually exclusive language selection and software domain, lift $= 1$ indicates no association, and lift > 1 implies an association with higher lift values

suggesting a stronger association.

Table 3.4 Association between level0's functionality domains & language selections.

Functionality Domain	Language Selection	Support	Confidience	Lift
application library	css-javascript-php	2.20%	7.11%	1.51
middleware	c-c++-python	1.66%	12.65%	1.41
end-user application	php-shell	1.50%	2.98%	1.39
end-user application	css-html-ruby	1.82%	3.62%	1.38
application library	css-html-javascript	8.80%	28.42%	1.33
end-user application	css-html-javascript	1.89%	14.56%	1.29
end-user application	java-kotlin	1.02%	2.02%	1.18
application library	objective c-ruby-swift	1.72%	5.55%	1.10
middleware	css-html-javascript-php	3.59%	27.35%	1.03
middleware	css-html-javascript-python	3.38%	25.71%	1.01
end-user application	c-c++-cmake	3.97%	7.88%	1.01

The analysis revealed a noteworthy relationship between language selection and the examined functionality domains, although certain associations appeared weaker in comparison. For instance, the association between css-html-javascript-php and middleware was relatively weaker at level 0, as shown in Table 3.4. Similarly, the association between c-c++-cmake and end-user application exhibited a comparable pattern. Moreover, at level 1, the relationship between css-html-javascript and multimedia displayed a moderate association, as outlined in Table 3.5. Notably, the majority of level-0 (3) and level-1 (7) domains demonstrated a strong correlation with language selections.

When examining the level 0 domains exclusively, the associations mined primarily relate to various types of applications, such as application library or end-user application. Moreover, the language selections demonstrated a degree of diversity within specific domains, suggesting that multiple language combinations are associated with a single domain. For instance, the development of end-user applications could involve choices like php-shell, java-

Table 3.5 Association between level1's functionality domains & language selections.

Functionality Domain	Language Selection	Support	Confidience	Lift
simulation	go-shell	1.28%	7.95%	3.24
multimedia	css-html-javascript	1.28%	48.00%	2.59
multimedia	css-html-javascript-php	1.38%	52.00%	2.19
multimedia	css-html-javascript-ruby	1.38%	52.00%	2.18
multimedia	css-html-javascript-python	1.28%	48.00%	2.10
simulation	makefile-python-shell	1.92%	11.92%	1.90
application suites	css-html-javascript-ruby	1.49%	45.16%	1.89
engineering/development	c-c++-cmake	3.83%	14.63%	1.86
spreadsheet	css-html-javascript	1.70%	30.19%	1.63
spreadsheet	css-html-javascript-php	1.92%	33.96%	1.43
spreadsheet	css-html-javascript-ruby	1.92%	33.96%	1.42
engineering/development	html-javascript-typescript	3.83%	14.63%	1.37
spreadsheet	css-html-javascript-python	1.70%	30.19%	1.32
engineering/development	objective c-ruby-swift	1.49%	5.69%	1.27
communication	css-html-javascript-python	2.66%	28.74%	1.26
engineering/development	html-javascript-python	2.56%	9.76%	1.19
email	css-html-javascript-php	3.30%	25.41%	1.07
application suites	css-html-javascript-python-shell	1.28%	38.71%	1.07
communication	css-html-javascript-ruby	2.34%	25.29%	1.06
email	c-c++-cmake	1.06%	8.20%	1.04

kotlin, or objective c-ruby-swift. This diversity can be attributed to the fact that software within the same domain is often developed within different software ecosystems or for different platforms, such as Android or iOS, influencing the selection of programming languages. In the case of the non-application domain, middleware, associations indicate that developers commonly opt for combinations like c-c++-python or languages such as php and python paired with css-html-javascript.

In terms of the level 1 domains, which represent specific kinds of end-user applications, the association analysis revealed a strong connection between the multimedia domain and languages paired with css-html-javascript. This language selection pattern is also commonly observed in the development of other types of end-user applications, including spreadsheet, communication, and email. These associations suggest that the combination of css-html-javascript is extensively used in the creation of various application software. Comparing these findings with the results in Table 3.4, the strong association between end-user application and css-html-javascript further emphasizes the consistency of the association analysis results between the two levels of functionality domains investigated. In contrast, level 1 functionality domain associations are generally stronger than those at level 0. This is attributed to greater language diversity within each higher-level domain, resulting in weaker associations with specific language selections.

3.3.2 Language selection's security relevance

Employing the NBR model, the study delved into the intricate relationship between security vulnerabilities and the selection of programming languages. Within this framework, the predictive encoding of language selections with weighted effects and the subsequent examination of the number of commits dedicated to rectifying security-related issues served as the focal point. The present section is dedicated to unraveling the security implications of language selections from dual perspectives. Firstly, the exploration encompasses the assessment of Differential Vulnerability Proneness among Language Selections, scrutinizing whether specific

language choices display heightened vulnerability susceptibility. Secondly, the investigation delves deep into the domain of Topic Traits of Security Relevance, aiming to establish a correlation between language preferences and distinct categories of security vulnerabilities.

Delving into the intricacies of multi-language software analysis, the study probes the potential links between language selections and the magnitude as well as the thematic aspects of developers' activities during the project development phase, specifically in addressing security vulnerabilities. This analytical approach implicitly acknowledges the evolutionary nature of software security within the dynamic landscape of multilingual environments.

3.3.2.1 Differential language selection's vulnerability proneness

Quantifying the comprehensive association between language selection and vulnerability proneness was achieved through the implementation of the NBR model. In this model, language selections were encoded with weighted effects serving as predictors, while the count of vulnerability-fixing commits was utilized as the dependent variable. Elaborated insights into this model can be found in Table 3.6.

The independent factors, namely, language selection size and project size, operate as control variables within the study. Although not the primary focus, their significance, as anticipated, remains conspicuous. All other independent factors, function as indicator variables, characterized as language selection variables (e.g., c-c++-shell). The coefficients associated with these variables serve to compare each language selection (per project) with language selections' weighted mean across all projects. The coefficient can be categorized into one of three distinctive classes: (1) Insignificance, as determined by p values exceeding 0.05. (2) Significance coupled with positivity. (3) Significance combined with negativity. A positive coefficient indicates that the specific language selection (e.g., c-c++-shell) is correlated with a heightened frequency of commits dedicated to rectifying security vulnerabilities. This correlation suggests a heightened vulnerability susceptibility compared to a language selection by average. Conversely, a negative coefficient implies that a language selection (e.g.,

Table 3.6 Language selection's vulnerability proneness.

Independent factors	Coeff.	Std. Error
(Intercept)	1.4672	0.051 ***
project age	0.0001	0.001 ***
language selection size	0.0483	0.004 ***
css-html-javascript	-0.0841	0.073
python-shell	0.2818	0.069 ***
go-shell	0.3234	0.077 ***
c-c++-python-shell	-0.1922	0.201
javascript-python	-0.0925	0.113
css-html-javascript-shell	0.1522	0.092 *
c-c++-python	-0.0300	0.162
objective-c-ruby	-0.2838	0.120 *
html-python	-0.4557	0.109 *
css-html-javascript-python	-0.0666	0.101
c++-python	0.4613	0.144 **
html-ruby	-0.1324	0.121
c-python	0.6253	0.222 ***
c-c++-shell	0.7641	0.098 ***
java-shell	0.2766	0.096 **
javascript-shell	-0.2041	0.084
javascript-php	0.1061	0.088 **
html-java	-0.1493	0.064

html-java) is less susceptible to vulnerabilities than the average case, signifying a decreased likelihood of prompting vulnerability-fixing commits.

In a nutshell, the coefficient linked to the language selection variable serves as an indicator of the expected change in the logarithm of the dependent variable (i.e., #vulnerability-fixing commits), assuming all other independent variables remain constant. By establishing a base factor (the mean of anticipated changes across all language selections), denoted as κ, and with a specific coefficient for a given language selection as γ, predictions can be made using the formula: $\mathbf{N} = \kappa \times e^{\beta}$. To illustrate, consider a project with an average language selection that reports five vulnerability-fixing commits among all types of commits. The expected number of vulnerability-fixing commits would be emph$5 \times e^{0.7641} = 10.74$ when utilizing c-c++-shell, significantly surpassing the average of five. Conversely, opting for html-python would result in fewer vulnerability-fixing commits ($5 \times e^{-0.4557} = 3.17$).

3.3.2.2 Topic traits of security relevance

To gain a deeper understanding of this connection, I explored how language selection aligns with distinct vulnerability categories, considering the thematic traits present in the commit data, such as logs and altered code. To achieve this, I constructed individual NBR models for each of the three categories (as indicated in Table 3.1). These models mirror the overarching relationship model, with the sole difference being that the dependent variable represents the count of vulnerability-fixing commits associated with the specific category. Refer to Table 3.7 for detailed information on these three models, denoting statistical significance with marks like ***p$<$0.001, **p$<$0.01, *p$<$0.05.

The deviance observed in each category-specific NBR model is notably smaller when compared to the deviance in the general vulnerability proneness model presented in Table 3.6. This discrepancy suggests that language selection has a more pronounced impact on the susceptibility to specific vulnerability categories than on vulnerability susceptibility overall.

Porous defenses. In this category, vulnerabilities arise from security defense technique

Table 3.7 Language selection's proneness to the three vulnerability categories.

Independent factors	Porous defense		Risky resource mgmt.		Insecure interaction	
	Coeff.	Std. Err.	Coeff.	Std. Err.	Coeff.	Std. Err.
(Intercept)	-2.167	0.157 ***	0.4746	0.070 ***	-2.7066	0.149 ***
project age	0.0001	0.001 *	0.0002	0.009 **	0.0087	0.013
language selection size	0.0811	0.010 ***	0.0624	0.005 ***	0.0767	0.010 ***
css-html-javascript	-0.3234	0.218	-0.1741	0.105 *	0.3042	0.195 *
python-shell	0.5527	0.212 **	0.3627	0.093 ***	0.3159	0.189
go-shell	0.4895	0.221 *	1.0764	0.099 ***	-0.5575	0.264 *
c-c++-python-shell	-0.7087	0.641	-0.6020	0.249 *	-2.1799	0.180
javascript-python	0.7464	0.339 *	-0.8805	0.161 ***	1.8017	0.333 ***
css-html-javascript-shell	0.8216	0.281 **	0.1136	0.132	1.2334	0.301 ***
c-c++-python	-0.2442	0.509	0.0927	0.202	1.8819	0.295
objective-c-ruby	-0.2630	0.424	0.0855	0.170	-1.4854	0.626 *
html-python	0.0735	0.315	-1.0742	0.163 ***	0.5449	0.287
css-html-javascript-python	0.9098	0.306 *	0.2371	0.144	-1.6434	0.308 *
c++-python	-0.1056	0.468	1.4384	0.181 ***	-0.8860	0.583
html-ruby	-0.4525	0.357 *	-0.6938	0.188 ***	1.6767	0.289 ***
c-python	-0.9907	0.714	1.2994	0.279 ***	-1.7480	0.253
c-c++-shell	-0.3065	0.315	1.5537	0.123 ***	-1.8835	0.443 ***
java-shell	-0.3145	0.316	0.7744	0.124 ***	-1.0087	0.367 **
javascript-shell	-0.4194	0.264	-0.2721	0.120 *	-1.5733	0.294 ***
AIC	25342		26358		13278	
BIC	25575.03		26591.08		13511.08	
Log Likelihood	-12634		-13142		-6602	
Deviance	7962.3		7679.5		4327.3	

misuse or absence, like 'missing authentication for critical function'. The study confirms a strong link between language selections and these vulnerabilities, consistent with the overall findings (Table 3.6). Certain choices, like css-html-javascript-python, showed notable susceptibility, especially python's impact when combined with web languages. Conversely, selections (e.g., html-ruby) exhibited a negative impact on secure stance.

Risky resource management. This vulnerability category, accounting for 48% of the vulnerability-fixing commits, results from mishandling vital system resources, as outlined in [101]. Instances include 'integer overflow/wraparound' and 'uncontrolled format string'. Notably, language selection's impact on these vulnerabilities outweighed its influence on the 'porous defenses' category. An exception was observed with java-javascript, it significantly affected the latter but had minimal effect on resource management vulnerabilities. Most language selections associated with these vulnerabilities above the average include c and c++, known for their susceptibility to memory-related vulnerabilities [142, 88, 111].

Insecure interaction. Within this category, 16% of the addressed issues are attributed to vulnerabilities arising from insecure data exchange between components, modules, and systems [101] (e.g., 'cross-site request forgery,' 'cross-site scripting'). Despite its relatively lower impact, language selection still plays a significant role, particularly in web language combinations such as javascript-php and java-javascript. These vulnerabilities frequently exploit these languages' interfaces, with 'cross-site scripting' often involving injected javascript code [145]. Understanding these dynamics is crucial for effective mitigation strategies in security analysis.

In conclusion, the research revealed a robust connection between vulnerability susceptibility and language selections in the studied projects. Moreover, language selection exhibited a stronger association with specific vulnerability categories rather than vulnerabilities in the overall category.

3.3.3 Factors contributing to the relevance

Previous findings have established a statistically significant connection between language selection and the security of the selected software. However, the strength of this association alone does not provide a comprehensive understanding of the underlying causes. Earlier research has suggested that cross-language interactions may represent critical vulnerability points in multi-language systems generally [103]. Similarly, inter-language dependencies have been identified as contributors to the vulnerabilities in JNI (Java Native Interface) applications specifically [53]. Consequently, the investigation delved into examining the correlation between cross-language interfacing mechanisms and the resulting vulnerability of the cross-language code. This analysis aimed to elucidate the effects of these interactions on the security relevance of the language profile, thereby explaining the observed relevance. I commenced by providing an overview of the diverse mechanisms employed in cross-language interfacing, followed by an exploration of the same two dimensions of security relevance as those studied in previous research. Furthermore, it is plausible that the security relevance of language selection is influenced by the individual characteristics of the chosen languages. Thus, I also scrutinized how each language contributes to the vulnerability susceptibility of multi-language code.

3.3.3.1 Overview of interfacing mechanisms

On each project, PolyFax conducted a comprehensive analysis of the interfacing mechanisms, uncovering instances where language selections exhibited hybrid mechanisms. The investigation revealed the presence of eight distinct combinations of mechanisms, distributed independently as follows: pure FFI 8.68%, FFI IMI 24.55%, FFI EBD 1.19%, IMI EBD 23.87%, FFI IMI EBD 1.26%, pure IMI 29.23%, pure EBD 5.87%, and pure HIT 5.36%.

Implicit interfacing emerged as the prevailing approach, accounting for 86% of the projects utilizing IMI, EBD, or both. This dominance can be ascribed to multiple factors. Firstly, a limited subset (20/66) of the language pairs considered during the development of PolyFax

to support FFI, primarily due to the scarcity of languages that support foreign functions. Additionally, the inherent advantages of implicit interfacing, including reduced component coupling and simplified implementation complexities, contributed to its widespread adoption. This trend was further reinforced by the availability of robust frameworks like gRPC [56], facilitating seamless interfacing between languages such as c-python-java, and ruby. Furthermore, a notable prevalence of EBD was observed, predominantly driven by projects employing the language combination {javascript-css-html}. This observation aligns with earlier research emphasizing the prevalent use of general-purpose programming languages in conjunction with domain-specific languages [103].

3.3.3.2 Differential vulnerability proneness of language interfacing

PolyFax performed an in-depth analysis of interfacing mechanisms in each project. This analysis led to the identification of language profiles characterized by hybrid mechanisms. In total, eight distinct combinations of these mechanisms were identified and treated as independent variables in the corresponding NBR models, as outlined in Table 3.8.

Table 3.8 Interfacing mechanism's vulnerability proneness.

Independent factors	Coeff.	Std. Error
FFI	0.2817	0.092 **
FFI IMI	0.1676	0.095 *
FFI EBD	-0.1454	0.415 *
FFI IMI EBD	0.7576	0.380
IMI	0.6441	0.164 ***
IMI EBD	-0.3059	0.092 *
EBD	-0.0811	0.089
HIT	0.4998	0.498

The NBR analysis highlighted the significant impact of language interfacing, whether

explicit (e.g., FFI) or implicit (such as IMI, either alone or combined with FFI), on the vulnerability susceptibility of multilingual code. Notably, the three mechanisms (FFI, FFI IMI, IMI) that exhibited significant vulnerability-proneness constituted the majority (53.1%) of the language profiles within the dataset. This outcome underscores the considerable influence of how selected languages interact within multilingual code on their vulnerability. It is a well-established fact that the use of multiple languages can render a system more susceptible than employing a single language alone [53]. The findings serve as a complement to this existing understanding, offering compelling evidence of the language interfacing mechanism's pivotal role in determining the overall security implications of utilizing multiple languages.

The regression analysis revealed three key indicators (FFI, FFI IMI, IMI) with notably positive and significant coefficients associated with the response variable, namely, the number of security vulnerabilities. This finding underscores the strong vulnerability proneness of language interactions, highlighting both direct and indirect interaction impacts.

Based on our comprehensive survey and analysis, we identified two plausible reasons contributing to the vulnerability of multilingual software: (1) Inherent complexity in developing multilingual systems: Previous research has confirmed the challenges developers face in comprehending and maintaining multilingual systems, especially concerning cross-language modifications [103]. (2) Insufficient technical support for quality assurance in multilingual programs: A survey of 1,000 research papers on program analysis published over the past five years in the ACM digital library indicated that a mere 0.8% of these publications were related to cross-language analysis. Moreover, the majority of these studies were limited to language selections such as c and java [160, 80]. This scarcity hinders the development of cross-language analysis tools necessary to ensure the quality of multilingual software.

3.3.3.3 Topic traits of interfacing effects

To gain insight into the specific effects on individual vulnerability categories, I constructed a separate NBR model for each category. The detailed results of the three models are presented

in Table 3.9 (marks: ***p<0.001, **p<0.01, *p<0.05).

The findings revealed that the three most vulnerable mechanisms overall were closely linked to each of the two prevalent vulnerability categories (refer to Table 3.1), which significantly contributed to the overall association's strength. Intriguingly, it became apparent that both FFI and FFI IMI displayed below-average vulnerability proneness concerning 'Insecure interaction' vulnerabilities. This observation stemmed from the fact that these vulnerabilities were primarily associated with web languages, which do not typically engage with FFI mechanism for interactions.

Table 3.9 Interfacing mechanism's proneness to the three vulnerability categories.

Independent factors	Porous defense		Risky resource mgmt.		Insecure interaction	
	Coeff.	Std. Err.	Coeff.	Std. Err.	Coeff.	Std. Err.
(Intercept)	-1.2328	0.152 ***	-1.5902	0.059 ***	-2.0517	0.096 ***
FFI	0.5434	0.315 **	0.4733	0.109 ***	-1.3708	0.180 *
FFI IMI	0.3562	0.321 *	0.4758	0.114 ***	-1.1683	0.239 *
FFI EBD	1.8357	0.266 *	0.7160	0.525	1.0950	0.634 *
FFI IMI EBD	0.3540	0.471	0.8610	0.487 *	-2.0167	1.186
IMI	1.1156	1.368 **	0.7457	0.531	0.6490	0.643 *
IMI EBD	-0.4470	0.320	-0.3917	0.113 **	-0.3215	0.241
EBD	-1.6342	0.312 *	-0.3942	0.108 ***	0.3044	0.233 **
HIT	-0.0088	0.270	1.2619	0.126 ***	-0.1634	0.642
AIC	25082		26224		13330	
BIC	25163.89		26305.89		13411.89	
Log Likelihood	-12528		-13099		-6652	
Deviance	7521.4		7374.5		3897.1	

This explanation finds support in the notable positive association between the 'Insecure interaction' vulnerability category and EBD, the predominant interfacing mechanism across web languages, such as javascript-php. Similarly, this rationale further elucidates the significant impact of FFI EBD on this particular vulnerability category. To validate this, we

conducted a manual examination of 10 projects utilizing {FFI EBD} and found that 7 of these projects employed the language selection {c-c++-css-html-javascript}, thereby emphasizing the prevalence of web languages contributing to 'insecure interactions'.

In conclusion, certain interfacing mechanisms demonstrated a higher susceptibility to specific vulnerability categories, yet the per-category effects of all mechanisms remained consistent with their overall vulnerability impacts. The vulnerability susceptibility to 'insecure interaction' vulnerabilities was notably associated with the utilization of EBD. However, the strength of this association was counterbalanced by the application of IMI. Additionally, the mechanisms that exhibited the highest vulnerability proneness overall (FFI, IMI) were also most susceptible to 'porous defense' vulnerabilities.

3.4 Implications

Attention to cross-language vulnerabilities. Significant vulnerability-proneness in multilingual code has been identified and confirmed by extensive case studies [107, 160, 80, 2, 64]. However, research on these vulnerabilities remains scarce. Out of over 1,000 papers on program analysis reviewed from the ACM digital library's recent publications, only 0.8% were generally related to cross-language analysis. Notably, this specific language combination did not rank among the top 20 language selections [107, 160, 80, 2, 64]. Addressing this critical gap and urgent issue requires the research community's investment in tools and studies targeting cross-language defects, transcending the limitations of Java-C (JNI) software.

Practical recommendations for researchers. Previous studies on GitHub, such as those by Ray et al. [142] and Berger et al. [11], have indicated the minimal influence of individual languages on software susceptibility to defects. However, this research sheds light on the substantial effects of language selection on the vulnerability proneness of multilingual code. These effects were strongly correlated with language interfacing mechanisms, particularly FFI and IMI, enabling direct data interoperations through code-level invocations. Thus, prioritizing the examination of multilingual code using these mechanisms to address

explicit (e.g., via foreign functions) and implicit (e.g., via RPC) cross-language interactions is strongly recommended.

Practical recommendations for tool builders. When creating tools for multilingual security, it's advisable to combine machine learning (ML) and deep learning (DL) models with program analysis approaches. These models can incorporate features related to language selections, such as size and language-specific properties, as well as interfacing details like mechanism category and interface parameters. Training and validating these models on a comprehensive vulnerability dataset can enhance their effectiveness. Given the variety in language choices and interfacing techniques, focusing on language-independent analyses in code analysis approaches—such as computing cross-language information flow for identifying multilingual vulnerabilities—becomes crucial.

Practical recommendations for developers. Certain language selections (e.g., c-python) were notably more vulnerable than others, as were specific interfacing mechanisms (e.g., FFI, IMI) compared to others. Opting for less vulnerable selections and mechanisms can bolster the security of multilingual code. Additionally, using less vulnerability-prone mechanisms for interfacing between selected languages is recommended. Prioritizing selections and mechanisms less susceptible to specific categories of critical vulnerabilities is also advisable, based on Tables 3.7 and 3.9.

3.5 Related Work

Study of quality effects of languages. Recent research has explored the fault proneness of design smells in multi-language software systems [2] and the impact of inter-language dependencies on code quality [53]. However, these studies were restricted to the Java-C language pairing and a small number of sample projects, typically 10 or fewer [1]. Mayer's research [103] highlighted that most developers encountered at least one bug related to cross-language linking, and using multiple languages increased the complexity of bug fixing. Some studies have focused on individual languages rather than language combinations [142,

170]. In contrast, our work stands out by investigating the security implications of language selection in real-world multi-language projects. Conversely, prior studies have examined the vulnerability proneness of single-language projects in the context of general code defects, including security defects, but not specifically in terms of security vulnerabilities [72, 11]. Such results can be compared with and complement our findings.

Analysis of language interactions. Bissyandé and colleagues conducted a study to quantify the proximity between various languages, offering valuable insights into language interoperability [14]. Additionally, separate investigations focusing on polyglotism revealed strong associations among languages, highlighting specific language combinations commonly utilized in practical contexts [151, 102]. In my research, I approach language interactions from a distinct perspective, delving deeper into the intricate relationships between languages within selected contexts. Furthermore, my study uniquely shifts the focus from the mere co-occurrence of languages to an exploration of the underlying security implications of language interactions, particularly through interfacing.

Commit-based vulnerability identification. Similar to classifying vulnerabilities in PolyFax, previous studies have also delved into identifying vulnerabilities in code repositories through commit analysis. For example, D2A employs static analyzers and learning-based classifiers to detect vulnerability-fixing commits [171]. VCCFinder uses code metrics and GitHub metadata to train an SVM classifier for labeling vulnerability-contributing commits [123]. Another tool leverages commit messages and bug reports to classify vulnerability-related commits [172]. Notably, our approach demonstrated superior accuracy, achieving 83% compared to 53% by D2A, 36% by VCCFinder, and 50% by [172] for commit classification.

Real-world vulnerability datasets. The study dataset of vulnerability-fixing commits, totaling 141,380 instances, surpasses D2A's dataset [171] in size (18,653 commits) and accuracy (30%). Additionally, it complements smaller, C/C++ focused CVE-based datasets such as BigVul [44] and CVEfixes [12].

CHAPTER FOUR

CROSS-LANGUAGE DYNAMIC INFORMATION FLOW ANALYSIS

Contemporary software ecosystems are predominantly multilingual, comprising integral code components scripted in diverse programming languages [102, 72, 103, 80]. The last decade has witnessed an upsurge in the pervasiveness of multilingual software, marked by an increase in the language number incorporated into each system [91]. Given the pivotal character in today's digital landscape, there is a critical imperative to methodically ensure the security of multi-language software. However, the current landscape reveals a conspicuous absence of comprehensive techniques to safeguard multilingual systems. Unlike their single-language counterparts, vulnerabilities in multilingual systems may not only reside within each language component but also manifest *at and across* the interfaces connecting various language units. Consequently, a holistic (cross-language) approach to validate multilingual programs against insecure properties becomes indispensable.

In response to this challenge, I conceived PolyCruise, an innovative cross-language dynamic information flow analysis tailored for multilingual systems. This approach seamlessly integrates minimal language-specific static analysis along with language-agnostic dynamic analysis, facilitating the identification of diverse vulnerability types. PolyCruise underwent successful validation on 14 previously unknown security issues on real-world multi-language systems, showcasing scalability, cost-effectiveness, and efficacy in discovering vulnerabilities.

4.1 Motivation

The interaction mechanisms between language components can be categorized into two main types. The first type is the uniform approach, which involves interprocess communication. This approach is universally applicable and not specific to any particular language, enabling communication based on an agreed-upon protocol, such as Remote Procedure Call (RPC). The second type is the language-specific approach, known as the foreign function interface

(FFI), which facilitates direct function invocations between language components, making it strongly language-dependent. Empirical studies have shown that popular languages such as Java, Python, PHP, and Ruby support FFI for C modules according to their official documentation. However, the interfaces vary significantly across languages. For example, Java interacts with C functions via native function calls, while Python's interaction with C is more complex and variable, often employing techniques like using ctypes or building C modules as Python extensions.

In the context of static analysis, the straightforward approach involves modeling all the language interfaces and then integrating the data flow at the boundaries between languages. However, the diverse interfaces and the dynamic features of certain languages can challenge this method. On the other hand, a dynamic analysis engine can conservatively link the data flow across languages by ensuring that the instrumentation covers all potential data flow paths. This dynamic analysis approach leverages the sequential execution events within a thread during runtime to establish connections between different language components.

The necessity and challenges of dynamic cross-language analysis are exemplified through three code snippets, depicted in Figure 4.1. In snippet (a), Python calls a C function using ctypes. An issue related to least privilege violation is present in line c3 of the Ca.so file, as data flows from line p5 of the Pa.py file. Exploiting this vulnerability enables an unauthorized intruder to gain access to files without proper authorization. Single-language analyzers on the Python side fail to detect this vulnerability, as the data flow is terminated at the boundary. Snippet (b) demonstrates Python's bidirectional interaction with C through a Python extension. Sensitive data flows from line p3 to line c9 (sink point) along the red lines. Both single-language analyzers are unable to identify this vulnerability, as they cannot construct a complete call graph in this context. In snippet (c), Python interacts bidirectionally with C through a Python extension, where two invocations of the C function are implicit. A static analyzer is unable to detect the presence of these function calls, making it impossible to identify the information leakage. Cross-language analysis is crucial

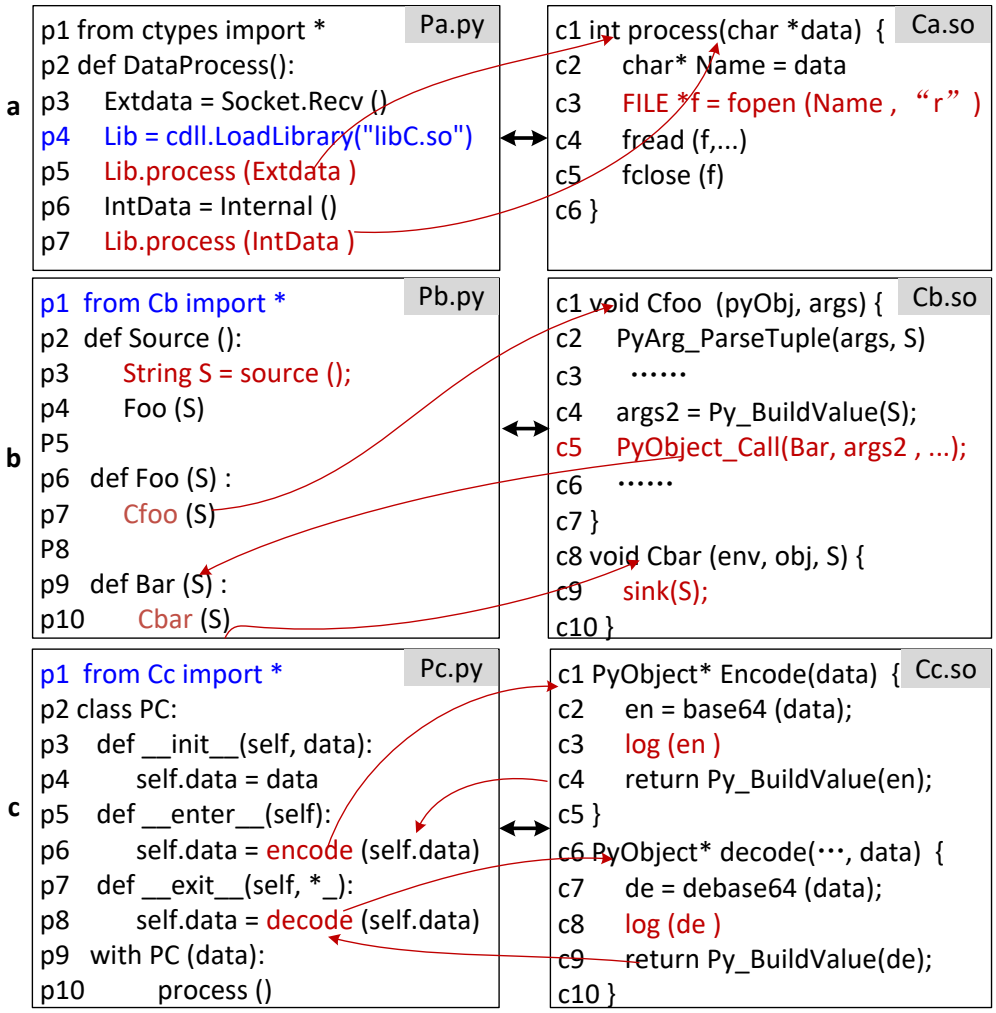

Figure 4.1 An illustration of cross-language vulnerabilities in Python-C program.

for vulnerability detection in multilingual programs. However, static approaches have clear limitations in handling different language interfaces, as seen in (a) and (b), and in accounting for the dynamic characteristics of the languages, as observed in (c).

The provided code snippets underscore the critical role of comprehensive, dynamic cross-language analyses in detecting potential security vulnerabilities resulting from interactions between distinct programming languages.

4.2 Approach

To enable practical security support for multilingual software, I undertook a dynamic information flow analysis (DIFA) approach, a fundamental technique in various security applications. However, multilingual DIFA faces significant challenges, including the disparities in language semantics and the need for cost-effective analysis in large-scale systems. To address these challenges, I developed a hybrid analysis strategy in our framework, utilizing language-independent symbolic representations and a language-agnostic dynamic analysis. This approach allows to compute data flow paths while accommodating diverse language characteristics, ensuring scalability and efficiency. The prototype, named PolyCruise, leverages symbolic dependence analysis for approximate dependencies and generates a dynamic information flow graph (DIFG), enabling various vulnerability discovery through plugins.

4.2.1 Overview of PolyCruise

Figure 4.2 presents an outline of PolyCruise, focusing on its approach to analyzing multilingual programs. The framework operates through two distinct phases, both integrating static and dynamic analyses.

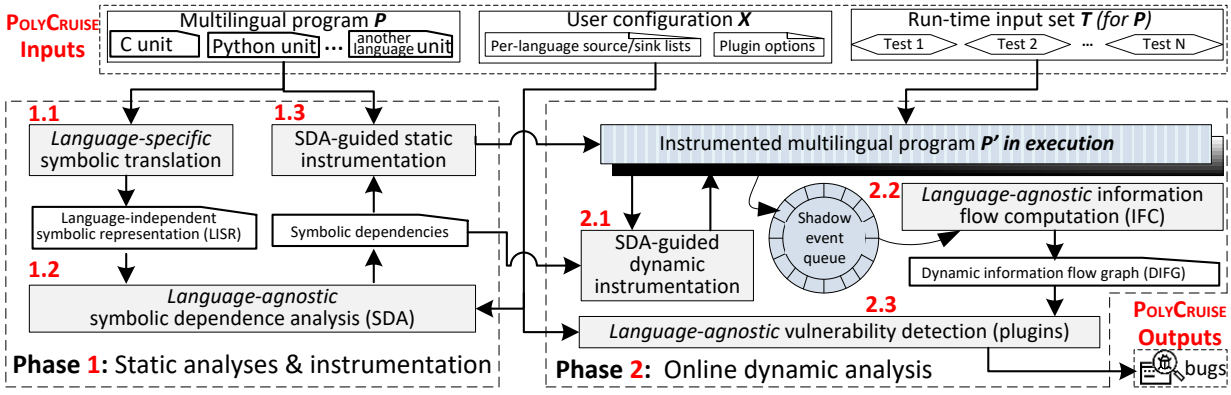

Figure 4.2 The overall design of PolyCruise.

In the initial phase, PolyCruise processes the program's language units into a language-independent symbolic representation (LISR). This representation aids in conducting a sym-

bolic dependence analysis (SDA) to track dependencies within each unit. Subsequently, the framework employs the static analysis to narrow down the instrumentation scope, minimizing the runtime overhead in the subsequent phase. This phase aims to approximate the dynamic information flow from specified sources to sinks, across different language units. For interpreted languages like Python, the subsequent phase initiates dynamic instrumentation while the statically instrumented program runs with a given input set. This dynamic analysis operates in real time, utilizing a shadow event queue to buffer execution events. The framework continuously conducts runtime information flow computation (IFC) between sources and sinks, updating a dynamic information flow graph (DIFG) with the results. Additionally, leveraging the DIFG, PolyCruise accommodates various security applications for detecting vulnerabilities. These applications function as plugins, persistently scanning for information flow paths associated with particular vulnerabilities, including data leaks. Identified vulnerabilities are then incorporated into the framework's outputs.

4.2.2 Static analyses and instrumentation

To amass the runtime data vital for its Dynamic Information Flow Analysis (DIFA), PolyCruise necessitates the instrumentation of the provided program P. Emphasizing the core principle explicated in the design overview, the instrumentation approach aims to acquire runtime data in a manner that transcends the confines of individual programming languages. This strategy enables a language-agnostic dynamic analysis, thereby reducing the dependence on language-specific analyses. Adhering to this principle, the initial phase of PolyCruise encompasses three principal stages, delineated as follows.

4.2.2.1 Symbolic translation (Step 1.1)

To diminish the reliance on language-specific analyses, PolyCruise aspires to a language-agnostic approach, employing Symbolic Dependence Analysis (SDA). This method systematically generates uniform instructions for instrumentation by initially translating every lan-

guage unit within P into a symbolic representation, which is language-independent called LISR. The uniqueness of LISR lies in its capacity to offer a standardized representation that ensures consistent SDA results across diverse programming languages.

As a particular note, LISR serves as a foundational element, providing a standardized and consistent representation of various programming languages. This uniformity guarantees coherence in SDA results, regardless of the specific programming language in use. Additionally, when contrasted with traditional data dependence analysis, SDA on LISR adopts a more lightweight methodology, focusing solely on capturing symbolic dependencies. This selective strategy discards resource-intensive analyses, such as pointer analysis or reachability solving. Consequently, the SDA-guided instrumentation method meticulously probes pertinent statements, striving to enhance efficiency and minimize computational overhead.

Definition. The symbolic dependencies derived from SDA aim to approximate data dependencies between statements within a language unit. Consequently, the LISR representation acknowledges three primary types of statements: line (representing assignments), call (denoting function calls), and return (indicating return statements). Additionally, it recognizes two distinct categories of symbols: global (referring to global variables) and function (pertaining to functions). Below is the definition of LISR's formal syntax:

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

In the formal syntax provided for LISR, P, denoted as a program, is delineated as a succession G^* for global symbols, succeeded by function symbols, F^*. A straightforward identifier distinctly identifies each symbol. A global symbol is constructed from an expression of e. Conversely, an F encompasses the return type τ, the function name f, a sequence of

parameters x*, and a sequence of statements S*. τ exclusively signals if f yields a symbol, with the return type having no bearing on the symbolic dependence computation within the context of SDA.

Moreover, a statement S can manifest in any of the following three forms: a function call statement [x=]f(e*) (with optional return value and possibly none), a line statement [x=]e*, and a return statement return e. In this context, the expression e can assume the form of a symbol x, a constant C, or ε, where ε signifies an empty string. It's noteworthy that since LISR primarily functions for the approximation of data dependence, it exclusively captures variable definitions and corresponding uses, excluding other conventional language syntax features (e.g., operators) from its representation.

Translation. As per the aforementioned LISR definition, transforming a language unit into its symbolic translation involves a simple process of light syntactic parsing. This parsing entails the translation of each statement within the unit's source code that falls under the specified three categories or encompasses either of the two types of data symbols outlined in the LISR framework. The principal aim is to capture the declarations and usages of variables. Given the limited scope of the syntactic analysis required and the absence of any semantic scrutiny, Converting a language's code into its LISR equivalent is a relatively straightforward task.

LISR distinguishes three main code entities: statement, global variable, and function. Algorithm 3 outlines the translation process for LISR, it translates each entity based on its type. For global entities, global symbols are extracted (line 5). Function definition entities are symbolized as LISR expressions (line 7). Regarding statement translation (line 8-21), for call statements (line 9), the left symbol is the return value (if exists), and the function call is the right symbol. Return statements (line 15) extract the symbol for an appropriate LISR return statement. Other statement types represent left symbols with definitions and right symbols with uses (line 19-21).

Figure 4.3 illustrates the conversion of a language unit (left column) into LISR (right

Algorithm 3: Generate the LISR form of a given code entity

Input: **E**: a given code entity of the target language: one of the three types: *global*,*function*,*statement*

Output: S_{lisr}: the corresponding LISR for **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
```

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

Figure 4.3 An example of the symbolic translation.

column), where the global variable gValue with type typeA (Line 2) is denoted as gValue of global symbol, while the statement at Line 3 is represented as a function symbol Output(arg), followed by a subsequent call statement without a return value. LISR, unlike conventional IRs such as LLVM's IR, focuses solely on capturing essential information (def/use) necessary for subsequent analysis in PolyCruise, enabling cross-language functionality by presenting language-agnostic essential data, despite varying language semantics.

4.2.2.2 Symbolic dependence analysis (SDA) (Step 1.2)

After translating a specific language unit into LISR, the subsequent imperative involves the computation of the unit's symbolic dependencies. This process entails meticulously examining the def/use of symbols stipulated within the LISR, transcending the constraints of any particular programming language.

Definition. In symbolic dependence analysis, two statements, S_i and S_j, are symbolically dependent if the intersection of their use and def sets is not empty: $U(S_i) \cap D(S_j) \cup D(S_i) \cap U(S_j) \neq \varnothing$, where D(S) and U(S) represent the def and use sets for S.

Besides approximating true flow dependencies [61] ($D(S_i) \cap U(S_j) \neq \varnothing$), the analysis

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

Figure 4.4 Symbolic def-use chains of the code in Figure 4.3.

considers potential anti-dependencies [61] ($U(S_i) \cap D(S_j) \neq \varnothing$) to ensure SDA soundness. Lacking pointer/reference analysis, the SDA operates without aliasing information for the analyzed language unit, risking the oversight of true dependencies induced by aliasing effects. In Figure 4.3, if V at Line 9 is S'alias after Line 8, accounting for anti-dependencies would include Lines 13 and 14 to Line 9's symbolic dependence set, potentially capturing dynamic information flow paths. Neglecting this could lead to missing crucial flow paths, as only statements that are symbolically dependent on Line 9 would be examined during runtime.

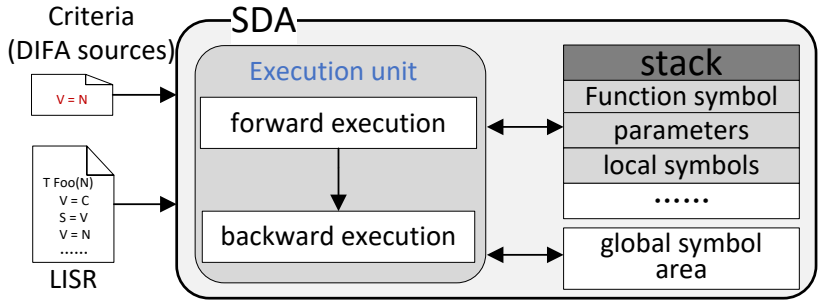

Figure 4.5 An overview design for the analysis of symbolic dependence.

Figure 4.4 contains symbolic definitions and uses corresponding to the example depicted in Figure 4.3. Let S_i represent a statement at Line i, and assume (S_9, V) as the criteria. Con-

sidering true flow dependencies, $D(S_9) \cap U(S_{11}) \neq \varnothing$. Additionally, for anti-dependencies, $U(S_8) \cap D(S_9) \neq \varnothing$. Consequently, the symbolic dependence set of S_9 is S_8, S_{11}.

Dependence computation. Following the provided definition, the computation of symbolic dependencies within a language unit is illustrated in Figure 4.5. This process involves utilizing a set of predetermined criteria and the symbolic representation of the program. The computation starts from a collection of entry functions within the target libraries or executables, maintaining a runtime stack and a global symbol area (GSA) until the termination of the program.

Algorithm 4 delineates the intricacies of how the Symbolic Dependence Analysis (SDA) calculates the overall program's symbolic dependencies. The algorithm is structured into three sequential procedures: computeSD $\rightarrow$ computeSDoF $\rightarrow$ computeMain. The SDA accepts predefined criteria for a specific language unit alongside the corresponding LISR. Then, it performs symbolic analysis, capturing both true/flow and anti-dependencies, in both forward and backward manners. This procedure depends on the management of two separate memory areas: a GSA for overseeing global symbols and a stack that encompasses the local symbol area (LSA) to monitor local symbols.

Within SDA, the initiation involves accepting entry functions from target programs and a pre-established set of criteria as inputs. The methodology characterizes the entry function as the main function within executables or the designated interface functions for external use in libraries. The criteria set is predesigned by users of the framework.

As the first step in the computeSD process, all entry functions are added to a First-In-First-Out (FIFO) queue $\mathbb{Q}_F$ (line 2). Following this, entry function is executed sequentially along with its corresponding Symbolic Dependence Summary (SDS) using the computeSDoF procedure. The SDS of a function is presented as a bitmap, indicating accessibility of its return value or sequential parameters from the criteria by computing Def-Use-Associations (DUA) at a specific call site. For example, with an 8-bit SDS, the SDS of Output at call-site S_{13} is calculated as 01000000 in the given scenario. In this representation, the first bit, 0,

Algorithm 4: Procedure of symbolic dependence analysis

Input: $\mathbb{E}_F$:entry functions, $\mathbb{C}$: criteria (e.g., DIFA sources)

Output: $\mathbb{S}_s$:$\mathbb{C}$'s symbolic dependences

```
1  Function computeSD (E_F, C)
2      Q_F ← initQueue (E_F));                          // Q_F is a FIFO queue of functions
3      G_sym ← ∅;
4      while Q_F ≠ ∅ do
5          F ← Q_F.pop();
6          Stack ← ∅;
7          SDS_F ← getSDS (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
```

signifies an unreachable return value, while the second bit, 1, denotes the first parameter's accessibility. Since Output possesses only one parameter, the left bits are set to zero.

SDS of an entry function can be initialized as -1 by default or with a predefined value defined in the criteria set (line 7). The Stack is reset for each entry, while the GSA remains consistent across all functions. Following the insertion of a global symbol into the GSA upon being reached from a criterion, the entry functions referencing the symbols of global are, again enqueued to $\mathbb{Q}_F$ for reevaluation, ensuring data consistency (line 11). Once $\mathbb{Q}_F$ is empty, the SDA outputs a statement set as the guidance for the instrumentation.

Algorithm 5 outlines the process for computing symbolic dependencies within each function. The algorithm begins by initializing the local symbol area (LSA) based on the function's Symbolic Dependence Summary (SDS), which specifies the formal parameters to be incorporated into the LSA (line 2). Subsequently, the algorithm iterates through the forward and backward (line 4 and line 7, respectively) computation procedures until the count of reachable statements levels off, signifying that all feasible statements have been gathered. The backward computation mirrors the forward process but operates in reverse order.

In the computeMain procedure (Algorithm 6), the SDA conducts symbolic execution for

Algorithm 5: the procedure of SDA on each function

Input: $\mathbb{F}$:an entry function, $SDS_{\mathbb{F}}$:the (current) SDS of $\mathbb{F}$, *Stack*: stack, G_{sym}: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 input function. Initially, the SDA checks whether the current function is already in the stack; if not, it is pushed onto the stack to prevent redundant recursive invocations, as the SDS of the function remains unaltered with the fixed inputs. The algorithm then iterates through and executes each statement in three distinct categories:

Line: If the Use of a statement falls within the set of criteria, either under LSA or GSA, the SDA propels the Def into the corresponding area (either LSA or GSA), thereby facilitating the propagation of symbolic dependence along the execution flow.

Call: The SDA coordinates the examination of the Call statement to discern interprocedural dependencies. Before initiating the callee's execution, the SDA obtains the definition of the callee and constructs an invocation context, including the call site and an initial SDS of the callee derived from the actual parameters. This initial SDS delineates symbols from the parameters that will enter the callee, facilitating the SDA in initializing the callee's LSA. In situations where SDA encounters difficulty retrieving the definition of the callee (e.g., for a library function) and its initial SDS is non-zero, a cautious strategy is implemented. SDA designates the callee's SDS as -1, signifying that the return value and all function parameters are considered reachable from the criteria. If possible, SDA invokes computeSDoF for the callee. Following the execution of the callee, SDA updates the LSA of the current function

Algorithm 6: The main logic of computing symbolic dependence

Input: $\mathbb{E}_F$:entry functions, $\mathbb{C}$:pre-defined criteria

Output: $\mathbb{S}_s$:statement set for instrumentation

```
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
               // get callee's initial SDS:actual->formal args
14             Callee, Sds_c ← initCallee (S_i);
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 𝕃_sym after call
20         else if getType (S_i) == Return then
21             if S_i ∈ {𝕃_sym ∪ 𝔾_sym} then
22                 setRetBit (SDS_f) ;                            // return to the callsite
23         if isReachable (S_i) == true then
24             S_s.push(S_i) ;                                    // save the stmts for instrumentation
25     Stack.pop(𝔽);
26     SDS_f ← SDS_f | summarize (𝕃_sym, 𝔽) ;                     // update function's SDS
27     return SDS_f;
```

with the callee's SDS, which indicates the symbols that flow from the callee back to the current function after return.

Return: The SDA conducts a thorough analysis of the Use associated with a Return statement. If the Use within the LSA or GSA suggests that the return value of the current function is accessible based on the criteria, the SDA adjusts the return bit of the SDS to 1 accordingly. Throughout the execution of each statement, the SDA consolidates reachable statements into a set. Upon completing the execution of all statements, the SDA updates the SDS of the function with the LSA before exit—an essential step, particularly when the function involves output parameters.

4.2.2.3 SDA-guided static instrumentation (Step 1.3)

During the final stage, PolyCruise conducts static instrumentation on every compiled-language unit. This involves the integration of probes to collect runtime data crucial for its dynamic information flow analysis (DIFA). The static instrumenter precisely aims at the statement set determined by the SDA as symbolically dependent on a specific source within the unit. Following this determination, probes are introduced exclusively at those statements to capture the pertinent runtime data. The data, gathered during the execution of the program, will be further elaborated upon in the subsequent phase. It's worth noting that if program P lacks any compiled-language unit, this particular step can be overlooked without disrupting the overarching process.

4.2.3 Online dynamic analysis

PolyCruise executes program P' while performing online analysis on it, regardless of whether it has been statically or dynamically instrumented. This continuous process remains active until the program P' concludes. The online methodology is indispensable for managing extended or continuously running programs, especially those serving as persistent services. Storing the complete execution trace externally for subsequent offline analysis, as indicated

by prior research [21, 20], may prove impractical in these particular scenarios.

4.2.3.1 SDA-guided dynamic instrumentation (Step 2.1)

In the case of units that are programmed with interpreted languages, particularly with intricate dynamic features, solely relying on static instrumentation might not be adequate for capturing essential runtime data. The complex nature of these units often renders their code inaccessible to static analysis. To address this, PolyCruise performs dynamic instrumentation on these units, enabling the comprehensive collection of runtime data. As elucidated in Step 1.3, the focus is specifically on probing statements that exhibit symbolic dependencies on the provided sources. Furthermore, both instrumentation phases target the same category of runtime facts, as expounded upon below.

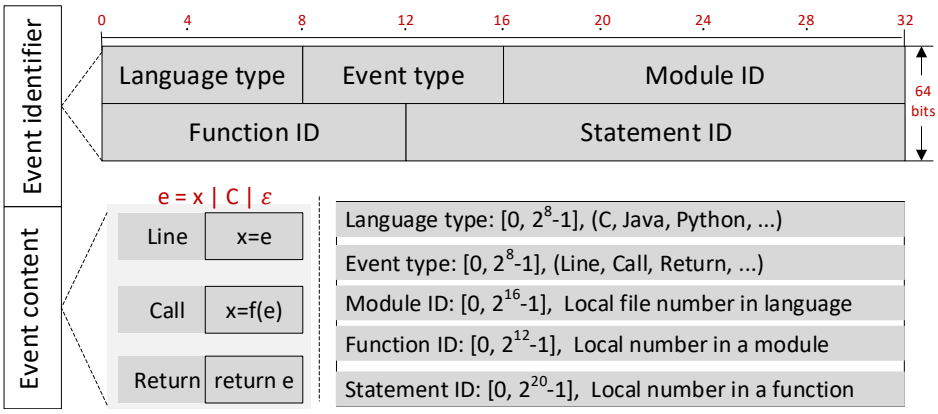

Figure 4.6 Format of execution events at runtime.

A Language-Independent Execution Event (LIEE) commences with a 64-bit event identifier, succeeded by the event content, as shown in Figure 4.6. Within the event identifier, the event type serves as a primary field that guides the analyzer in decoding the event content. While other fields do not actively contribute to dynamic analysis, they establish a direct link from an execution event to a static statement. This linkage empowers the analyzer to trace the original path leading to the identified vulnerability. PolyCruise incorporates three distinct event types, namely Line, Call, and Return, each aligning with different types of

program statements (S). The formal definition of these event types is outlined below.

$$\mathbf{S} \quad ::= \text{x} = \text{e} \mid \text{x} = \text{f(e)} \mid \text{return e}$$
$$\text{e} \quad ::= \text{x} \mid \text{C} \mid \varepsilon$$

An expression $\vec{e}$ can be a symbol x to represent a basic datatype (e.g., an integer), a constant C to represent a pointer value (address), or ε. The event contains a statement translated from the original statement following the syntax above. A crucial point here is that the instrumenter should instrument the address of a non-basic type rather than its symbol. The address is used to compute aliases and real data dependencies at runtime.

4.2.3.2 Information flow computation (Step 2.2)

Definition. Consider the execution P' composed of a collection of threads $\Phi = \varphi_1, ..., \varphi_n$. These threads have the potential to share code and/or data through global or shared variables. The DIFG representing this execution is denoted as $\mathbb{G}\Phi = \langle G\varphi^*, s^*, t^* \rangle$, where $\varphi \in \Phi$. This structure comprises thread graphs G_φ, each commencing from an entry point s and concluding at an exit point t. A thread graph G_φ is constructed from a series of function graphs G_f connected by interprocedural edges, involving either call or return events. Each G_f is a collection of nodes that directly correspond on a one-to-one basis with the event sequence $\vec{e}$. Here, $\vec{e}_k$ signifies the k-th event in the sequence $\vec{e}$.

Each node within the DIFG symbolizes an individual execution event. The edges within this graph, categorized into one of the four types listed below, serve to represent flow facts:

(1) $\vec{e}_{itcf}$: inter-thread control flow edge. When examining two events $\vec{e}_i \in \varphi_m$ and $\vec{e}_j \in \varphi_n$, where $\vec{e}_i$ represents a call-site for thread creation, $\vec{e}j$ denotes the first node of φ_j, and φ_i serves as the predecessor of φ_j, the notation $\vec{e}itcf = \vec{e}_i \rightarrow \vec{e}_j$ indicates an intra-thread control flow from $\vec{e}_i$ to $\vec{e}_j$. This connection signifies the control flow between the thread creation event in φ_m and the initiation of φ_n.

(2) $\vec{e}_{cf}$: intra-thread control flow edge, including intra- and inter-procedural (function) edge. In the context where two events $\vec{e}_i$ and $\vec{e}_j$ belong to the same thread φ: If $\vec{e}_i$ is a

call event and $\vec{e}_j$ is the entry event of the corresponding callee, then $\vec{e}_i \rightarrow \vec{e}_j$ represents an inter-procedural edge, specifically a call edge. If $\vec{e}_i \rightarrow \vec{e}_j$, it forms an intra-procedural control flow edge, provided that $\vec{e}_i$ and $\vec{e}_j$ occur sequentially within the same function. When $\vec{e}_i$ is a return event and $\vec{e}_j$ is the call event corresponding to the call-site, the connection $\vec{e}_i \rightarrow \vec{e}_j$ is characterized as an inter-procedural edge, specifically functioning as a return edge.

(3) $\vec{e}_{itdd}$: inter-thread data flow edge. When considering two consecutive events, $\vec{e}i$ in φi and $\vec{e}j$ in φj in chronological order, the relation $\vec{e}i \rightarrow \vec{e}j$ is established as an inter-thread data dependence edge under the condition that the use operation $\mathrm{U}(\mathrm{e}_j)$ within φj is part of the set of dependencies $\mathrm{D}(\mathrm{e}_i)$ within φi. This dependency relationship is contingent on $\mathrm{U}(\mathrm{e}_j)$ referring to a shared or global variable, with $\mathrm{D}(\mathrm{e}_i)$ denoting the latest write operation on that specific variable.

(4) $\vec{e}_{dd}$: intra-thread data flow edge, including intra- and inter-procedure (function) edge. When considering two events $\vec{e}_i$ and $\vec{e}_j$ within the same thread φ, the edge $\vec{e}_i \rightarrow \vec{e}_j$ is an intra-procedural data flow edge if $\mathrm{U}(\vec{e}_j) \in \mathrm{D}(\vec{e}_i)$ and $\vec{e}_i$ and $\vec{e}_j$ occur sequentially within the same function. If $\vec{e}_i$ is a return event and $\vec{e}_j$ is the call event corresponding to the call-site, the relationship $\vec{e}_i \rightarrow \vec{e}_j$ represents an inter-procedural edge, functioning as a return edge. In the case of a call event $\vec{e}_i$ with $\vec{e}_j$ in the corresponding callee, the edge $\vec{e}_i \rightarrow \vec{e}_j$ is an inter-procedural edge, specifically identified as a call edge.

Algorithm 7 illustrates the primary steps involved in the incremental construction of the Dynamic Information Flow Graph (DIFG). When an event $\mathbb{E}$ is received, the algorithm starts by decoding the event content and retrieving the thread graph G_φ. Subsequently, it assigns a new node CurNode to the event and retrieves its predecessor in G_φ. If G_f for $\mathbb{E}$ does not exist, indicating that $\mathbb{E}$ represents the current function's first event, the algorithm creates a G_f. In cases where G_f is NULL, implying that $\mathbb{E}$ serves as the current thread's first event, an inter-thread control flow edge is established. When G_f exists, the algorithm initially adds intra-procedural control and data flow edges if applicable. If $\mathbb{E}$ represents a call event, the callee's DIFG, G_{callee}, is obtained, and inter-procedural control and data flow

Algorithm 7: Incremental construction of DIFG

Input: $\mathbb{E}$: an execution event, φ: the $Thread$ number

Output: $\mathbb{G}_\Phi$: the updated DIFG

```
1  Function constructDIFG (E_F, C)
2      ēvent ← decodeEvent (E);
3      G_φ ← getOrAddGraph (ē_d, φ);
4      G_f ← getFuncDIFG (G_φ, ēvent);
5      CurNode ← newNode (G_φ, ēvent);
6      PreNode ← getPreNode (G_φ);
7      if G_f == NULL then
8          G_f ← addFuncDIFG (G_φ, ēvent);
9          if PreNode == NULL then
10             createItcfgEdge (G_Φ, G_φ) ;                    // inter-thread CF edge
11     else
12         TailNode_f ← getTail (G_f);
13         createCfgEdge (TailNode_f, CurNode);
           // compute intraprocedural data dependence
14         while TailNode_f != NULL do
15             if U[CurNode] ∈ D[TailNode] then
16                 createDDEdge (TailNode_f, CurNode);
17                 break;
18             TailNode_f ← TailNode_f -> previous
19         if IsCallEvent(ēvent) then
20             G_callee ← getFuncDIFG (G_φ, ēvent);
21             createCfgEdge (G_f, G_callee) ;                 // inter-procedure CF
22             createDDEdge (G_f, G_callee) ;                  // inter-procedure DD
           // compute DD for global or shared variables
23         computeGlobalDD (G_Φ, CurNode);
```

edges are subsequently added. It is important to note that a call event can only be captured after executing the corresponding callee during dynamic analysis. Lastly, if existing shared or global variables are referred to U[$\mathbb{E}$], the algorithm computes global data dependence. This procedure involves a form of global search within the execution context.

4.2.3.3 Security applications (plugins) (Step 2.3)

PolyCruise facilitates a range of security applications, addressing the general source-sink problem. To showcase the practical utility of this technique, the emphasis is placed on vulnerability detection leveraging the Directed Interprocedural Flow Analysis (DIFA) within PolyCruise. Various detector plugins compute the information flow paths between these sources and sinks through a reachability analysis of the most recently updated Dynamic Information Flow Graph (DIFG), addressing different information flow security vulnerabilities. As the DIFG is incrementally constructed and updated, any identified vulnerabilities are reported as security bugs, contributing to the comprehensive security measures integrated within PolyCruise (refer to Figure 4.2).

4.3 Implementation

PolyCruise has been developed and applied to Python-C programs along with seven plugins, each designed to detect a specific type of vulnerability on top of dynamic information flow.

4.3.1 Static analysis

LISR translation. The implementation of C-LSI was realized on the LLVM compiler framework [76]. The translation process of C-LSI involves traversing the LLVM intermediate representation (IR) generated by the compiler frontend (Clang) and converting it into a Language-Independent Symbolic Representation (LISR). For Python-LSI, I reused pypredictor [161] (for Python 2.7) to transform Python code into the form of static single assignment (SSA), generating the corresponding abstract syntax tree (AST). The tool was subsequently

updated to support Python 3.6 and higher versions, and LISR is translated from the source in SSA form.

Instrumentation. Utilizing the results from the Symbolic Data Analysis (SDA), an LLVM pass [76] was implemented to statically instrument the C sources. The execution event follows the specification defined in Figure 4.6. Regarding the Python units, sys.settrace (built-in API) was used for dynamic instrumentation with the guidance of SDA results. Then, statement details were captured during runtime, and the events were constructed accordingly; by combining the earlier generated ASTs and dynamic stack information,

The tracing module was implemented as a C library, facilitating direct integration with both language components.

Symbolic dependence analysis. The SDA was implemented as a simplified symbolic execution engine using C++. This choice was made since the Language-Independent Symbolic Representation (LISR) of language components does not include branches except for function invocation. The primary objective of SDA is to calculate all statements that can be reached from the predefined criteria in a language component.

To guarantee the smooth flow of data among different language components and enhance the scalability of the framework, a two-level implementation of the SDS was undertaken for the entry functions of diverse language components. At the initial level, a conservative computation strategy is employed, assuming that all parameters of entry functions are reachable from the predefined criteria. Specifically, the SDA calculates the invocation relationships among functions within each language component. It identifies functions that are invoked by no other functions, designates them as entry functions, and initializes their SDSs as -1. While this approach may lead to redundant pins, it ensures coverage of all potential data flow paths across languages.

The framework provides users with the flexibility to customize the second level of the SDS through configuration. This empowers users to define SDS for each language interface according to their specific requirements. Consequently, the framework accommodates only

the data flow paths that users find relevant or interesting. With these customized SDSs and user-defined criteria as a guide, the SDA can accurately and efficiently compute all feasible statements within the language components, ensuring alignment with the user's specific needs and preferences.

4.3.2 Runtime libraries

The runtime library consists of two functionalities: tracing library and dynamic event queue. It is implemented as a C library to link it to mainstream language components (e.g., Python, Java, Ruby).

Tracing library Each event is tagged with the corresponding thread number in the tracing API to facilitate analysis for multi-threaded programs. Additionally, to minimize any potential side effects on the target program, the tracing API promptly returns after pushing an execution event into the event queue.

Dynamic event queue: The queue is implemented as a bidirectional circular queue initialized in shared memory. Throughout runtime, the target program utilizes the tracing API to push execution events to the queue, while the analysis engine polls events for dynamic information flow analysis. Due to its thread- and process-safe design, the queue implementation is suitable for deployment in real-world programs.

4.3.3 Dynamic information flow analysis engine

The dynamic information flow analysis engine (DIFAE) was implemented using the C programming language. Structured around functional components, DIFAE comprises five key units: event parser, memory database, online DIFA, and security application basis.

Event parser. Adhering to the event definition outlined in Figure 4.6, the event parser decodes the events in the string to extract def-use information and event types of each event.

Memory database. To ensure the processing capacity for analyzing large-scale programs, we developed an efficient memory database leveraging the memory pool and hashing. In con-

trast to traditional databases, we adopted the following strategies to ensure high efficiency:
1. Light-weight implementation: We focused solely on implementing essential functionalities such as hash, memory pool, and data operations (e.g., add, query, and delete).
2. High memory efficiency: With a data size of 64 bytes on average, the database can accommodate over ten million data units with just 1 Gigabyte of memory.
3. Scalable memory pool: Managing memory units based on a scalable memory pool helps prevent frequent memory allocations and releases.

For the performance evaluation of the database, we conducted tests on the creation and query operations under an Ubuntu 18.04 desktop environment with a 2.30G CPU and 16GB DDR4. The results demonstrated that the database can create 10 million records in 1.7 seconds and execute 3.3 million queries per second when a table contains 10 million records.

Online DIFA. Online DIFA is performed incrementally based on the memory database, where each dynamic event activates the updating process of the dynamic information flow graph (DIFG). During the implementation of Algorithm 7, two primary concerns demanded attention: the thread graph and the language boundary.
1. Thread graph: To handle the chaotic events occurring during the analysis of multi-threaded or multiprocess programs, DIFG is structured with a set of thread-level graphs. This approach enables the sequential analysis of events within a thread, allowing DIFAE to compute intra- and inter-procedural data dependence for each event within a specific thread graph. Interthread data flow is facilitated by maintaining a shared global cache among all the thread graphs. for the write operation of the memory, DIFAE updates the cache to track the latest writing location for the memory address. Similarly, for the read operation of the memory, the engine retrieves the address's latest writing location and then establishes a data flow edge from the write to the read point. While this search may slow down its efficiency, we adopt a "local first" strategy to minimize excessive searching. Upon discovering a local dependence, the engine terminates the search in the global scope.
2. Language boundary: SDA ensures comprehensive instrumentation covering all potential

cross-language data flow paths. Nevertheless, challenges arise due to differences in data encapsulation and conversions between languages, posing obstacles for dynamic data dependence computation at runtime. For instance, the process of converting parameters from Python to C through C extension involves transitions from Python type to Pyobject to C type [133], while parameters flowing from Java to C through Java Native Invocation follow the sequence from Java type to jobject to C type [114]. Rather than modeling all interface APIs at language boundaries to guide parameter correlation across different language components, I opt for a field-insensitive approach for parameter passing in foreign function invocations, as illustrated in Figure 4.7.

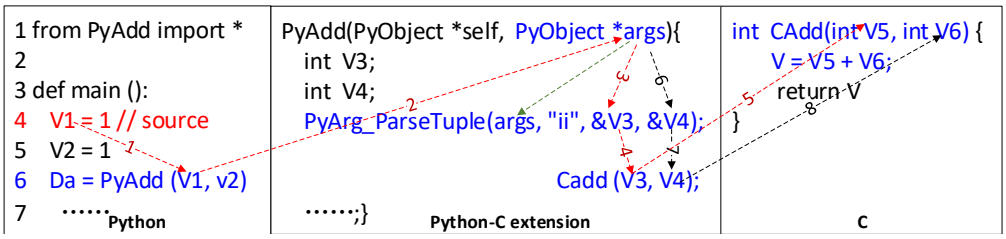

Figure 4.7 An example of field-insensitive parameter passing.

In the given example, line 4 in Python is set as the criterion. Variable V1 follows edge 1 to the line 6 call site and then proceeds along edge 2 into a PyObject args that encodes V1 and V2. Due to the unknown method for decoding args, which is language-dependent, a cautious approach is adopted. Two data flow edges 3, 6 are added from the definition point of args to line 4 of the Python-C extension: edge 3 represents the flow of V1 into V3, while edge 6 indicates the flow of V1 into V4 (simulated data flow). While this conservative computation may introduce side effects, it ensures the soundness and language independence of the framework, making it adaptable to support other languages. Conversely, once a vulnerability is reported on a "simulated" data flow path across language components, it may be both authentic and effective, potentially originating from an undefined source.

Security application basis: DIFAE offers a collection of interfaces designed to develop security applications (plug-ins) in the C language. More precisely, a framework user can efficiently implement a specific vulnerability detection plug-in using just 100 lines of C code.

The integration of the plug-in into the framework is streamlined, requiring the user to specify the location, entry point, and source/sink of the plug-in for seamless incorporation. This approach ensures a quick and straightforward process for users to enhance and extend the functionality of the framework with their custom vulnerability detection modules.

4.3.4 Limitations

While PolyCruise is primarily designed for precise dynamic analysis of multi-language programs, its current implementation can lead to the generation of imprecise flow paths due to its handling of parameters at language boundaries in a field-insensitive manner. Moreover, although PolyCruise has been implemented and thoroughly tested on Python 3.7, claiming to support most language features of Python 3.x, variations in Python versions might introduce changes in PolyCruise's behaviors.

It's important to note that, PolyCruise does not handle data flow across multiple processes. In the current implementation, I treat a process similarly to a thread, constructing an independent analysis context and dynamic information flow analysis for each. However, I have not yet handled interprocess communication mechanisms, leading PolyCruise to fail to capture data flow across processes due to the independent address spaces.

4.4 Evaluation

PolyCruise is evaluated through four fundamental aspects: (1): PolyCruise'effective in terms of its precision. (2): PolyCruise's efficiency in terms of its costs. (3): PolyCruise's capacity for real-world vulnerability discovery.

4.4.1 Experiment setup

PyCBench: a Python-C micro bench. Due to the unavailability of benchmarks with ground truth written in Python-C for cross-language analysis, a manual micro-benchmark set was devised, named PyCBench, to ensure the framework's accuracy. The PyCBench test suite, as

shown in Table 4.1 (GenF: general flow, GF: global flow, FieldSen: field sensitivity, ObjSen: ObjectSensitivity, DynInv: dynamic invocation), was developed to evaluate both static and dynamic cross-language analyses (Python and C). The suite includes forty-six hand-crafted micro applications categorized into five groups, covering seven typical security vulnerability types. PyCBench has been made available alongside PolyCruise and will continue to receive updates and improvements in subsequent research.

Table 4.1 PycBench's features and outline.

Vulnerability	**GenF**	**GF**	**FieldSen**	**ObjSen**	**DynInv**	**Sum**
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
Sum	15	9	8	10	4	46

In addition to PyCBench outlined in Table 4.1, a collection of twelve other popular multilingual open-source projects primarily developed in Python and C was obtained from GitHub, as shown in Table 4.2. The benchmarks were curated with specific constraints to ensure their practicality and effectiveness. These included criteria such as the projects having around 1,000 or more stars, indicating their substantial and renowned presence within the open-source community, and a predominant development in Python and C (or C++), with both languages contributing over 30% of the project size. Moreover, the projects were required to have multiple developers involved, ensuring ongoing maintenance and updates, and a rich set of test cases to guarantee high-quality and thorough testing. Detailed documentation was also necessary for easy installation, usage, or testing. Further information about each project can be found on the respective open-source project websites.

To conduct the integration tests of the benchmarks using PolyCruise, the Pytest [132]

Table 4.2 Real-world multilingual benchmarks with main languages of Python-C.

Benchmark	**Size (KLoC)**	**C/C++%**	**Python%**	**#Tests**
Bounter [17]	3.5	48.2%	50.9%	190
Immutables [66]	5.9	55.0%	44.3%	152
Simplejson [67]	6.4	37.6%	59.8%	31
Japronto [125]	9.4	50.4%	48.2%	15
Pygit [93]	17.0	57.4%	44.6%	241
Psycopg [126]	27.5	50.8%	48.2%	198
Cvxopt [36]	56.0	60.8%	39.0%	78
Pygame [129]	207.0	54.3%	44.7%	324
PyTables [131]	219.8	52.1%	46.6%	6355
Pyo [4]	259.1	50.8%	48.8%	51
NumPy [112]	919.7	36.1%	63.7%	16002
PyTorch [138]	6419.2	56.2%	35.2%	4146

and Unittest [137] frameworks were customized to automate the test case execution and collect the associated analysis results. In certain benchmarks, only a small number of paths were identified by PolyCruise based on the default source/sink configuration. To enhance the runtime coverage and increase the likelihood of discovering vulnerabilities, a tool was devised to extract all Python function interfaces as sources.

Experimental methodology. For the real-world benchmarks, ground truth was not available; therefore, manual validation was conducted by randomly sampling ten resulting information flow paths between unique source/sink pairs. Each path was validated using its statement-level details by referencing the respective subject's source code. The path was considered a true positive if the information flow from the source to the sink was exercised without any sanitization in between; otherwise, it was considered a false positive. Precision was computed based on this sampling. To compute recall, ground truth was manually constructed for the three least complex subjects against the execution for which PolyCruise found the least paths.

To evaluate the effectiveness of PolyCruise, the tracing of each program statement is

collected on both the PyCBench and five real-world benchmarks. The precision and recall were computed for each benchmark and test using the traced paths as ground truth. PolyCruise's efficiency and scalability are evaluated separately on its dynamic and static and analysis parts. Specifically, SDA's efficiency, in terms of both time and memory usage, was measured. For Phase 2, the slowdown factor of run-time was compared among the original, SDA-instrumented, and completely instrumented versions of each benchmark. All experiments were conducted on Ubuntu 18.04 with an Intel i7-10875H CPU and 16GB RAM.

4.4.2 Effectiveness of PolyCruise

Effectiveness on PyCBench. The analysis results of PolyCruise on PyCBench are summarized in Table 4.3 (INT-LP: inter-language path, ITR-LP: Intra-language path). In the evaluation, PolyCruise successfully identified almost all information flow paths in PyCBench, with only three false positive alarms. Two of these false positives were in the group labeled as "Field sensitivity", while the other was in the "Object sensitivity" group. These false alarms were a result of PolyCruise's field-insensitive approach to language boundaries, which is implemented to ensure language independence. As a consequence, two benign flows across languages were misdiagnosed. However, these false positives can be optimized and eliminated in the implementation. The precision of PolyCruise was 93.5%, while the recall was 100%. Though the micro-benchmarks may not entirely capture the intricacies of various scenarios of real-world applications, the design of PyCBench took into account common program analysis problems, ensuring the correctness of the implementation.

Effectiveness on practical benchmarks. The effectiveness results of PolyCruise on the five real-world projects are displayed in Table 4.4 (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.). Through manual validation of a total of 486 tests, 15 paths were identified as ground truth. PolyCruise generated 17 paths, of which 15 were validated as true positives, resulting in an overall

Table 4.3 Effectiveness on PyCBench.

Group	**#INT-LP**	**#ITR-LP**	**#Alarm**	**#False alarm**
General flow	10	4	14	0
Global flow	9	0	9	0
Filed sensitivity	8	0	8	2
Object sensitivity	9	2	11	1
Dynamic invocation	4	0	4	0
Summary	40	6	46	3

precision of 88.2% and recall of 100%. For the two false positives observed in Cvxopt, The primary reason was the field-insensitive handling of the language interface. Furthermore, it was confirmed that, among the 15 vulnerability-associated paths, 10 were exploitable, resulting in a precision of 58.8% under the security context. Although these findings may not be generalizable, they provide confidence in the effectiveness of PolyCruise when applied to real-world multilingual systems.

Table 4.4 PolyCruise's effectiveness on real-world benchmarks.

Benchmark	$\mathbf{P}_g$	$\mathbf{P}_p$	**#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%

4.4.3 Efficiency of PolyCruise

The efficiency of PolyCruise was evaluated on twelve real-world programs, amounting to a total of 8142.5 thousand lines of code (KLOC). This assessment involved measuring various aspects in both static and dynamic analysis. In the realm of static analysis, the focus was on

three properties: (1) the time consumption of the Static Data Analysis (SDA), (2) the peak memory utilization of the SDA, and (3) the rate of instrumentation. Additionally, during dynamic executions, the assessment included two metrics: (1) the runtime slowdown factor, and (2) the peak memory in Dynamic Information Flow Analysis (DIFA).

Slowdown factor. The computation of the slowdown factor typically involves dividing the execution time with the SDA version (E_{SDA}) by the execution time of the pure version (E_{pure}). However, during the evaluation process, a consistent initialization phase overhead in the Python interpreter for each execution. For instance, in the test runs, the unittest or pytest frameworks consistently took several seconds to prepare for the case execution, regardless of the program under test. A straightforward constant slowdown factor connects the execution times T_{SDA} and T_{pure}. However, for a more accurate representation, it is advisable to consider the initialization times I_{SDA} and I_{pure}, along with the execution times E_{SDA} and E_{pure}, introducing a slowdown factor S that exclusively affects the execution time.

$$T_{pure} = I_{pure} + E_{pure}$$

$$T_{SDA} = I_{SDA} + S \cdot E_{pure}$$

Consequently, the slowdown factor is computed as follows:

$$S = \frac{T_{\text{SDA}} - I_{\text{SDA}}}{T_{\text{pure}} - I_{\text{pure}}}$$

Efficiency of SDA. Table 4.5 provides the values of three measurements of SDA. In general, although SDA's time and memory usage increase as the size of the targets increases, SDA can complete its analysis in a matter of minutes even for programs with millions of lines (e.g., PyTorch). For smaller programs, SDA can finish in seconds or even milliseconds.

Regarding the instrumentation rate, with the guidance of SDA, the rate varies between 43% (Pygit) and 62% (Pyo). The main reason for these results is the conservative computation at language boundaries, which covers all possible data flow paths across languages. Nevertheless, in real-world tests, only a few paths need to be traced when the source and

Table 4.5 SDA performance, SDA-T:time, SDA-M:memory, and instrumentation rate.

Benchmark	SDA-T (sec)	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%
Pygit	0.13	14.54	43%
Psycopg	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	7414.95	51%

sink are fixed. A straightforward optimization for the framework users is configuring the second-level SDS for boundary interfaces to reduce the instrumentation rate.

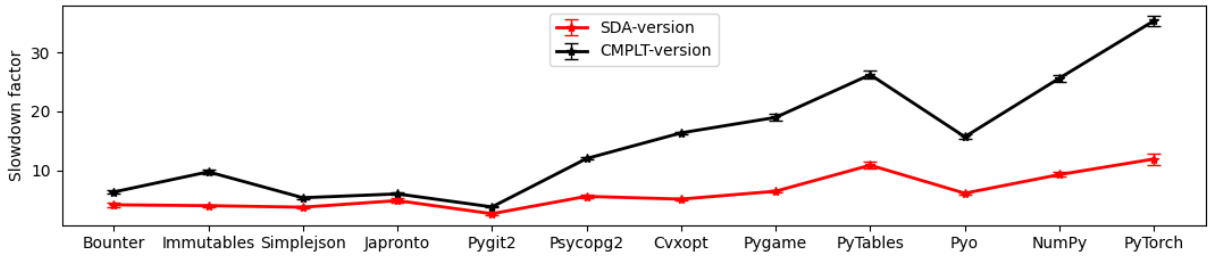

Figure 4.8 Comparison of slowdown factor between SSDA- and CMPL-version.

Runtime slowdown and memory usage. In each benchmark, three versions were compiled: the pure version, the SDA-based instrumentation version (SDA-version), and the complete instrumentation version (CMPL-version). Ten random test cases were selected for each benchmark, and the performance metrics (i.e., memory use, time cost) were collected for these cases across the three versions. Relative to the data of the pure version, the runtime

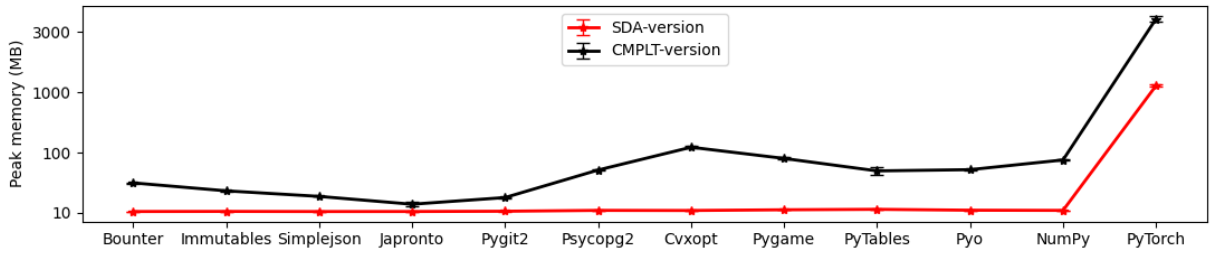

Figure 4.9 Comparison of peak memory usage between SDA- and CMPL-version.

slowdown factor was calculated using the formula (1) for both the SDA-version and CMPL-version in each benchmark. Finally, a set of ten factors was obtained for each benchmark. Figure 4.8 presents the comparison of average slowdown factors between the two versions for each benchmark, along with standard errors.

Compared to the pure version, the SDA-version exhibits runtime slowdown factors ranging from a minimum of 2.71 in Pygit to a maximum of 11.96 in PyTorch. Referring to the instrumentation rate provided in Table 4.5, PolyCruise achieves a lower rate in Pygit (43%) compared to PyTorch (51%), thereby potentially leading to a lower slowdown factor in the former during runtime. Given the computational intensity of PyTorch relative to other benchmarks like Pygit, it generates a significantly larger number of execution events during runtime, averaging 55 million events across ten random PyTorch cases. This notable increase in the number of events contributes to the higher slowdown factor experienced.

However, in contrast to the CMPL-version, the SDA-version demonstrates superior time and memory efficiency across all twelve benchmarks, as illustrated in Figure 4.8 and 4.9. Specifically, the SDA-version enhances efficiency by reducing the slowdown factor by 18.3% (in Japronto) to 66.2% (in PyTorch), and curbing memory usage by 16.2% (in Japronto) to 67.1% (in Cvxopt).

In summary, PolyCruise demonstrates a significant performance improvement compared to the CMPL-version. Despite this advancement, the conservative computation in SDA results in notable slowdown factors during runtime across all practical benchmarks.

Table 4.6 New vulnerability discovery of PolyCruise.

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

4.4.4 Real-world vulnerability discovery

Based on the program executions of the practical benchmarks, as depicted in Table 4.2, PolyCruise successfully detected 14 previously unknown vulnerabilities in the real-world benchmarks. These vulnerabilities are detailed in Table 4.6 (IntOf: Integer-overflow, BufOf: Buffer-overflow, and InCc: Incomplete-comparison).

4.5 Related Work

ORBS [13] provides language-independent support for multilingual program analysis, but scalability issues limit its practical application, even in its improved version [77]. Semantic summarization, as demonstrated by Dillig et al. [40], suffers from significant information loss and complex language semantics [80, 160]. Unified intermediate representations such as pyLang (PyLLVM) and JLang enable multilingual analysis, yet the complexity of various language features and the need for extensive engineering work pose substantial challenges [97, 169, 7]. Dynamic techniques like Truffle [74] and DroidScope [164] encounter practical limitations related to laborious runtime component implementation, potential discrepancies in program behavior, and overall efficiency challenges. Various studies, including those by Tan et al. [150], Li et al. [83], Afonso et al. [3], Brucker et al. [19], Bae et al. [10], Lee et al. [79], Brown et al. [18], Dinh et al. [41], and Li et al. [82], have targeted specific language

combinations, although extending support for additional language pairs remains challenging.

In contrast, PolyCruise diverges from existing methodologies, as PolyCruise employs a language-independent SDA approach that mitigates runtime slowdown with minimal reliance on language-specific analysis, indicating potential adaptability for new language combinations and providing the first cross-language DIFA implementation for Python-C programs.

CHAPTER FIVE

HOLISTIC GREYBOX FUZZING OF MULTI-LANGUAGE SYSTEMS

The prevalence of multiple programming languages in software systems introduces security threats alongside various benefits. However, state-of-the-art techniques are insufficient for such systems. PolyCruise's effectiveness is hindered by the input coverage; existing fuzzing techniques predominantly target single-language software, leaving multi-language systems inadequately protected. To address this, I introduce PolyFuzz, a comprehensive greybox fuzzer that efficiently tests multi-language systems by utilizing cross-language coverage feedback and a sophisticated model capturing semantic relationships across languages. The evaluation against single-language fuzzers on 15 multi-language and 15 single-language benchmarks demonstrated that PolyFuzz obtained 25.3–52.3% higher code coverage and uncovered twelve previously unknown multi-language vulnerabilities, along with two vulnerabilities in single-language benchmarks, got five CVEs assigned. These results underscore the significance of holistic approaches like PolyFuzz in securing multi-language software systems.

5.1 Motivation

Greybox fuzzing has exemplified significant efficacy in exposing vulnerabilities in real-world software [100]. Notably, prominent fuzzers like AFL (American Fuzzy Lop) [99] and libFuzzer [95] have successfully identified over 16,000 vulnerabilities across various projects [98]. However, our understanding suggests that these leading fuzzers primarily focus on single-language programs [100], while the majority of contemporary projects (more than 80%) are developed using multiple programming languages [147].

The application of single-language fuzzers to multilingual code poses several challenges, notably as follows:

(1) Feasibility for different languages: The complexity and diversity of interfaces between various languages in multi-language software lead to variations in runtime input formats

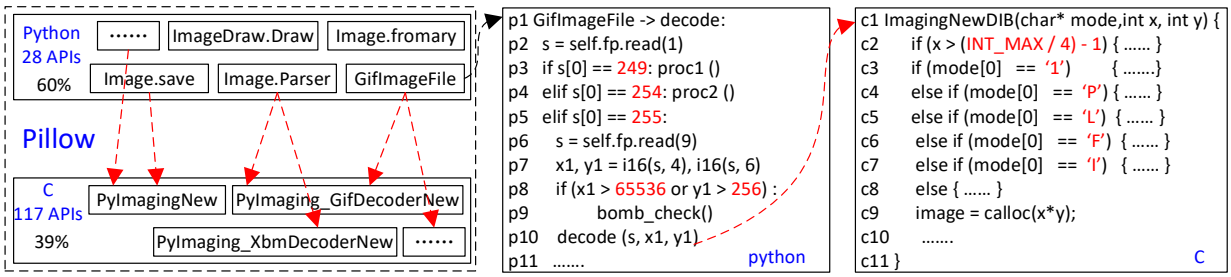

Figure 5.1 A real-world multi-language software: Pillow.

across language units and APIs. It becomes impractical to construct suitable invocation contexts and fuzzing instances for all the APIs.

(2) Inefficiency due to incomplete feedback: In the context of fuzzing a multilingual program, single-language fuzzers commonly face challenges in advancing the fuzzing process. This difficulty arises from the absence of comprehensive coverage feedback, resulting in inefficient performance—particularly when interacting with black-box components within the system.

(3) Reproducibility of vulnerabilities: Engaging in fuzzing at the level of individual language units disrupts the semantic connections between these units, introducing potential discrepancies in vulnerability detection. The inherent limitation lies in the lack of a comprehensive understanding of the complete system execution path, potentially resulting in false positives during vulnerability assessment. Acknowledging this challenge underscores the importance of adopting approaches that consider the whole context for vulnerability detection.

These drawbacks of single-language fuzzers necessitate the development of a cross-language fuzzing technique that achieves holistic, whole-system fuzzing (WSF). However, WSF encounters its challenges, notably the inefficiency resulting from the shortage of initial seeds for exercising cross-language behaviors, and the high percentage of redundant inputs generated due to random mutation guided by control flow coverage [173]. These challenges require a more precise and informed approach to mutation, involving cross-language information flow analysis. Building upon these insights, the development proposal for a cross-language fuzzing framework, named *tech*, aims to facilitate efficient and comprehensive fuzzing of multi-language systems.

5.2 Approach

To enhance the effectiveness of fuzzing in multi-language software environments, I pioneered the development of PolyFuzz. This innovative tool facilitates comprehensive code coverage analysis across the entire system, providing valuable insights for methodically optimizing seed scheduling.

A crucial element of PolyFuzz is its initial phase, the seed generation process, specifically crafted to tackle the prevalent issue of limited initial seeds in multi-language systems. In this phase, PolyFuzz utilizes sensitivity analysis to explicitly map semantic relationships among various input segments and branch predicates, informed by regression analysis. Following this, it smoothly transitions to conventional fuzzing, dynamically adapting and reverting to seed generation as required.

To seamlessly handle diverse language combinations, PolyFuzz integrates a streamlined language-specific analysis for holistic coverage measurement. This analysis selectively captures only the essential variable values for constructing the regression model. Facilitated by a custom intermediate representation (IR), PolyFuzz standardizes runtime value probing across different languages, making the majority of PolyFuzz language-agnostic.

5.2.1 Overview of PolyFuzz

In Figure 5.2, the comprehensive design of PolyFuzz is depicted. This cutting-edge tool operates on three crucial Inputs:

(1) The multilingual program P, encompassing a collection of language units for testing purposes. (2) A set of adaptable fuzzing drivers for P, catering to the requirements of various fuzzing interfaces. (3) The initial tests for P' for the fuzzing process.

The functioning of PolyFuzz is orchestrated through three core functional units:

Unit 1: Static analysis and instrumentation, involving a meticulous procedure of static language-specific analysis (LSA) on P. This process entails translating the sources into an

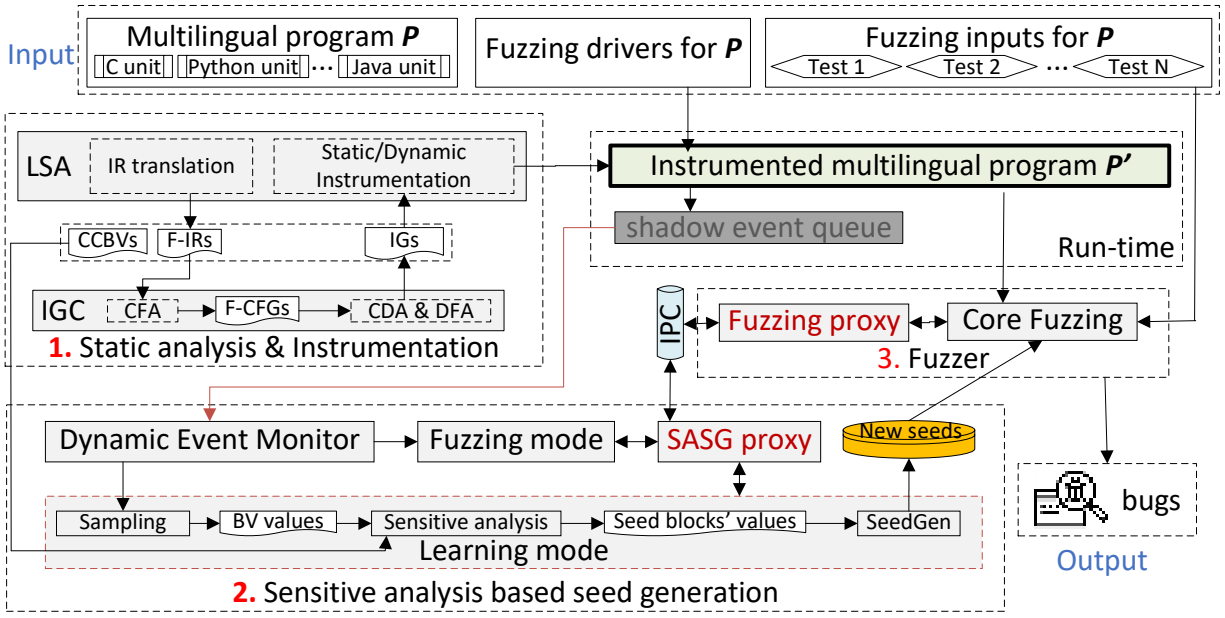

Figure 5.2 An overview of PolyFuzz's architecture.

intermediate representation for each function (F-IR). Simultaneously, it parses the constant constraints of branch variables (CCBV) to facilitate effective seed learning in Unit 2. The subsequent step involves the computation of instrumentation guidance, merging the results of control dependence analysis (CDA) and data flow analysis (DFA) to generate the final guidance for both static and dynamic instrumentation.

Unit 2: Sensitivity Analysis-based Seed Generation (SASG) operates in two distinct modes, namely Fuzzing mode and Learning mode. In the Fuzzing mode, it directs the fuzzer (Unit 3) to initiate fuzzing on the instrumented multilingual program P'. Simultaneously, it monitors dynamic events, capturing the coverage of branch variables (CoBV). Upon detecting changes in CoBV triggered by a seed S, it switches to Learning mode. This mode entails conducting random sampling (mutation) on S by seed block, performing sensitive analysis between seed blocks and branch variables, and leveraging the collected constants of branch variables from Unit 1 to predict the values of seed blocks. The generated new seeds are then employed to enhance the fuzzing process before reverting to Fuzzing mode.

Unit 3: The language-agnostic fuzzer functions in tandem with SASG's learning procedure, prioritizing and loading new seeds into the seed queue. Operating on path-coverage guidance,

the Fuzzer collaborates with SASG to improve coverage and effectively detect vulnerabilities. In the event of triggering vulnerabilities, the Fuzzer reports the associated seed as PolyFuzz Outputs for further analysis and reproduction.

5.2.2 Static analysis and instrumentation

To streamline and expand the language-specific analysis within PolyFuzz, the static analysis and instrumentation (SAI) phase is strategically devised. This phase is dedicated to reducing complexity and simplifying the overall language-specific analysis, thereby enhancing the language extensibility of the system.

5.2.2.1 IR translation

To fulfill the specific demands of PolyFuzz, two distinct types of instrumentation are imperative: (1) Instrumentation at the basic block level to facilitate path-coverage computation. (2) Instrumentation at the definitions of branch variables to enable sensitive analysis. Accommodating these requirements necessitates the implementation of both intra-procedure control flow and data flow analyses. However, as these intensive program analyses must be applied to each language unit within multilingual systems, they pose a considerable obstacle to the seamless extension of the algorithm to new languages. This complexity also contributes to the challenging maintenance of PolyFuzz.

In response to this challenge, we have devised an intermediate representation (IR) that retains the essential and simplified syntax required for the unified program analysis of multilingual programs. This IR, known as SAIR within the context of the paper, serves as a unifying foundation that streamlines the analysis process and facilitates a more cohesive approach to handling multilingual codebases.

SAIR definition. Based on the outlined instrumentation requirements, the SAIR is defined with an emphasis on two fundamental program elements: basic blocks and the definitions of branch variables. Consequently, we establish the SAIR with a specific global symbol, namely

the function. Additionally, two distinct types of statements, LINE (denoting assignments) and CMP (for comparisons), along with two value types, integer (I) and other (O), are incorporated into the SAIR's structure. Below is the formal syntax of the SAIR:

$$
\begin{aligned}
P &::= F^* \\
F &::= \tau f(x^*)S^* \\
S &::= [x =]e^* \mid [cmp]e^*, e^* \\
e &::= \tau x \mid C \mid \varepsilon \\
\tau &::= I \mid O
\end{aligned}
$$

In the representation, a program P is expressed as a sequence F^* consisting of function definitions. Each function F is characterized by a return type τ, a function name f, a sequence x^* of parameters, and a set of statements S^*. The return type τ of f is simplified into two fundamental types: I (integer) and O (other). Two kinds of statements exist in this context: a line statement [x=]e^* covering non-compare statements like assignments, calls, and unified returns with assignments, and a cmp statement [cmp]e^*,e^* specifying the comparison of two variables. Expressions e can take one of three forms: a variable x with type τ, a constant C, or ε indicating an empty string. Additionally, a return tag τ is categorized as either a general type T or ε.

For control flow analysis, SAIR retains the basic block information from the sources. As only branch variables with integer types are considered in data flow analysis, SAIR simplifies the variable types to just two categories, namely integer and other. Furthermore, to streamline the analysis process, SAIR condenses call and return statements into unified assignments, as the focus primarily lies on intra-procedure analysis.

Translation. Given the SAIR definition mentioned earlier, the translation of a language unit occurs via a lightweight syntactic parsing process on a function-by-function basis.

Algorithm 8 provides a comprehensive overview of the SAIR translation process. The SAIR translator initially converts the declaration to SAIR format for a function definition (line 2). Subsequently, it iterates through all the basic blocks (lines 4 to 15). Within each

basic block, the SAIR translator sequentially parses each statement (lines 6 to 15).

In the case of a CMP statement (line 9), the translator decodes the uses and generates a corresponding CMP statement in SAIR. If this CMP statement involves a parameter that includes an integer constant, the translator captures the branch variable information, which includes a unique identifier, the prediction type, and the value of the constant (line 11). As for other statements, they are translated into LINE statements following the SAIR format.

Algorithm 8: Translate a given function to SAIR

```
   Input: 𝔽: a given code function
   Output: F_sair: the SAIR of 𝔽
1  Function translate2SAIR (𝔽)
2      S_sair ← getFDeclaration (𝔽);                    //translate function declaration
3      F_sair.append (S_sair);
4      foreach B_i in 𝔽 do
5          B_sair ← initBlock (F_sair, B_i) ;           //initialize current basic-block
6          foreach S_i in B_i do
7              if isCMP (S_i) then
8                  _, Uses = getDefUse (S_i);
9                  S_sair ← getCmpSAIR (Uses[0], Uses[1]);
10                 if hasIntConstant (S_i) then
11                     dumpBrVariable (S_i);            //dump branch variables with int const
12             else
13                 Def, Use = getDefUse (S_i);
14                 S_sair ← getLineSAIR (Def, Use);
15         B_sair.append (S_sair);                      //insert S_sair to current basic-block
16     return F_sair
```

5.2.2.2 Instrumentation guidance computation

With SAIR as input, Instrumentation Guidance Computation (ICG) takes three language-independent program analysis phases: (1) Intra-procedure control flow analysis (CFA), (2) Intra-procedure control dependence analysis (CDA), (3) Intra-procedure data flow analysis (DFA). Then ICG decides which blocks should be instrumented according to the results of CDA and DFA.

Algorithm 9: Procedure of ICG on SAIR

Input: $\mathbb{F}$: a given function of SAIR

Output: S_{icg}: a set of statements for instrumentation

```
1  Function icgComputation (𝔽)
2      CFG ← getFCfg (𝔽);                                          //get function CFG
3      DomBB ← calDomofBB (CFG);                                   //compute dominance on CFG
4      PDomBB ← calPostDomofBB (CFG);                              //compute post-dominance on CFG
5      S_BB ← ϕ;
6      foreach B_i in 𝔽 do
7          if isEntry (B_i, 𝔽) then
8              S_BB.append (B_i) ;                                 //always instrument the entry block
9          else
10             if isFullDominator (B_i, DomBB) || isFullPostDominator (B_i, PDomBB) then
11                 continue;
12             S_BB.append (B_i) ;                                 //instrument the non-(post)dominators
13     S_BV ← calReachability (CFG);                               //cal instrument-sites for branch-vars
14     S_icg ← merge (S_BB, S_BV);
15     return S_icg
```

The detailed procedure for calculating the minimal instrumentation sites for a given SAIR function $\mathbb{F}$ is outlined in Algorithm 9. The algorithm commences by constructing a control flow graph (CFG) (line 2). With CFG as the input, it proceeds to compute the dominant and post-dominant relationships between the basic blocks within CFG through the process of Control Dependence Analysis (CDA) (lines 3 to 4). During the iteration of each basic block (lines 6 to 12), the decision to include a block in the block set S_{BB} is contingent upon its impact on the control-flow-path distinction. Consequently, the entry block of CFG is marked for instrumentation and added to S_{BB}.

Under specific conditions, a block is identified as a non-instrumented block if it dominantly influences all of its immediate descendants, exhibiting behavior akin to a full post-dominator. This categorization is based on its limited impact on the path distinction within the CFG. Concurrently, the algorithm employs Data Flow Analysis (DFA) [59] to derive a set of definitions for all branch variables (S_{BV}). Ultimately, the algorithm yields S_{icg} as the output, carefully eliminating redundant elements from the consolidated sets S_{BB} and S_{BV}.

Algorithm 10: Seed partition and sampling

Input: $\mathbb{P}$: instrumented program

Input: $\mathbb{S}$: a seed triggering new CoBVs

Input: $\mathbb{L}$: list of preset values of block-length, e,g, {1, 2, 4, 8}

Input: $\mathbb{N}$: the number of sampling

Output: SBP_{list}: lists of SBP

```
1  Function seedPtSampling (ℙ, 𝕊, 𝕃, ℕ)
2      SBP_list ← ϕ;
3      foreach L_i in 𝕃 do
4          Pos ← 0;
5          while Pos + L_i <length (𝕊) do
6              S_Bi ← 𝕊 [Pos : Pos + L_i];          //extract a block with length L_i
7              N_s ← 0;
8              while N_s <ℕ do
9                  S' ← randMutate (𝕊, S_Bi);      //mutate S_Bi and generate new seed S'
10                 execute (ℙ, S');                //execute ℙ with new seed S'
11                 BV_list ← collectBrValues ();
12                 updateSbBvPairs (SBP_list, S_Bi, BV_list);
13                 N_s ← N_s + 1;
14             Pos ← Pos + L_i;
15     return SBP_list
```

5.2.3 Sensitive-analysis-based seed generation

Sensitive-Analysis-Based Seed Generation (SASG) is responsible for monitoring the coverage of branch variables (CoBV) during the fuzzing process. When a seed triggers new CoBV, SASG initiates seed learning, involving three primary phases: (1) seed partition and sampling, (2) sensitive analysis, and (3) seed generation, as elaborated below.

5.2.3.1 Seed partition and sampling

Instead of treating the seed as a mere byte stream [48], the seed partition process divides it into blocks of equal length L (e.g., 1-byte block, 2-byte block, 4-byte block), effectively treating the seed as a block stream. This approach is more practical, considering that an integer branch variable is often influenced by a block of bytes with a specific length, rather than just a single byte. During the sampling process, the seed blocks are randomly mutated, and the instrumented program's execution observes the values of branch variables. Consequently, a list of value pairs of seed-blocks and branch-variables (SBP) is generated for further sensitive analysis.

Algorithm 10 presents a detailed stepwise procedure for seed partition and the generation of Seed Block Profiles (SBPs). The algorithm takes four distinct inputs: the instrumented program ($\mathbb{P}$), a seed ($\mathbb{S}$), a predefined list of partition lengths ($\mathbb{L}$), and the sampling quantity $\mathbb{N}$. For each partition length L_i (line 3), the seed undergoes $\mathbb{N}$ random mutations, block by block (lines 5 to 14). In the sampling process (lines 9 to 12), a new seed $S^{'}$ is generated by randomly mutating block S_{Bi}. After executing $\mathbb{P}$ with seed $S^{'}$, all reached branch variables are collected into BV_{list}, and the SBPs list, SBP_{list}, is updated with information from S_{Bi} and BV_{list} during the current execution.

5.2.3.2 Sensitive analysis

The sensitive analysis phase involves constructing regression models based on the SBP lists to discern the functional associations between branch variables and seed blocks. By utilizing

Algorithm 11: Procedure of sensitive analysis

Input: PC: preset parameter combinations of regression models

Input: SBP_{list}: a list of SBP

Input: BV_{set}: a set of integer constants of branch variables

Output: S_{sb}: a set of seed block values

```
1  Function sensitiveAnalysis (SBP_list, BV_set)
2      RM_list ← {Rbf, Polynomial, Linear};          //initialize regression model list
3      Acc_opt ← 0;                                  //initialize the accuracy as 0
4      Rm_opt ← φ;
5      Train, Test ← split (SBP_list);
6      foreach RM[i] in RM_list do
7          Rm_i, Acc_i ← getModel (PC, RM[i], Train, Test);
8          if Acc_i > Acc_opt then
9              Acc_opt ← Acc_i;
10             Rm_opt ← Rm;                          //selected the optimal RM
11     S_sb ← predict (Rm_opt, BV_set)
12     return S_sb
13 Function getModel (PC, RM[i], Train, Test)
14     Acc_i ← 0;
15     Rm_i ← φ;
16     foreach pc in PC[i] do
17         rm ← trainModel (RM[i], pc, Train);       //train a model for each pc
18         res ← predict (rm, Test);
19         acc ← calAccuracy (res, Test);
20         if acc > Acc_i then
21             Acc_i ← acc;
22             Rm_i ← rm
23     return Rm_i, Acc_i
```

the acquired models, it becomes possible to predict the values of seed blocks using the integer constants extracted during the static analysis (Unit 1). This approach facilitates the generation of new seeds based on the predicted seed blocks, increasing the likelihood of either hitting or reversing the comparison conditions at branches.

Algorithm 11 provides a comprehensive overview of the sensitive analysis process, encompassing model training, selection, and prediction. The algorithm is designed to accept preset parameter combinations of regression models, an SBP list, and a set of integer constants of branch variables as input. The initialization of three regression types as RM_{list} (line 2) marks the outset of the process. For each regression type, multiple models are trained with the preset parameter combinations on the training data, with the selection of the model that exhibits the highest accuracy on the test dataset for the specific regression type (lines 13 to 23). Subsequently, an optimal model Rm_{opt} is identified based on its accuracy among the three regression models (line 10). Finally, the algorithm leverages Rm_{opt} in conjunction with the constants of branch variables to predict the potential values of the seed block.

5.2.3.3 Seed generation

During the phase of seed generation, the process involves the creation of new seeds by assembling the values of the seed blocks sequentially. Illustrated in Figure 5.3, the predicted values of seed blocks (sb_0, sb_1, ..., sb_n) are utilized to formulate a seed SD as a sequence of seed blocks, represented as SD = {sb_0[k] sb_1[m] ... sb_n[o]}, where k, m, and o are random variables. The seed space SS is further defined as SS = x $\times$ y $\times$ ... $\times$ z, with x, y, and z representing the number of values in blocks sb_0, sb_1, and sb_n, respectively.

To address the efficient production of seeds, we approach the seed generation problem by transforming it into a path construction problem within a directed graph (DG). Initially, we establish a dummy entry node S and an exit node E for DG (as depicted in Figure 5.3). Subsequently, the depth of DG corresponds to the number of seed blocks, denoted as n, with the graph nodes at depth i representing the values in seed block sb_i. Notably, a seed is

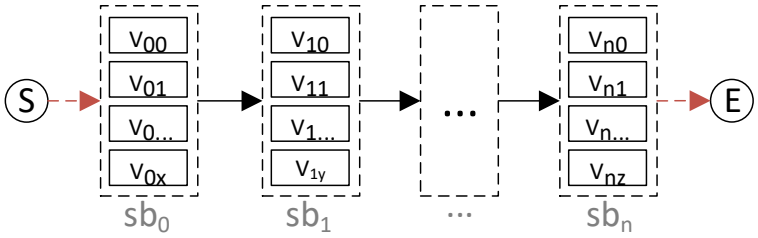

Figure 5.3 The block representation of a seed.

equivalent to a path from S to E, excluding the dummy nodes.

Nevertheless, practical scenarios often witness an escalation in seed length and the complexity of the fuzzed program, which may potentially trigger an explosion issue within the path (seed) space. The former can lead to a heightened graph depth, while the latter may introduce more branch variables, further amplifying the predicted values (graph nodes) in seed blocks. As a preventive measure against such explosion issues and to ensure the generation of a maximum number of effective paths, the seed generation phase is subdivided into two sub-steps: (1) Weighted sampling and (2) Path construction.

Weighted sampling. Due to the diversity of the coverage of branch variable (CoBV) in different seed blocks, sampling should also be biased to the blocks with good CoBV. Given a limited path space M as budget and the number of predicted seed-block n, for a seed block sb_i, the sampling number SN_i on sb_i is calculated as follows:

$$
\begin{aligned}
&(1) \quad SN_{avg} &&= power(M, \tfrac{1}{N'}) \\
&(2) \quad W_i &&= N_{bvi} \ / \ \textstyle\sum_{j=1}^{n} N_{bvj} \\
&(3) \quad SN_i = &&\begin{cases} SN_{avg} + (N_{bvi} - SN_{avg}) \times W_i & N_{bvi} \neq 0 \\ 1 & N_{bvi} = 0 \end{cases}
\end{aligned}
$$

In formula (1), the power function is used to compute the average sampling number SN_{avg} for all seed blocks, considering them to be of equal importance. Subsequently, the weight of block sb_i is determined based on the number of covered branch variables, as outlined in formula (2). In formula (3), the calculation of SN_i is contingent upon adjusting the average (SN_{avg}) sampling size either upwards or downwards, depending on the weight when $\mathrm{N}_{bvi} \neq 0$.

If N_{bvi}=0, as previously discussed, the block is assigned the original seed value, effectively setting SN_i to 1 for that particular block.

Algorithm 12: Path construction in a depth-first traversal

```
   Input: SBL: a seed block list
   Input: SNL: the list of weighted sampling numbers for SBL
   Input: D: the max depth of the graph
   Input: p: current path in construction
   Input: d: current depth of the path p
   Output: PL: a list of paths
1  Function getPathByDF (SBL, SNL, D, p, d)
2      SN_d ← SNL[d];                          //get weighted sampling number for block d
3      SB_V ← randomSampling (SBL[d], SN_d);   //sampling SN_d seed block values
4      foreach v in SB_V do
5          p[d] ← v;                           //insert v to the position d of p
6          if d == D then
7              insert (PL, p);                 ;//a full path generated
8          else
9              getPathByDF (SBL, SNL, D, p, d + 1);   //recursively process d+1
10     return
```

Path construction. Algorithm 12 outlines the path construction procedure in a depth-first traversal. At the current depth d, the algorithm initiates by obtaining the pre-calculated weighted sampling number (SN_d) and then proceeds to randomly sample SN_d seed block values (SB_V) (see line 3). Following this, the algorithm iterates through all the values (nodes) present in SB_V (lines 4-9). More specifically, for each value v within SB_V, it is inserted into the current path p at slot d. In case the iteration reaches the exit node (with a maximum depth of D), a new path is generated (see line 7). If not, the algorithm recursively executes the path generation procedure for the next depth d+1 (line 9).

5.2.4 Fuzzer

The Fuzzer is designed with two primary units: Fuzzing proxy and Core fuzzing. The Fuzzing proxy plays a crucial role in facilitating communication with collaborative programs (such

as SASG in this context) and controlling the operational status of Core fuzzing. The communication is structured around three fundamental message types: start_up, switch_mode, and new_seed.

The start_up message is utilized for the initial handshaking process, ensuring the functional status validation of both communication parties. During the seed sampling process in learning mode, SASG uses the switch_mode message to notify the fuzzer posed. This precautionary measure is implemented to prevent potential conflicts arising from shared memory read/write operations that occur when both the fuzzer and SASG execute simultaneously.

Upon receiving the new_seed message, the Fuzzing proxy loads the new seeds from the database and alerts Core Fuzzing to initiate the fuzzing process with the newly loaded seeds. Core fuzzing operates by the traditional fuzzing process, encompassing seed selection, mutation, and bug reporting. Notably, in this design, all block information from various language units is mapped onto the same shared memory byte-map. This approach allows for the calculation of block- (or path-) coverage without differentiating between the language types of the tested programs, thus fulfilling the language-agnostic requirement.

5.3 Implementation

The PolyFuzz implementation supports programs written in one or more of three widely used languages: Python, Java, and C. The key components of PolyFuzz are illustrated in Figure 5.4, featuring a shared C component responsible for static analysis and instrumentation. The essential component comprises three fundamental libraries: the SAIR parser, the instrumentation guidance computation (IGC), and DynTrace. The language-specific analysis layer has access to these libraries through the wrapper situated in the language interfacing layer. For its fuzzing core, PolyFuzz integrates AFL++ [46]. The overall implementation spans approximately 12KLoC, with 0.6 KLoC dedicated to Java, 1.2 KLoC to Python, and 0.3 KLoC to C.

Language-specific analysis and instrumentor. For each language, a SAIR translator

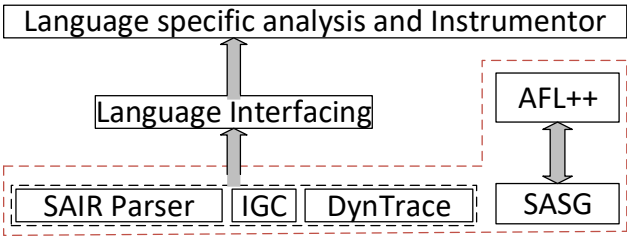

Figure 5.4 An overview of PolyFuzz's implementation.

and instrumentor have been implemented. In C, an LLVM pass is used for translation, along with IGC for guidance computation, and dynamic tracing APIs from the DynTrace library. The Java implementation, built on Soot, incorporates JNI wrappers for DynTrace and IGC. In Python, the SAIR translator and instrumentor are developed separately, employing a static parser based on AST and a dynamic instrumentor using Pybind. The process of adding support for new languages remains straightforward, leveraging the existing C libraries for complex algorithmic functionalities.

IGC. In the IGC module, the fusion of intraprocedural control dependence and data flow analysis is rooted in the program's SAIR. The resulting output articulates each instrumentation guidance as a tailored value pair <block-id, statement-id> specific to individual functions. Notably, the IGC implementation prioritizes thread safety, facilitating concurrent operation with compilers or program analysis frameworks such as LLVM [76] and Soot [75].

Dynamic tracing (DynTrace). The C-based library implementation serves a triple role. Initially, it provides an API to initialize the shared-memory byte map crucial for AFL++'s coverage computation. Subsequently, it offers an API for setting up the shadow event queue, responsible for caching covered branch variables during SASG. Lastly, it encompasses APIs for tracing dynamic events, capturing block information, and logging branch variables. These accessible APIs can be directly called by language instrumentors or accessed through tailored wrappers for specific language interfaces, such as the JNI wrapper for Java, ensuring seamless integration into the fuzzing targets.

SASG and AFL++. To enhance the efficiency of the fuzzing process, SASG and AFL++ run concurrently. Furthermore, SASG employs a staggered seed generation approach, prompt-

ing AFL++ to load and initiate fuzzing every 8K seeds generated. In the adaptive model selection phase, regression accuracy is assessed by calculating the mean distance between predicted and ground-truth seed-block values during model validation. A regression is considered unsuccessful if the accuracy falls below 80

Limitations. For a clean runtime environment during each fuzzing session, PolyFuzz operates in non-persistent mode [46], creating a new process for each execution. While this strategy ensures user convenience and system stability, it can slightly impede the fuzzing efficiency due to the overhead of process forking. Moreover, the current implementation assumes the fuzzing target single process, potentially limiting the detection of cross-process bugs and making it challenging to fuzz multilingual code involving multiple processes, such as cases where a C unit invokes other language units through separate processes.

5.4 Evaluation

PolyFuzz is evaluated through four primary aspects: effectiveness on real-world multilingual programs, effectiveness on single-language programs, the impact of sensitivity analysis in PolyFuzz, and real-world vulnerability discovery.

5.4.1 Experiment setup

Baselines. For evaluation, the comparison included PolyFuzz and three prominent single-language fuzzers from the Google OSS-Fuzz framework [52]: Honggfuzz for C, Jazzer for Java, and Atheris for Python. The evaluation was extended to modified versions of Jazzer and Atheris incorporating C code coverage measurement, labeled Jazzer-C-ext and Atheris-C-ext. Additionally, the sensitive analysis phase was disabled in a version of PolyFuzz (PolyFuzz-NSA), which was then compared with AFL++ specifically on C benchmarks.

Benchmarks and initial input seeds. The evaluation encompassed testing PolyFuzz on fifteen real-world multilingual systems (ten Python-C and five Java-C programs) as shown

Table 5.1 Fifteen real-world multi-language benchmarks.

Benchmark	**Size**	**Languages**	**#BV**	**#BV-IntConst**
Libsmbios [38]	8.3	Python:30.4% C:64.2%	6866	3269 (47.6%)
Tink [51]	257.7	Python:7.2% C++:33.5%	66282	27962 (42.2%)
Pillow [124]	75.8	Python:60.0% C:38.6%	15628	9090 (58.2%)
Ultrajson [153]	5.1	Python:34.3% C:64.8%	1361	903 (66.1%)
Aubio [9]	42.9	Python:25.4% C:73.3%	3445	2232 (64.8%)
Bottleneck [128]	16.9	Python:49.5% C:48.6%	3384	1814 (53.6%)
Pycurl [127]	14.6	Python:54.8% C:40.7%	433	264 (61.1%)
Simplejson [67]	6.2	Python:61.4% C:38.6%	858	544 (63.4%)
Msgpack [109]	15.1	Python:48.7% C:50.1%	2322	1056 (45.5%)
Pycryptodome [81]	65.5	Python:43.5% C:56.1%	2842	1595 (56.1%)
Jep [110]	18.9	Java:25.4% C:56.6%	2856	1454 (50.9%)
Jansi [47]	5.2	Java:66.6% C:22.7%	386	121 (31.3%)
Jna [71]	129.4	Java:76.9% C:16.1%	3017	941 (31.2%)
Onenio [113]	29.1	Java:86.0% C:14.0%	4371	1132 (25.9%)
Zstdjni [96]	47.9	Java:6.8% C:88.7%	47803	20384 (42.6%)

in Table 5.1 ((Size in KLOC, BV: branch variable, BV-IntConst: branch variable with constant integer constraints)), chosen based on code size, language distribution, and branch variable characteristics. Additionally, 5 benchmarks were randomly selected from Google's OSSFuzz for C, Python, and Java, respectively, as shown in Table 5.2, to assess PolyFuzz's performance on single-language projects. Custom drivers were crafted for PolyFuzz to suit all fifteen multilingual systems. For Atheris (applied to the ten Python-C programs) and Jazzer (employed for the five Java-C programs), the drivers were adjusted to accommodate variances in the test-input interface. In the case of single-language projects, OSSFuzz drivers were utilized for all single-language fuzzers, and fresh drivers were crafted exclusively for PolyFuzz. To ensure equitable comparisons, all fuzzers underwent execution on the same

benchmark with identical initial inputs.

Table 5.2 Fifteen single-language benchmarks selected from OSSFuzz.

Benchmark	**Size**	**Languages**	**#BV**	**#BV-IntConst**
Bleach [108]	14.4	Python	1035	119 (11.5%)
Sqlalchemy [148]	391.9	Python	30637	2187 (7.1%)
Urllib [154]	18.5	Python	1948	121 (6.2%)
Pyyaml [163]	24.3	Python	2196	107 (4.9%)
Pygments [130]	96.6	Python	4993	381 (7.6%)
Jsonsanitizer [116]	2.3	Java	326	237 (72.7%)
Commonscompress [6]	73.7	Java	8563	5771 (67.4%)
Zxing [174]	47.1	Java	4453	3059 (68.7%)
Jsoup [70]	25.3	Java	2109	1101 (52.2%)
Javaparser [69]	183.9	Java	7743	4683 (60.5%)
Efsprogs [152]	118.4	C	19439	13279 (68.3%)
Bind [68]	275.4	C	56428	33538 (58.8%)
Civetweb [30]	521.7	C	6615	4080 (61.7%)
Cyclonedds [43]	225.9	C	22551	14286 (63.3%)
Igraph [65]	212.1	C	63013	35043 (55.6%)

Performance metrics. In the experimental setup, two primary fuzzing metrics were under consideration: the count of covered basic blocks and the number of triggered bugs. The three baseline single-language fuzzers all relied on basic block coverage as the main performance indicator. Given that, PolyFuzz employed AFL++ [46] as its core fuzzing agent, the reported metric included the number of paths identified by AFL++'s algorithm for reference. When comparing PolyFuzz and PolyFuzz-NSA, the third metric was the number of paths found. The coverage results were averaged over five repetitions of a 24-hour runtime. Additionally, the number of detected bugs played a pivotal role. Recognizing potential inaccuracies in

the count of unique crashes reported by different fuzzers, all reported issues underwent meticulous manual validation. For this validation, a proof of Concept (PoC) was created to replicate each issue using the crash-triggering inputs. Bugs were considered new only if their call stack differed from all other confirmed bugs.

5.4.2 Effectiveness on multilingual programs

Table 5.3 Performance comparison on the Python-C benchmarks.

Benchmark	PolyFuzz				Atheris		Atheris-C-ext	
	#Block	#PythonBlk	#Path	#Bug	#PythonBlk	#Bug	#Block	#Bug
Libsmbios	198	51	35	1	24	0	149	0
Tink	2139	97	755	0	33	0	1891	0
Pillow	1363	1034	233	1	706	1	915	1
Ultrajson	377	126	151	1	39	0	238	0
Aubio	453	187	91	1	126	0	160	0
Bottleneck	1359	25	634	7	14	0	886	2
Pycurl	239	38	19	0	26	0	205	0
Simplejson	374	97	86	0	82	0	197	0
Msgpack	245	48	78	0	43	0	223	0
Pycryptodome	572	243	64	0	185	0	493	0
Total	7319	1946	2147	11	1278	1	5357	3
Improve	—				52.3% ↑	10 ↑	36.7% ↑	8 ↑

In Table 5.3, the comparative outcomes between PolyFuzz and Atheris/Atheris-C-ext for the 10 Python-C benchmarks are presented. The table details the total number of basic blocks covered (#Block) and the count of basic blocks covered in the Python unit (#PythonBlk) by PolyFuzz. For Atheris, coverage measurement is limited to Python code (#PythonBlk), whereas Atheris-C-ext incorporates both Python and C code coverage in the overall (#Block). Analogously, Table 5.4 illustrates the performance contrast between

Table 5.4 Performance comparison on the Java-C benchmarks.

Benchmark	PolyFuzz				Jazzer		Jazzer-C-ext	
	#Block	#JavaBlk	#Path	#Bug	#JavaBlk	#Bug	#Block	#Bug
Jep	418	145	59	0	90	0	354	0
Jansi	332	309	244	1	242	0	261	0
Jna	711	476	189	0	362	0	579	0
Onenio	364	316	131	0	289	0	312	0
Zstdjni	151	84	21	0	47	0	71	0
Total	1976	1330	644	1	1030	0	1577	0
Improve	—				29.1% ↑	1 ↑	25.3% ↑	1 ↑

PolyFuzz, Jazzer, and Jazzer-C-ext across the 5 Java-C benchmarks, replacing #PythonBlk with #JavaBlk to signify the basic blocks covered in the Java unit.

Coverage. In Table 5.3, PolyFuzz exhibits 36.7% and 52.3% more basic block coverage than Atheris-C-ext and Atheris in the overall and Python units, respectively. Similarly, as indicated in Table 5.4, the respective coverage improvements over Jazzer amount to 25.3% and 29.1%. These results highlight the significant enhancement in code coverage by PolyFuzz across the entire system and specific language units. Analysis of the multilingual benchmarks reveals that the C code ranges from 38.6% to 73.3% in the Python-C programs and 14.0% to 88.7% in the Java-C programs (Table 5.1). While Atheris and Jazzer fail to respond to coverage changes within C units, enabling native fuzzing substantially improves the coverage evolution with complete coverage guidance. However, despite these advancements, PolyFuzz still outperforms the extended baselines due to its superior seed generation capabilities. By leveraging whole-system coverage and learned regression models, PolyFuzz identifies favored seeds and explores untouched branches and program paths, achieving higher coverage compared to both Atheris-C-ext and Jazzer-C-ext.

Bug triggering. The bug-triggering outcomes, as outlined in Table 5.3 and Table 5.4,

show that Atheris identified a single bug in Pillow, and Atheris-C-ext revealed two bugs in Bottleneck, with neither Jazzer nor Jazzer-C-ext detecting any bugs. In contrast, PolyFuzz successfully triggered 12 bugs across six projects, including 11 in Python-C and one in Java-C programs. These results underwent manual verification, and proof-of-concept (PoC) reproductions were generated for each bug. The comprehensive system coverage awareness of PolyFuzz not only propels the evolution of the fuzzing process for achieving high code coverage but also significantly improves bug detection in real-world multilingual projects.

5.4.3 Effectiveness on single-language programs

In this evaluation, Extensive comparisons between PolyFuzz and Atheris, Jazzer, and Honggfuzz are performed across 15 real-world single-language benchmarks, emphasizing the two performance metrics.

Table 5.5 Performance evaluation on the Python benchmarks.

Benchmark	PolyFuzz			Atheris	
	#Block	#Path	#Bug	#Block	#Bug
Pyyaml	853	703	1	826	1
Bleach	1023	353	0	796	0
Sqlalchemy	1096	18	0	1047	0
Pygments	1276	229	0	799	0
Urllib	534	71	0	496	0
Total	4782	1474	1	3964	1
Improve	—			20.1% ↑	0 –

Coverage. Tables 5.5-5.7 indicate that PolyFuzz achieved 20.1%, 11.0%, and 10.1% more coverage of basic blocks compared to Atheris, Jazzer, and Honggfuzz on these single-language benchmarks. In contrast to the multi-language benchmarks, all the fuzzers in this case utilized complete system coverage as feedback. Nevertheless, PolyFuzz outperformed all three baseline fuzzers. Table 5.2 highlights that these benchmarks had a significant proportion of

Table 5.6 Performance evaluation on the Java benchmarks.

Benchmark	PolyFuzz			Jazzer	
	#Block	#Path	#Bug	#Block	#Bug
Zxing	4604	1923	0	4575	0
Jsoup	3408	1082	0	3261	0
Javaparser	4729	377	1	3821	1
Commonscompress	339	453	0	296	0
Jsonsanitizer	595	309	0	547	0
Total	13675	4144	1	12319	1
Improve	—			11.0% ↑	0 –

branch variables with constant integer constraints, ranging from 4.9% in Pyyaml to 72.7% in Jsonsanitizer (5th column). Utilizing its SASG module, PolyFuzz efficiently generated seeds from these constant branch constraints. These generated seeds, when used as inputs, enabled PolyFuzz to achieve improved block coverage with fewer random mutations. Overall, PolyFuzz demonstrated an impressive ability to discover more favorable seeds by effectively mutating seeds generated based on sensitivity analysis.

Table 5.7 Performance evaluation on the C benchmarks.

Benchmark	PolyFuzz			Honggfuzz	
	#Block	#Path	#Bug	#Block	#Bug
Efsprogs	1173	302	0	1049	0
Bind	4154	2982	0	3955	0
Civetweb	232	157	0	195	0
Cyclonedds	1091	592	0	1003	0
Igraph	431	454	0	228	0
Total	7081	4457	0	6430	0
Improve	—			10.1% ↑	0 –

Bug triggering. Two issues are unveiled in PolyFuzz during testing: a Recursion error emerged in Pyyaml, and a JVM hang occurred in Javaparser, with no bugs detected in the C benchmarks. The Pyyaml bug was also identified by Atheris using a different seed input, confirmed through the creation of a Proof of Concept (PoC) to reproduce the bug. Similarly, the bug discovered by Jazzer was verified to be identical to the one triggered by PolyFuzz. Although PolyFuzz did not exhibit a clear advantage in bug triggering compared to these single-language fuzzers, its advanced code coverage capabilities suggest the potential for uncovering additional bugs.

5.4.4 Importance of sensitivity analysis in PolyFuzz

In this evaluation, both PolyFuzz and PolyFuzz-NSA support cross-language fuzzing, employing distinct strategies. The comparative performance analysis between the two fuzzers for 15 real-world multilingual programs was based on two vital metrics. The detailed comparison results are presented in Table 5.8. Additionally, the effectiveness of sensitivity analysis was assessed by comparing PolyFuzz with AFL++ across five C benchmarks. The results of this evaluation are documented in Table 5.9.

Coverage. In the realm of basic block and path coverage, PolyFuzz showcases a substantial edge over PolyFuzz-NSA. Take Pillow as an example, where PolyFuzz outperformed by covering 320 (30.7% ↑) more basic blocks and 86 (58.5% ↑) more paths compared to PolyFuzz-NSA. Overall, PolyFuzz recorded a 17.4% surge in basic block coverage and an impressive 21.8% rise in path coverage. It's noteworthy that PolyFuzz-NSA, with its holistic coverage awareness, surpassed single-language fuzzers by covering 30.5% more Python blocks than Atheris and 19.5% more Java blocks than Jazzer. Similarly, juxtaposed with AFL++ on the C benchmarks, PolyFuzz achieved a 7.6% boost in basic block coverage and an 11.4% uptick in path coverage under the same feedback mechanism. This comparison accentuates the broad efficacy of SASG in PolyFuzz, particularly extending its influence beyond multi-language fuzzing and suggesting potential benefits for enhancing the performance of existing

Table 5.8 Performance evaluation between PolyFuzz and PolyFuzz-NSA.

Benchmark	PolyFuzz			PolyFuzz-NSA		
	#Block	#Path	#Bug	#Block	#Path	#Bug
Libsmbios	198	35	1	174	30	1
Tink	2139	755	0	1771	627	0
Pillow	1363	233	1	1043	147	1
Ultrajson	377	151	1	318	120	0
Aubio	453	92	1	349	85	0
Bottleneck	1359	634	7	1321	516	2
Pycurl	239	19	0	198	18	0
Simplejson	374	86	0	239	54	0
Msgpack	245	78	0	201	67	0
Pycryptodome	572	64	0	469	50	0
Jep	418	59	0	364	51	0
Jansi	332	244	1	313	211	0
Jna	711	189	0	671	181	0
Onenio	364	131	0	343	119	0
Zstdjni	151	21	0	145	19	0
Total	9295	2791	12	7853	2292	4
Improve	—			17.4% ↑	21.8% ↑	8 ↑

Table 5.9 Performance evaluation between PolyFuzz and AFL++.

Benchmark	PolyFuzz			AFL++		
	#Block	#Path	#Bug	#Block	#Path	#Bug
Efsprogs	1173	302	0	1066	217	0
Bind	4154	2982	0	3996	2813	0
Civetweb	232	157	0	198	145	0
Cyclonedds	1091	592	0	1016	531	0
Igraph	431	454	0	308	322	0
Total	7081	4457	0	6584	4028	0
Improve	—			7.6% ↑	11.4% ↑	0 −

single-language fuzzers.

Bug triggering. Notably, Only 4 of the 12 bugs uncovered by PolyFuzz were detected by PolyFuzz-NSA. indicating its weaker bug-finding capability compared to SASG-enhanced PolyFuzz. Nonetheless, PolyFuzz-NSA still outperformed single-language fuzzers in bug triggering (Table 5.8 vs Tables 5.3 and 5.4).

5.4.5 Real-world vulnerabilities discovery

Table 5.10 summarizes the 14 previously unknown vulnerabilities discovered by PolyFuzz, with 5 CVEs assigned.

5.5 Related Work

Single-language fuzzing. The majority of fuzzers concentrate on C/C++ [100], employing techniques like MOpt, GreyOne, REDQUEEN, ProFuzzer, and PATA for greybox fuzzing [98, 48, 8, 166, 92]. These fuzzers also prioritize efficient seed generation and selection strategies [167, 23, 156, 16, 117, 168], with some exploring reinforcement learning and code transformation to enhance their fuzzing capabilities [157, 119]. In contrast, PolyFuzz distinguishes itself through its approach: (1) treating the input as a seed-block stream rather

Table 5.10 Vulnerabilities detected by PolyFuzz.

Benchmark	#Bug	Status	PoC	Symptom	#CVE
Libsmbios	1	pending	✓	segment fault	0
Pillow	1	fixed	✓	out of memory	1
Ultrajson	1	fixed	✓	segment fault	1
Aubio	1	pending	✓	memory leak	0
Bottleneck	7	pending	✓	segment fault	1
Jansi	1	pending	✓	out of memory	1
Pyyaml	1	pending	✓	recursion error	0
Javaparser	1	confirmed	✓	JVM hung	1
Total	14	——			5

than a byte-stream, (2) predicting seed-block values through regression modeling instead of acquiring more seeds after mutation, (3) implementing sensitivity analysis-based seed generation with core regression modeling, and (4) dynamically selecting the regression model on-the-fly, setting it apart from Eclipser's fixed linear model [29] (refer to Algorithm 11).

Multi-language testing. Tools like AMLETO and GILLIAN employ custom intermediate representations (IRs) for testing embedded software and conducting multi-language symbolic execution, with a focus on languages such as VHDL, SystemC, JavaScript, and C [45, 144]. Similarly, the mutation testing tool in [54] facilitates mutant generation across various languages using regular-expression-based transformations. However, none of these tools specifically target the testing of programs composed of code units in multiple languages simultaneously, distinguishing them from PolyFuzz, which caters to the testing of *multilingual* code. Moreover, tools like FANS and FAVOCADO focus on fuzzing Android native system services and JavaScript engines, respectively, but they operate as single-language fuzzers, lacking the capability to handle cross-language code interaction [94, 41]. Notably, no prior work has explicitly addressed holistic fuzzing of multi-language programs as PolyFuzz does.

Cross-language security analysis. NDroid enables dynamic taint analysis (DTA) for JNI code in Android apps, revealing cross-language information leakage [162]. Likewise, PolyCruise facilitates application-level dynamic information flow analysis (DIFA) to uncover vulnerabilities at language interfaces in software units [86]. Despite their potential for identifying security issues, these analyses rely on existing run-time inputs, posing challenges for supporting additional language combinations. In contrast, PolyFuzz addresses these challenges by generating inputs to expose vulnerabilities in multi-language software, ensuring enhanced extensibility. Moreover, the cross-language DTA/DIFA approach in prior works can inform comprehensive multi-language fuzzing strategies, akin to the methodology employed in PolyFuzz.

Program intermediate representation (IR). Existing IRs such as LLVM-IR and Soot/Jimple [76, 75] cover complete code semantics, limiting their practicality as a unified IR across multiple languages. Conversely, the custom IR in PolyCruise is designed for complex data-flow analysis [88], unsuitable for greybox fuzzing. In contrast, the new custom IR in PolyFuzz is tailored for fuzzing, capturing critical information like control-flow/branching and value types, enhancing its compatibility with various languages [165]. It enables minimal language-specific analysis for enabling coverage measurement, gathering branch variables for regression model training. Harmonizing run-time value probing across languages renders the majority of PolyFuzz language-agnostic.

CHAPTER SIX

COLLABORATIVE FUZZING OF PYTHON RUNTIMES

With Python's pervasive usage and critical role, ensuring the security and reliability of its runtime system becomes paramount. Despite the ongoing reporting of real-world bugs in Python runtimes, automated bug detection tools are significantly lacking. To address this gap, I propose PyRTFuzz, a pioneering fuzzing technique engineered to comprehensively test Python runtimes, encompassing the language interpreter and its runtime libraries. Leveraging a combination of generation- and mutation-based fuzzing, PyRTFuzz seamlessly integrates static and dynamic analysis to extract runtime API descriptions. It employs a declarative specification language for generating diverse Python code and implements a customized type-guided mutation strategy for constructing structure-aware application inputs. Extensive evaluation across three major CPython versions revealed PyRTFuzz to be highly effective, uncovering 61 new demonstrably exploitable bugs, primarily within the interpreter and runtime libraries. This underscores the tool's scalability, cost-effectiveness, and significant potential for further bug discovery.

6.1 Motivation

Empirical study on CPython bugs. CPython, the primary implementation of Python [134], has been actively maintained for over two decades. Analyzing 98.3K historical issues from its repository, I identified 23.4K bug-related issues, with an average of over 1,000 reported annually since 2008. In the last five years, the number has consistently remained close to 2,000, as depicted in Figure 6.1. A manual examination of 500 bug-related issues indicated that over 98% of the bugs were triggered during Python application development, highlighting the necessity for robust testing tools to ensure CPython's quality and the potential for testing the language runtime through its applications.

Upon analyzing the bug distribution in CPython, it was found that a significant major-

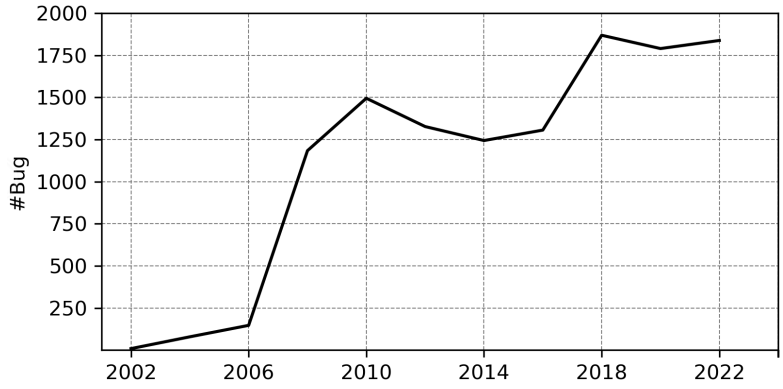

Figure 6.1 The number of bugs over twenty years in CPython.

ity (86.8%) of the bugs were located in the Python runtime libraries, while the remaining 13.2% were in the Python interpreter core. This analysis was conducted across 165 modules extracted from the CPython source code, with 164 modules having reported bugs. These findings underscore the importance of comprehensive testing for both Python runtime libraries and the interpreter core. They highlight the necessity for automated techniques to effectively test the entire Python runtime to uncover potential vulnerabilities.

```
1      program A as seed    | program B generated from A
2    ---------------------------------------------------
3    n = 4                  | n1 = 4
4    s = [0,3,5,8,7,9]      | s1 = [0,3,5,8,7,9]
5    for i in range (0, n):| for n2 in range (0, n1):
6        s [i] = n          |     for n3 in range (0, n2):
7        s [n] = i          |         s1[n3] = n2
8                           |         s1[n2] = n3
```

Figure 6.2 Example of grammar-based code generation.

Python runtime fuzzing. In Python runtime testing, generating diverse and valid applications is vital. However, existing techniques may not be enough. In Figure 6.2, program B is generated from program A using code bricks extracted via a grammar-based approach like CodeAlchemist. Despite being correct, such methods face limitations:

(1) Unawareness of diverse domains. Most bugs were in various Python runtime libraries

spanning 165 domains. Current methods focus on code generation but overlook runtime API utilization (Figure 6.2), potentially missing bugs.

(2) Insufficient application inputs. Covering various domains through diverse Python applications is insufficient for effective Python runtime testing. Recent CPython bug reports suggest that unit tests alone are insufficient, as they cover only a limited range of inputs to the runtime. Utilizing existing compiler testing techniques [58] may execute a newly generated program only once (as depicted in Figure 6.2), limiting the exploration of potential bugs with different input scenarios.

(3) Lack of holistic testing. Python applications operate within the Python runtime environment, encompassing both the interpreter core and runtime libraries. Focusing solely on either component is insufficient for comprehensive testing. To thoroughly assess the Python runtime, it is imperative to consider both the interpreter core and runtime libraries, along with their interactions.

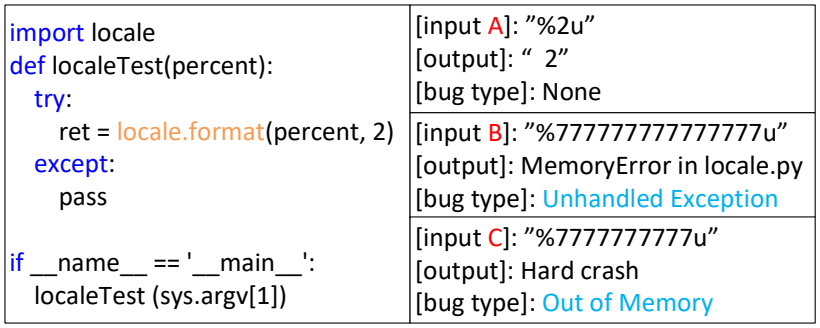

Figure 6.3 Motivating example: bugs occur in the interpreter and runtime library.

In Figure 6.3, the API locale.format is shown formatting the integer 2 using the percent input formatter. This example underscores several critical points: Firstly, the failure to generate applications for specific runtime modules like locale may overlook bugs within those modules (limitation (1)). Secondly, the generation of applications across various domains may not be adequate to uncover all bugs, particularly without considering diverse inputs (limitation (2)). Finally, the diverse results demonstrate the intricate interactions between the interpreter core and runtime libraries, emphasizing the need for holistic testing to address

such complexities (limitation (3)).

Driven by the empirical findings and observations, I propose a collaborative two-level fuzzing strategy to examine the Python runtime thoroughly. This approach integrates Level-1 generation-based fuzzing, responsible for creating Python applications, with Level-2 mutation-based fuzzing, which generates a diverse set of application inputs. By coordinating the two fuzzers, the framework aims to detect Python runtime bugs comprehensively.

6.2 Approach

In pursuit of comprehensive Python runtime testing, I introduced PyRTFuzz, an innovative two-level collaborative fuzzing framework. The initial step involves a hybrid analysis approach, utilizing static extraction and dynamic refinement to meticulously capture detailed runtime API descriptions. Leveraging these descriptions, I devised a specification-based language capable of generating domain-specific applications with varying control flow complexities. his process maintains data flow from application entry to API call through a top-down wrapping strategy. The fuzzing framework acts as a singleton, using one instance for both levels to share coverage feedback. Private data for each level, such as generators and seed queues, is initialized individually.

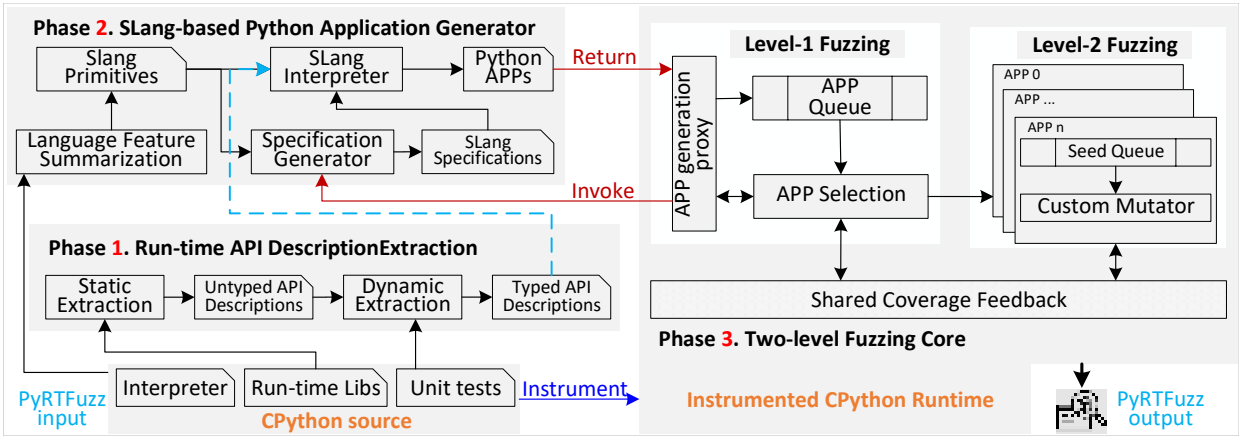

Figure 6.4 An overview of PyRTFuzz's architecture.

6.2.1 Overview of PyRTFuzz

In Figure 6.4, PyRTFuzz utilizes PyRTFuzz Inputs and the CPython source code in three main phases to uncover potential bugs or vulnerabilities. In Phase 1, the CPython source code is analyzed, extracting API description details through static and dynamic extraction. Static extraction analyzes runtime libraries module by module, presenting Untyped API Descriptions. Dynamic extraction involves running unit tests to obtain accurate type information, forming Typed API Descriptions for API-guided application generation.

In Phase 2, using Typed API Descriptions, PyRTFuzz employs a SLang-based Python APP generator (SLPyG) to create diverse applications. SLang scripts are randomly generated for a specified API and translated into Python applications with varied complexities. In Phase 3, the Two-level Fuzzing Core with shared coverage feedback operates at two levels. Level-1 Fuzzing uses a generation-based approach to fill the APP queue with Python applications for each API. APP selection initiates Level-2 Fuzzing on selected applications within a time budget. After one iteration of Level-2 Fuzzing, APP selection generates a more complex application with the associated API or selects another from the queue based on coverage changes. This iterative process continues until the APP queue is empty. Level-2 Fuzzing utilizes a mutation-based strategy with a Custom Mutator, working within a limited budget to efficiently cover all runtime APIs.

6.2.2 Run-time API description extraction

This phase is particularly crucial for Python application generation, aiming to accurately extract API descriptions in the following two subsections. Additionally, considering the multitude of Python versions, the dynamic and accurate extraction of API descriptions becomes even more essential.

6.2.2.1 Static extraction

In the static extraction phase, PyRTFuzz utilizes Python's standard AST parser to sequentially parse run-time library modules, extracting API descriptions. The fields of an API description are defined in Table 6.1.

Table 6.1 Field definitions for an API description.

No	**Field**	**Type**	**Comment**
1	module	string	module name
2	class	string	class name if exists
3	name	string	API name
4	argument	{string:string}	argument dict
5	return	{string:string}	return dict
6	exception	[string]	exception list may be thrown

Each defined field is crucial for generating functional Python applications. The module field specifies the module to import, and a valid class mandates creating an object before calling the API. Each API includes a 3-tuple of attributes: (name, argument, return). For an API with a given name, all arguments are cataloged in a dictionary, where the key represents the argument name, and the value signifies the argument type. Variable parameter APIs include a label "...:..." in the dictionary. In the static phase, the 'argument' and 'return' dictionaries start with None (Untyped), with subsequent updates in the dynamic phase. This is due to the limitations of static inference [60, 120, 106] in ensuring accurate type inferences. The 'exception' field meticulously compiles explicit exceptions triggered by the API or implicit ones from imported modules. This information is instrumental in guiding the handling of exceptions during application generation (§6.2.3).

As a reminder, functions extracted from source code may not all be the open APIs published to developers. Only those that pass the Import test are reserved in descriptions

to filter out these internal functions.

6.2.2.2 Dynamic extraction

Utilizing Untyped API descriptions as input, PyRTFuzz executes and scrutinizes unit tests to extract types in dynamic extraction, following the process outlined in Algorithm 13.

Algorithm 13: Dynamic API type extraction

```
   Input: U: unit test set
   Input: utSpec: untyped API specifications
   Output: tSpec: typed API specifications
1  Function dynTypeExtract (U, utSpec)
2      foreach u_i in U do
3          F ← dynInspect (u_i) ;                          //inspect, get all functions' frames
4          foreach f_i in F do
5              N ← getApiName (f_i) ;
6              Des ← getApiDescript (N, utSpec);
7              if Des is None then
8                  continue;
9              foreach arg in Des.arg do
10                 tDes ← updateArgType (Des, arg, f_i);
11             tDes ← updateRetType (f_i) ;
12     return tSpec;
```

PyRTFuzz processes one unit test in each iteration (line 2). For a given test, PyRTFuzz executes and examines the frame objects of functions (line 3), capturing local variables like parameters and return variables. From each frame f_i, it extracts the function name (line 5), which serves as input to retrieve the corresponding API description (line 6). When Des is valid, PyRTFuzz uses the argument names as keys to obtain actual parameters (live objects) from frame f_i, extracting types and updating Des to tDes. It also extracts the return type (lines 9–11).

While offering high accuracy, dynamic extraction may encounter false negatives due to incomplete execution coverage. In the case of PyRTFuzz, API type extraction only requires function-level coverage, which is achievable through unit testing. Furthermore, Python allows

variation in API argument or return types; PyRTFuzz ensures extraction of at least one type for each argument or return, ensuring the capability to generate valid application inputs.

6.2.3 SLang-based Python application generation

To comprehensively fuzz Python runtimes, SLang-based Python APP generation (SLPAG) focuses on generating diverse Python applications with APIs from the run-time libraries. It possesses three key capabilities: (1) constructing applications for different APIs, (2) creating applications with varied control flow complexities for a specified API, and (3) maintaining data flow from the application's entry to the API's call site for a specified application of an API. To achieve this, SLPAG introduces an API-independent, extensible script language (SLang) and designs an SLang interpreter to interpret and translate a SLang script (a sequence of SLang statements) into a Python application.

6.2.3.1 Slang definition

Accordingly, the formal syntax of Slang is as follows:

$$
\begin{aligned}
P &::= S^* \\
S &::= [c =]C(e)^* \\
e &::= c \mid A
\end{aligned}
$$

In SLang, a program (script) P is a sequence S^* of statements. Each statement S is exclusive of the *assignment* type. In every assignment, the right-value $C(e)^*$ signifies a built-in command C operating on an expression e, and the left-value c represents the result of $C(e)^*$. As an expression, e can be a variable c or an expression of a Python API A.

SLang primitive. A SLang primitive can be summarized but not limited to Python's language features (e.g., control flow structures). In PyRTFuzz, the primitives are designed into two categories: basic and extend.

Basic. The basic primitive is utilized to establish the foundational structure of a program, be it object-oriented or procedure-oriented, and to specify the program's entry point. This

primitive is tailored to exclusively accept an expression of a Python API as its input. Hence, each SLang script must initiate with precisely one basic primitive.

Algorithm 14: Procedure of a basic primitive

Input: *API*: a Python API description

Output: *P*: a Python application

```
1  Function basicCmd (API)
2      T ← typeList (API);                                   //prob API type list
3      P.insert (T);
4      D ← newFunc (demoFunc, argd);
5      p_d ← getPara (D);
6      paras ← decodeArgs (T, p_d);                          //prob decoder for API arguments
7      s_d ← newCall (API.name, paras);
8      D.insert (wrapExcp (s_d));
9      P.insert (D)
10     M ← newFunc (P, SLmain, argm);
11     p_m ← getPara (M);
12     s_m ← newCall (demoFunc, p_m);
13     M.insert (s_m);
14     P.insert (M)
15     return P;
```

Algorithm 14 delineates the comprehensive process for a basic primitive. It inputs a Python API description and initializes the program P with the probed API argument type list T (line 3). The algorithm begins by creating a demoFunc to encapsulate the invocation of the API (line 4–8). Notably, to facilitate input format-aware mutation, a function called decodeArgs is probed to decode the input (an encoded byte stream from the Fuzzer) into individual arguments for the API call (line 6). This ensures the type correctness of arguments. Furthermore, the invocation of the API is encapsulated in a try–except block to filter out expected exceptions and allow the Fuzzer to capture unhandled exceptions. The definition of demoFunc is then inserted into program P. Subsequently, the algorithm creates the main function SLmain for P (line 10–13). Following a similar procedure, SLmain invokes demoFunc and inputs its formal parameters to maintain inter-procedure data flow. Finally, P designates SLmain as its entry point for fuzzing.

Extend. The extend primitive is a flexible construct that can be derived from language control flow structures or programming patterns observed in real-world software development. Unlike the basic primitive, the extend primitive provides enhanced flexibility by allowing it to take the results of other primitives as input.

Algorithm 15: Procedure of an extend command

Input: P: a previously generated Python application

Output: P': a new Python application

```
1 Function extendCmd (P)
2     M ← getMain (P);                                   //get SLmain
3     B_M ← getBody (M);
4     p ← getPara (M);
5     Block ← newBlock (B_M, p);                         //wrap SLmain's body into a new block
6     P' ← setBody (M, Block);
7     return P';
```

Algorithm 15 outlines the procedure for a extend primitive, generating new applications by top-down wrapping the input program. Given a previously generated program P, the extend primitive retrieves the main function M (i.e., SLmain) along with its body and parameters. It then wraps M's body into a new block and replaces M's body with the block to create a new program P'. The new block can be either intraprocedural (e.g., a for or if block) or interprocedural (e.g., a function call). For the intraprocedural case, it wraps M's body to the block and then pushes it back. For the interprocedural case, it generates a new function with the same parameters as M, inserts M's body into the new function, and replaces the body of M with invoking the new function. This top-down wrapping approach ensures a seamless data transfer from the top (i.e., SLmain) to the bottom (i.e., the API call site) for primitives inducing interprocedural control and data flows, satisfying the data flow requirement from the application's entry point to the Python API's call site.

Seven SLang primitives are built on Python abstract syntax tree (AST) operators, as illustrated in Table 6.2. These primitives cover both basic intra-procedural (e.g., If) and inter-procedural (e.g., Call) control flow structures. Each primitive is designed to operate

Table 6.2 SLang primitives implemented in PyRTFuzz.

No	Command	Type	Comment
1	OO	basic	object-oriented program
2	PO	basic	procedure-oriented program
3	While	extend	a while structure wrapper
4	For	extend	a for structure wrapper
5	If	extend	a if structure wrapper
6	Call	extend	a call structure wrapper
7	With	extend	a with structure wrapper

universally on all Python runtime APIs, ensuring independence from the syntax or semantics of specific API usage. This quality of SLang fulfills the capability requirement (1).

```
1   OO (sqlite3.dbapi2.DateFromTicks)
2   ----------------------------------------
3   TypeList = ['ticks:int']
4   class demoCls():
5      def demoFunc(self, p):
6          try:
7              ticks = decodeArgs(TypeList, p)
8              sqlite3.dbapi2.DateFromTicks(ticks)
9          except (AssertionError) as e:
10             pass
11  def SLmain(x): # entry point
12      dc=demoCls()
13      dc.demoFunc(x)
```

Figure 6.5 Primitive OO to Python APP of equivalent semantics

In Figure 6.5, an illustration of applying the OO primitive to the Python API DateFromTicks in the library of sqlite3.dbapi2 (line 1) is presented. The corresponding Python application (lines [3–13]) demonstrates equivalent semantics. The variable TypeList specifies the name and type of the API's parameters. The OO command generates a new class named

demoCls and encapsulates the API within its method demoFunc (lines [6–10]). Specifically, a new variable ticks is defined by decoding the input parameter p (line 7). The call site for the API is then created and enveloped within a try-except block. Finally, the entry point SLmain is initialized, and a context for invoking demoCls.demoFunc is constructed, completing the application generation.

6.2.3.2 SLang specification generator

In accordance with the definition outlined in §6.2.3, a SLang specification comprises a series of SLang statements. Subsequently, a SLang specification generator (SLangSG) orchestrates the creation of a SLang specification, adhering to the process delineated in Algorithm 16. This entails the systematic construction of SLang statements sequentially utilizing primitives.

Algorithm 16: Generate a SLang specification

Input: Prm: SLang primitive set
Input: API: Python API name
Input: N: Statement number (≥ 1)
Output: SS: SLang specification

```
1  Function genSpecification (Prm, API, N)
2      P_b ← selectPrm (Prm, basic) ;          //randomly select basic primitive
3      S ← genStmt (R, P_b, API) ;             //statement: R = P_b (API)
4      pushBack (SS, S);
5      N ← N − 1;
6      while N > 0 do
7          P_e ← selectPrm (Prm, extend);      //randomly select extend primitive
8          S ← genStmt (R, P_e, R);            //statement: R = P_e (R)
9          pushBack (SS, S);
10         N ← N − 1;
11     return SS;
```

SLangSG, taking inputs of the primitive set (Prm), Python API name (API), and the statement number (N), executes a two-step generation process. Initially, it constructs a basic command statement by randomly selecting a basic primitive P_b from Prm and crafting an assignment statement R = P_b (API), where R functions as a built-in variable for storing the

outcomes of P_b on API. This inaugural statement sets the overall structure of the Python application tailored for API. Subsequently, it iteratively generates extend command statements in each iteration, where R = P_e (R) is formed by randomly selecting a extend command P_e from Prm, with P_e taking R as input and storing the result back into R. This process repeats until the script reaches the specified length N. This strategy, involving the selection of basic and extend primitives and determining the specification's length N, ensures the generation of Python applications for a specific API with diverse control flow complexities and programming patterns, satisfying the capability requirement (2).

```
1    R = OO (sqlite3.dbapi2.DateFromTicks)
2    R = For (R)
3    R = Call (R)
```

Figure 6.6 An example of SLang specification with three statements.

As an example, considering the API sqlite3.dbapi2.DateFromTicks, Figure 6.6 illustrates a randomly generated SLang specification comprising three statements. The first statement employs a basic primitive, OO, to produce an object-oriented program for the API, with the outcome stored in the built-in variable R. The second statement utilizes an extend primitive, denoted as For, on R, wrapping a for-structure around the program R and storing the result back in R. Analogously, the third statement employs a Call command to encapsulate a call structure around the program R, and the result is once again stored in R. Ultimately, R retains the execution results of the specification.

6.2.3.3 SLang interpreter

With a SLang specification as input, the SLang interpreter parses the specification and executes each statement in sequence to translate the SLang to Python.

The execution details of the SLang specification are outlined in Algorithm 17. Initially, it initializes the built-in variable R and the API spec Spec as None (line 2–3). Subsequently, it loads all SLang statements into memory, labeled as L_s (line 4), and proceeds to parse

and interpret the statements sequentially (line 5–13). For each statement, the interpreter extracts the primitive Cmd and corresponding input Arg (line 6). If Arg is the built-in variable R, the interpreter loads the value of R as input and executes Cmd (line 9). Here, R stores the program generated by the last statement, enabling the continuous superposition of primitives to create highly complex programs. When Arg is an API spec, the interpreter retrieves the API specification and stores it into Spec (line 11) as input for the execution of command Cmd. The execution result is then assigned to the variable R (line 12). After executing a statement, the temporary result (R) is stored in the stack for the next iteration. Once all statements are executed, the interpreter retrieves the Import information from the API spec Spec, inserts it into R, and generates the final Python application PyApp as the output (line 14).

Algorithm 17: Intepret SLang specification to Python APP

```
   Input: SS: SLang script
   Input: PySpec: Python API specs
   Output: PyApp: Python application
1  Function intepretSLang (SS, PySpec)
2      R ← None;                                   //Initialize built-in variable R as None
3      Spec ← None;                                //Initialize API spec as None
4      L_s ← parseSript (SS);                      //parse the script and get statement list
5      foreach s in L_s do
6          Cmd, Arg ← parseStmt (s);
7          if Arg is 'R' then
8              R ← loadR ();
9              R ← executeCmd (Cmd, R);
10         else
11             Spec ← getApec (PySpec, Arg);
12             R ← executeCmd (Cmd, Spec);
13         storeR (R)
14     PyApp ← import (Spec, R);                   //import all dependent modules
15     return PyApp;
```

Figure 6.7 illustrates a SLang specification (Figure 6.6) interpreted into a Python application. Essentially, each primitive encapsulates the input program in a top-down manner.

Firstly, a simple program is generated by translating the OO statement, as shown in Figure 6.5. Then, For wraps all the statements of SLmain (at the top level) into a for loop and embeds the for block back into SLmain. Similarly, in the third statement, Call wraps all the statements of SLmain into a new function, PyCall_1681926341, and embeds its invocation with input x back into SLmain. The distinction lies in the fact that the Call primitive induces inter-procedure control flow. Both the formal and actual parameters of PyCall_1681926341 are assigned as the formal parameters of SLmain, ensuring the correctness of data flow. Consequently, the final Python application is generated with import information.

```
1     from pyrtfuzz import decodeArgs
2     import sqlite3
3     TypeList = ['ticks:int']
4     class demoCls():
5       def demoFunc(self, p):
6         try:
7            ticks = decodeArgs(TypeList, p)
8            sqlite3.dbapi2.DateFromTicks(ticks)
9         except (AssertionError) as e:
10            pass
11     def PyCall_1681926341(x):
12         for F_g1 in range(0, 1):
13             dc=demoCls()
14             dc.demoFunc(x)
15     def SLmain(x): # entry point
16         PyCall_1681926341 (x)
```

Figure 6.7 Generated Python application from SLang specification in Figure 6.6

6.2.4 Two-level fuzzing core

During this phase, the fuzzing core operates within instrumented Python runtimes, employing a two-level Fuzzing approach encompassing both generation-based Level-1 Fuzzing and mutation-based Level-2 Fuzzing. The collaboration between these two levels, facilitated by shared coverage feedback, automates Python runtime testing, covering the interpreter core

and runtime libraries. The two-level design is structured to address distinct dimensions of Python runtime. In essence, Level-1 Fuzzing is geared towards generating diverse applications, providing a thorough examination of the Python interpreter. This phase spans multiple application domains, each featuring its runtime APIs. Within each domain, the SLang-based approach facilitates the creation of applications exhibiting varied complexities.

On the flip side, Level-2 Fuzzing focuses on generating diverse inputs and executing applications specified by Level-1 Fuzzing. Here, the dynamic extraction (outlined in §6.2.2) contributes precise application input types, enabling custom mutation tailored to each domain. This phase places a strong emphasis on testing the reliability of the runtime libraries, complementing the efforts of Level-1 Fuzzing. The coordinated efforts of these two levels ensure a comprehensive evaluation of the entire Python runtime.

To optimize fuzzing efficiency, I devised a unified custom mutator applicable to all applications. This mutator integrates the API type list and a data decoder into the target applications, as illustrated in Figure 6.7. Throughout Level-2 Fuzzing, the mutator generates type-correct values for each type in the TypeList variable, encoding them into a byte stream, and subsequently decoding them into individual API arguments within the applications. This approach eliminates the need to develop domain-specific mutators for each API, opening avenues for exploring deeper execution paths with type-correct inputs.

Algorithm 18 provides a comprehensive view of the two-level collaborative fuzzing loop within PyRTFuzz. The process commences with Level-1 fuzzing, establishing a dedicated Python application generation server with a remote invocation handle (H) for iterative API testing. Initial applications (InitApps) are generated for all APIs based on Python API descriptions (PyDesc), and after a calibration step to filter out non-executable or low-potential applications, the remaining ones are queued for Level-1 fuzzing. Comprehensive coverage feedback is ensured by dynamically instrumenting Python code in the runtime libraries. The fuzzer then continuously selects and fuzzes applications, monitoring for crashes or unexpected behaviors. Within each Level-1 iteration, a nested loop of Level-2 fuzzing is triggered

Algorithm 18: Overall procedure of the holistic two-level fuzzing

Input: *PyDesc*: Python API descriptions

Output: *Bugs*: Bugs detected

```
1  Function levelOneFuzz (PyDesc)
2      H ← startPyGen (PyDesc)                    //initialize the server handle for Python code generation
3      InitApps ← genInitAPPs (H);                //generate initial APPs for each API
4      OkApps ← validate (InitApps);
5      AppQueue ← initAppQueue (OkApps);
6      probPyRuntime ();                          //dynamic instrumentation of the Python runtime
7      while true do
8          foreach App in AppQueue do
9              while true do
10                 S ← randomSeeds ();
11                 B ← randomBudget ();
12                 M ← loadFuzzMain (App);
13                 levelTwoFuzz (M, S, B);
14                 covChanged ← covFeedback ();
15                 if covChanged == false then
16                     break;
17                 App ← genPyApp (H, App);       //generate more APPs around the API
```

for the selected application. Level-2 fuzzing begins by generating random seeds (S) and a time budget (B), followed by loading the main function (M) of the application. This level operates as traditional mutation-based fuzzing within the allocated time budget. After completion, overall coverage variation is collected, and new coverage detection prompts the fuzzer to continue operating on the API used in the application, generating more applications for Level-2 fuzzing.

6.2.4.1 Custom mutation at level-2

Diverging from the conventional holistic mutation strategy applied by greybox fuzzers, PyRTFuzz pioneers a tailored mutation approach to augment the efficiency of Level-2 fuzzing. This approach involves individually mutating values of distinct variables in the application input based on their types. The required type information is extracted from associated API descriptions obtained in Phase 1. Subsequently, the mutated values of these input variables are encoded into a byte sequence, collectively conveyed by the Level-2 core to the current application. During the execution of the fuzzed application, the mutated input byte sequence undergoes decoding into individual API arguments just before the API is invoked. This decoding process is facilitated by a probe seamlessly inserted into the application during its generation in Phase 2. The type-guided custom mutator ensures the production of type-correct values tailored to each specific runtime API. In scenarios where this input-format-aware mutation fails to deliver substantial runtime coverage gains over consecutive iterations, the Level-2 core might revert to the default mutation strategy.

6.3 Implementation

Implemented in accordance with the outlined design in Figure 6.4, PyRTFuzz comprises three core components: API description extraction, the SLang-based Python application generator, and the Two-level fuzzing core.

API description extraction. StExtr, the static analyzer, utilizes the standard Python

AST parser [136] to translate Python libraries from the CPython source into ASTs. It systematically iterates through all AST nodes to extract features outlined in §6.2.2. The obtained information is then stored in XML format as untyped API descriptions. For a more precise extraction of data type, I introduced DynExtr, a dynamic analyzer. DynExtr scrutinizes all CPython unit tests by leveraging Python's built-in tracing API (sys.settrace) to systematically update argument and return types within the API descriptions.

SLang-based Python application generator. To facilitate SLang-based Python application generation, I developed three sub-components. Initially, I designed a set of Python AST-based operators that facilitate AST editing, simplifying the development of SLang primitives. Leveraging these operators, I crafted seven primitives (refer to Table 6.2). Subsequently, I implemented a SLang interpreter responsible for translating SLang specifications into Python applications. Lastly, I created a suite of Remote Procedure Call (RPC) interfaces to enable remote invocation by the fuzzer.

Two-level fuzzing core. The implementation of the two-level fuzzing core utilizes Atheris [50] and libFuzzer [95]. Initially, interfaces for level-1 fuzzing were added to Atheris. To enable the execution of level-2 fuzzing within the same process as level-1, the entry point of each application (i.e., SLmain) was dynamically imported and executed through invoking SLmain. Subsequently, in libFuzzer, a single fuzzing core was initialized for the two-level fuzzing, allowing the sharing of coverage feedback between the levels. For the seed queues, one queue was initialized for level-1 fuzzing to store applications, while different queues were created for each application in level-2 fuzzing to store favored seeds, ensuring non-interaction between different level-2 fuzzing instances on applications. Regarding the level-2 time budget, it was implemented as a resizable time window, with the size reset based on coverage changes. This approach grants more fuzzing time to applications, triggering a higher number of covered basic blocks. The custom mutators support the random generation of values for the top 20 data types commonly used in Python runtimes, including integers, floats, strings, and lists.

The entire implementation of the two-level fuzzing core comprises 14KLoC of code,

encompassing 11.4KLoC of Python, 2.1KLoC of C++, and 0.5KLoC of Shell. Extensive testing was conducted on three commonly used CPython versions, including Python3.7.15, Python3.8.15, and Python3.9.15.

Limitations. PyRTFuzz operates as an in-process fuzzer, limiting its coverage feedback to a single running process. Consequently, APIs related to multi-process functionality, such as multiprocessing and pipe, cannot be fuzzed by PyRTFuzz. When generating Python applications, PyRTFuzz supports individual APIs without accounting for potential dependencies among them. This design choice can lead to two potential issues: (1) The generated applications may deviate from fair API usage practices, yielding unrealistic scenarios and potentially false-positive bug detections. (2) The applications may lack executability due to incomplete environments. For instance, an API relying on global variables or class members initialized by other APIs may encounter issues if the necessary initializations are absent in the generated application.

6.4 Evaluation

The evaluation of PyRTFuzz covers three main aspects: its effectiveness in fuzzing the Python runtime, the scalability of Python application generation, and the factors influencing its effectiveness. As the first fuzzer capable of fuzzing the entire Python runtime, direct comparisons with other tools are currently impractical. Nevertheless, extensive experiments on a 64-bit Ubuntu 18.04 system, featuring a 32-core CPU (AMD Ryzen Threadripper 3970X) and 256 GB memory, were conducted. Each fuzzer was run against target applications with identical configurations on a single CPU core for 5×24 hours and the experiments were repeated five times for reliability.

Benchmarks. Evaluated across three widely used versions of CPython—Python 3.9.15, Python 3.8.15, and Python 3.7.15—PyRTFuzz's effectiveness is summarized in Table 6.3. The table includes information on code size, the count of APIs and typed-APIs, and the number of valid initial seeds (applications). The Typed-API column indicates the number

of successfully extracted API descriptions, covering around 70% of APIs due to potential gaps in unit test coverage. Initial seeds for level-1 fuzzing were generated by PyRTFuzz with one application per runtime API, followed by calibration and exclusion of failed applications, retaining a significant proportion of initial seeds (e.g., 91.3% for Python 3.9.15). Initial seeds for level-2 fuzzing were randomly generated.

Table 6.3 Profiles of the 3 released CPython versions.

Benchmark	**Size (KLoC)**	**#API**	**#Typed-API**	**#L1-Seeds**
Python3.9.15	C: 529.5 Python: 287.6	4,208	2,998 (71.2%)	3,844 (91.3%)
Python3.8.15	C: 487.7 Python: 277.3	4,184	2,855 (68.2%)	3,481 (83.2%)
Python3.7.15	C: 416.4 Python: 267.9	4,115	2,773 (67.4%)	3,244 (78.8%)

Performance metrics. The evaluation of PyRTFuzz focused on two key metrics: the number of covered basic blocks and the detection of triggered bugs. Coverage results were obtained by averaging the count of basic blocks covered across five repetitions, each lasting 5×24 hours, ensuring comprehensive coverage of all runtime APIs. The assessment of bug detection involved a manual validation process, scrutinizing reported issues such as crashes, hang-ups, or unhandled exceptions. For each reported bug, a proof of concept (PoC) was created to reproduce the triggering inputs, and a bug was deemed valid only if its call stack differed from all other confirmed bugs.

6.4.1 Effectiveness of PyRTFuzz

Coverage. Figure 6.8 illustrates the evolving coverage dynamics of PyRTFuzz on Python 3.9.15 with specific parameter settings, including a 90-second time budget for level-2 fuzzing and a maximum APP specification size of 256. The left subplot depicts the continuous generation of applications by level-1 fuzzing, steadily increasing their numbers (green line). Notably, there are intervals, like the 40h–50h time zone, where the growth rate of applications slightly decelerates, indicating that applications generated during these intervals cover more

basic blocks. Simultaneously, the growing coverage (black line) corresponds to the expanding application count, with varied growth rates across time zones due to the diverse nature of generated applications. The right subplot normalizes the data per unit time zone (8 hours), providing a standardized view of coverage changes per application. Applications generated during the 40h–50h time zone exhibit a higher coverage per application, suggesting that level-2 fuzzing allocates more time to these applications while level-1 generates fewer applications, which aligns with the observed slowdown in application growth.

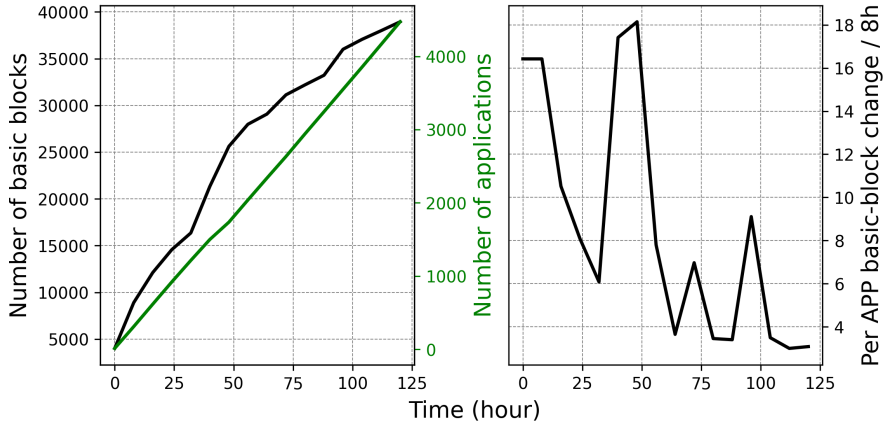

Figure 6.8 Coverage evolves over the timeline on Python 3.9.15.

Bug triggering. Using a 5 × 24 hours time budget, PyRTFuzz successfully identified a total of **61** bugs across three Python versions: **25** in Python 3.9.15, **15** in Python 3.8.15, and **21** in Python 3.7.15. After removing duplicate reports, 45 unique bugs were identified. A detailed breakdown is provided in Table 6.4.

6.4.2 Scalability of Python application generation

Figure 6.9 depicts the impact of different APP specification sizes on the time cost and memory usage during both APP specification and Python APP generation in PyRTFuzz. The left subplot reveals that for APP specification generation, the time cost and memory usage remain negligible even with a specification size of 4,096. However, in Python APP generation, both time and memory exhibit a linear relationship with the APP specification size. With a maximum specification size of 4,096, it takes 2,714.47 seconds to generate

Table 6.4 Bugs detected by PyRTFuzz.

Benchmarks	Bug Type	#Bug	PoC
Python 3.9.15	MemoryError	15	✓
	Out of Memory	4	✓
	RecursionError	3	✓
	Hang up	3	✓
Python 3.8.15	MemoryError	12	✓
	Out of Memory	2	✓
	Stack Overflow	1	✓
Python 3.7.15	MemoryError	8	✓
	RecursionError	6	✓
	Stack Overflow	4	✓
	Hang up	2	✓
	Out of Memory	1	✓
Total		61	—

Python APPs using 291.71 MB of memory. Considering the time budget constraints, the maximum acceptable APP specification size for PyRTFuzz is determined to be 1,024. The right subplot illustrates a linear correlation between Python APP and specification sizes with the implemented specification primitives, suggesting that increasing the specification size generally results in more complex Python APPs.

6.4.3 Factors affecting effectiveness

In adherence to the design considerations, three pivotal factors influencing the effectiveness of PyRTFuzz were identified: APP specification size, level-2 time budget, and the utilization of typed or untyped API descriptions. To assess their impact, fuzzing experiments on Python 3.9.15 were conducted for 5×24 hours, employing various values for each factor.

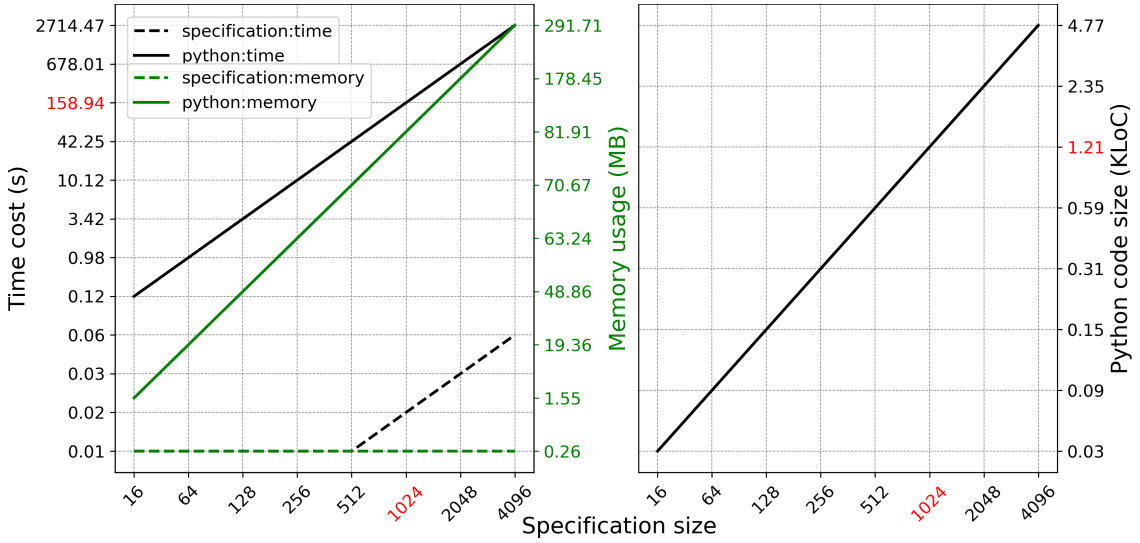

Figure 6.9 The time costs of Python application generation over specification sizes.

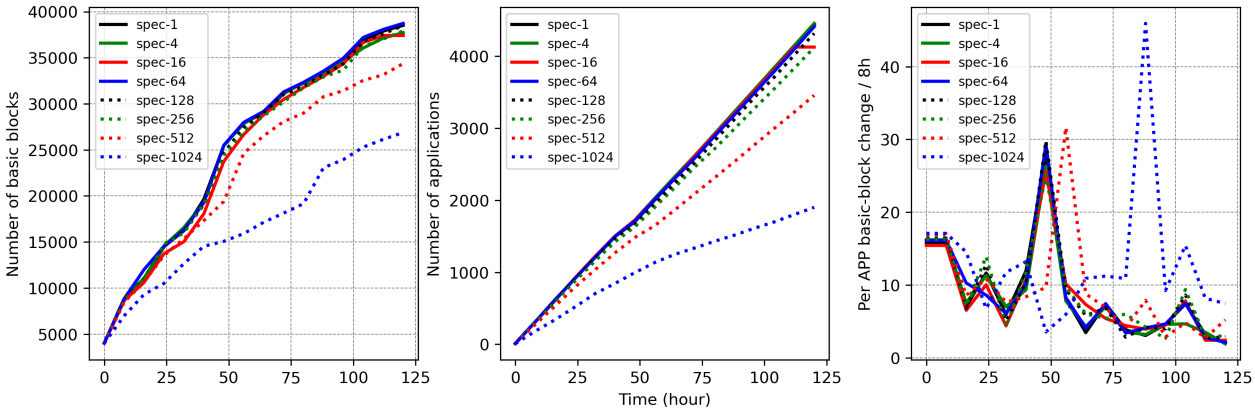

Figure 6.10 Coverage evolves with different APP specification sizes.

6.4.3.1 APP specification size

In Figure 6.10, different PyRTFuzz instances with APP specification sizes ranging from 1 to 256 exhibit nearly identical coverage trends, consistently surpassing the trend associated with specifications [512, 1024]. The [512, 1024] specification requires more time for APP generation (as indicated in §6.4.2), resulting in fewer generated APPs, and potentially less time for level-2 fuzzing. After normalizing the data, the [1024] specification achieves the highest basic blocks per application between [75, 100] hours, suggesting that larger specification sizes can potentially trigger more block coverage in level-2. Therefore, achieving a balance between generating APPs (level-1) and the APP fuzzing process (level-2) can improve coverage.

6.4.3.2 Level-2 time budget

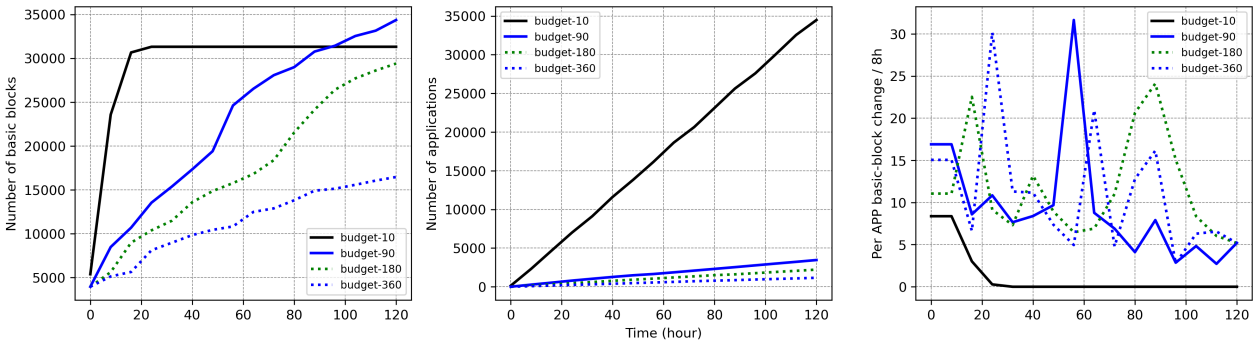

Figure 6.11 Coverage evolves with different level-2 budget.

In Figure 6.11, the coverage evolution of PyRTFuzz on Python 3.9.15 with different level-2 time budgets is illustrated. The left subplot demonstrates significant variations in overall coverage, with a rapid increase observed with a 10-second budget, resulting in a plateau after 20 hours. This suggests that too small a budget might hinder deep-path exploration. In contrast, a 90-second budget shows a continuous increase in coverage, outperforming budgets of 180 and 360 seconds. The influence of the level-2 budget on PyRTFuzz's effectiveness becomes apparent as it directly affects the discovery of additional basic blocks. While an increase in the budget tends to uncover more basic blocks, it's crucial to note that an

excessively large budget may introduce redundancy in the analysis, potentially leading to diminishing returns. Striking the right balance in budget allocation is essential for optimizing PyRTFuzz's performance in uncovering vulnerabilities.

6.4.3.3 Typed and untyped API descriptions

Figure 6.12 presents a comparative analysis of PyRTFuzz fuzzing on Python 3.9.15 using typed (solid) and untyped (dot) API descriptions.

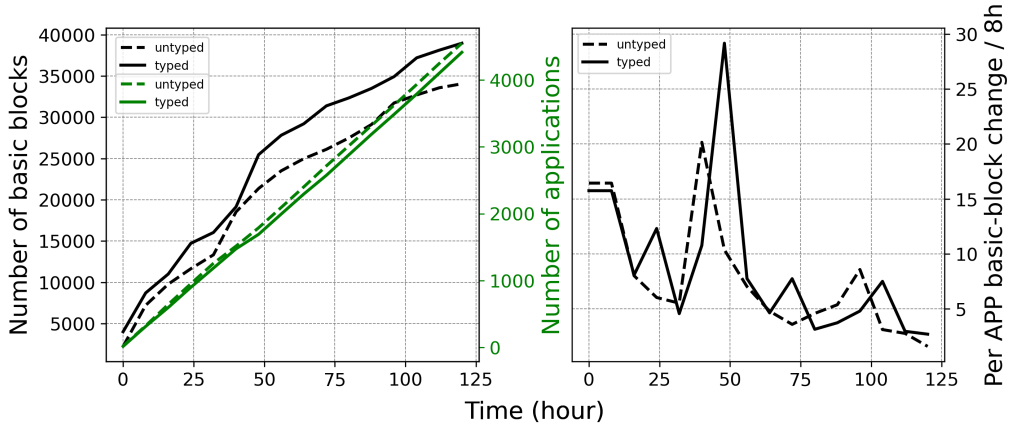

Figure 6.12 Coverage evolves with typed and untyped API descriptions.

The left subplot highlights PyRTFuzz 's consistent superiority with typed API descriptions regarding coverage feedback. While the untyped version may generate slightly more applications, this is attributed to the limited exploration of deep execution paths within a fixed time window, resulting in fewer new basic blocks being triggered. Consequently, the untyped version quickly exhausts the time window, transitioning to level-1 APP generation for subsequent level-2 fuzzing. In contrast, typed applications, with increased likelihood of exploring deep paths and triggering new coverage, enable PyRTFuzz to reset the time window and prolong level-2 fuzzing. After normalization (as shown in the right subplot), the peak value of the typed version is nearly 50% higher than that of the untyped version, underscoring the superior effectiveness of typed API descriptions in triggering basic blocks. Despite constraints in implementation, such as support for a limited set of data types and approximately 70% success in type extraction, these findings emphasize the substantial impact

of type information on enhancing PyRTFuzz's effectiveness, resulting in an improvement of up to 20%.

6.5 Related Work

Compiler fuzzing has seen advancements through techniques like generation-based fuzzers such as JSfunfuzz, TreeFuzz, Skyfire, and mutation-based counterparts like Superion, Fuzzil, LangFuzz, and DeepSmith [143, 118, 158, 159, 55, 63, 35]. However, these methods often prioritize testing compilers or interpreters while overlooking runtime libraries—a critical component of the language runtime. In contrast, PyRTFuzz pioneers a unique two-level collaborative fuzzing strategy to thoroughly assess the entire Python runtime, including both interpreter and runtime libraries [133].

Unlike conventional fuzzers confined to a single level, PyRTFuzz integrates both generation-based and mutation-based approaches in a two-level synergy. Collaborative fuzzers like EnFuzz, CollabFuzz, and Cupid collaborate at a single level, treating entities equally [26, 115, 57]. In contrast, PyRTFuzz introduces a distinctive two-level collaboration, where level-1 generates diverse applications for level-2 and receives feedback to guide its application generation [90].

In the domain of Python analysis and testing, earlier works such as PyPrecditor, PolyCruise, PolyFuzz, and Atheris showcase capabilities in analyzing and testing Python applications [161, 86, 89, 50]. However, PyRTFuzz stands out by prioritizing the comprehensive testing of the complete Python runtime, covering both the interpreter and runtime libraries [90]. This unique positioning establishes PyRTFuzz as an innovative and comprehensive solution for Python runtime testing.

CHAPTER SEVEN

FUTURE WORK

Looking ahead, the widespread adoption of multi-language software is anticipated across various domains. Ensuring the security of both the software and the hosting language runtimes becomes paramount within these diverse domains. Expanding upon my current endeavors, my forthcoming objectives are outlined as follows: Firstly, my unwavering commitment lies in advancing security testing for cross-language applications. Subsequently, my focus shifted toward the refinement of collaborative fuzzing techniques. This strategic enhancement aims to achieve a thorough testing framework for the Python runtime and subsequently, to extrapolate this refined approach to encompass a broader spectrum of language runtimes. Furthermore, I am dedicated to addressing the forefront of current interests. I am actively immersed in the security analysis of AI compilers and runtimes, encompassing prominent platforms such as PyTorch and MindSpore.

Cross-language security testing. PolyCruise and PolyFuzz have made significant advancements in cross-language information flow analysis and comprehensive greybox fuzzing. However, there is still room for improving cross-language vulnerability discovery. While PolyFuzz has enhanced fuzzing efficiency through semantic relationships and whole system coverage, its current limited semantics don't accurately model complex data flow information, which is crucial for vulnerability exploration, as prior research indicates. To address this, my next focus is on seamlessly integrating cross-language information flow analysis into holistic fuzzing. The plan involves developing methods like taint-guided holistic fuzzing using insights from cross-language data flow analysis and exploring directed fuzzing based on potentials identified by cross-language data flow analysis. These strategies aim to concentrate fuzzing efforts on areas highlighted by data flow analysis as having higher vulnerability likelihood, thus optimizing resources and intensifying security weakness identification.

Language runtime fuzzing. The SLang-based approach has successfully generated appli-

cations with varying control flow complexities tailored to specific runtime APIs. Nevertheless, it falls short in representing realistic software scenarios, mainly due to the lack of potential dependencies among runtime APIs. This deficiency hampers the ability to simulate real-world application interactions, potentially limiting its effectiveness in capturing intricate software behaviors. Additionally, comprehensive testing of interpreters necessitates consideration of broader language features or characteristics beyond control flow structures. Focusing solely on control flow features could result in incomplete testing, potentially overlooking critical language behaviors.

Furthermore, the security of mainstream language compilers and runtimes hosting various languages has emerged as a critical concern within multi-language software. This challenge is particularly pronounced in languages like JVM, JavaScript, Python, and their combined scenarios. Despite the theoretical promise of the proposed approach to span different language runtimes, practical implementation encounters obstacles stemming from the nuanced differences among languages. Bridging the gap between theoretical potential and practical application demands a focused endeavor to develop an adaptable framework. This framework should be flexible to accommodate the unique attributes of diverse language ecosystems, thereby enhancing runtime security across the spectrum of multi-language software and ensuring robust protection across varied language runtimes.

AI compiler and runtime security assurance. Much like traditional language compilers and runtimes catering to applications across various domains, the security of AI compilers and runtimes is gaining increasing importance within artificial intelligence and machine learning. As AI technologies advance and integrate into diverse applications, security assurance within AI compiler and runtime environments becomes a central concern, encompassing several crucial dimensions. Firstly, these compilers and runtimes often encounter sensitive information due to the nature of the data they handle. Consequently, establishing secure processing environments is paramount to avert unauthorized data exposure. Moreover, akin to conventional software, AI runtimes can exhibit vulnerabilities susceptible to exploitation

by malicious actors. Furthermore, the gamut of security threats extends to issues such as insecure execution or the injection of malicious models. These intricacies demand meticulous consideration within the architecture and operation of AI runtimes. As the landscape of AI technology evolves, secure compilers and runtimes will persist as a linchpin in constructing dependable and credible AI systems across various domains. As a significant stride within the scope of my long-term research, I am fully committed to investigating robust and feasible techniques that ensure the security of AI compilers and runtimes.

CHAPTER EIGHT

CONCLUSION

8.1 Summary

In my dissertation, I addressed the challenges associated with multi-language software vulnerabilities through a comprehensive approach. I conducted empirical investigations into vulnerability susceptibility, as evidenced by prior studies [84, 87, 85]. This research led to the development of PolyCruise [86], a novel dynamic cross-language information flow analysis technique. PolyCruise utilizes a unified intermediate representation and employs symbolic dependence analysis to achieve effective analysis of real-world multi-language software with large code bases. Notably, PolyCruise successfully identified vulnerabilities in open-source multi-language systems during its execution. However, as a dynamic approach, its effectiveness is influenced by input coverage limitations.

Recognizing the pivotal role of test input coverage, I introduced PolyFuzz [89], a comprehensive greybox fuzzing methodology. Through the incorporation of sensitivity analysis and whole-system coverage measurements, PolyFuzz generates powerful test inputs, thereby enhancing the identification of vulnerabilities.

Shifting the focus to runtime security, I developed PyRTFuzz [90], a two-level collaborative fuzzing framework designed for Python runtime. By combining generation- and mutation-based fuzzers, PyRTFuzz achieves comprehensive testing of Python runtime. Particularly, the use of a SLang-based approach ensures the generation of applications with diverse domains and complexities, thereby significantly contributing to fundamental software system safeguarding.

Looking forward, my plan involves strengthening cross-language security testing by integrating holistic fuzzing and comprehensive cross-language data flow facts. Furthermore, I will delve into compiler and language runtime testing. First, I intend to enhance collaborative fuzzing by merging program analysis and semantic mining techniques, and then

generalize these methodologies across diverse runtimes, encompassing platforms like JVM and JavaScript engines. Additionally, I am enthusiastic about exploring the intersection of machine learning frameworks and runtimes. This convergence between machine learning and security holds immense potential for pioneering research endeavors.

8.2 Implications

In summary, my Ph.D. work introduces generalizable methodologies that extend beyond specific instances, offering valuable insights and tools for researchers, industry professionals, developers, and security analysts in the broad domain of software security and testing.

8.2.1 Academic and industry impact

Researchers and academia. My study contributes empirical investigations into vulnerability susceptibility in multi-language software, shedding light on challenges and providing insights into dynamic cross-language information flow analysis. The development of PolyCruise, PolyFuzz, and the collaborative fuzzing framework PyRTFuzz establishes a solid foundation for advancing research in software security and testing methodologies.

Industry professionals. Practitioners in the software industry can gain practical approaches to identifying vulnerabilities in large code bases. The dynamic cross-language analysis technique in PolyCruise and the comprehensive greybox fuzzing methodology in PolyFuzz provide actionable insights for enhancing the security of multi-language systems.

8.2.2 Relevance to diverse researchers and developers

Developers and software engineers. The dissertation emphasizes the significance of input coverage in dynamic approaches like PolyCruise and addresses this concern with PolyFuzz, offering strategies to enhance vulnerability identification through improved testing methodologies. The collaboration of fuzzing techniques in PyRTFuzz for Python runtime security provides practical measures for safeguarding software systems during runtime.

Security analysts and testers. Stakeholders involved in security testing can gain valuable knowledge about challenges associated with multi-language software vulnerabilities. The presented methodologies, especially the collaborative fuzzing framework and the two-level approach in PyRTFuzz, offer practical tools for identifying and addressing security vulnerabilities in real-world scenarios.

8.2.3 Generalizable security solutions

My Ph.D. study generates a set of generalizable security solutions for multi-language applications and their respective language runtimes.

Holistic fuzzing approach. The integration of PolyFuzz and PyRTFuzz showcases a holistic fuzzing approach adaptable to various contexts. The developed methodologies can be generalized to enhance security testing in different multi-language software systems and runtime environments.

Cross-language data flow analysis. The use of a unified intermediate representation and symbolic dependence analysis in PolyCruise introduces a generalizable methodology for effective cross-language data flow analysis. This approach can be extended to other multi-language software systems with large code bases, providing a foundation for improving security analysis in diverse environments.

Collaborative fuzzing framework. The collaborative fuzzing framework developed for Python runtime in PyRTFuzz can serve as a generalizable methodology for testing and securing applications in different runtime environments. The two-level approach, combining generation- and mutation-based fuzzers, offers a comprehensive testing strategy adaptable to various programming languages and runtimes.

REFERENCES

[1] Mouna Abidi, Manel Grichi, and Foutse Khomh. "Behind the scenes: developers' perception of multi-language practices". In: *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. 2019, pp. 72–81.

[2] Mouna Abidi et al. "Are multi-language design smells fault-prone? An empirical study". In: *ACM Transactions on Software Engineering and Methodology* 30.3 (2021), pp. 1–56.

[3] Vitor Afonso et al. "Going Native: Using A Large-Scale Analysis of Android Apps to Create A Practical Native-Code Sandboxing Policy". In: *Network and Distributed System Security Symposium*. 2016, pp. 1–15.

[4] Ajax. *Pyo*. https://github.com/belangeo/pyo. 2020.

[5] Paul D Allison and Richard P Waterman. "Fixed–effects negative binomial regression models". In: *Sociological methodology* 32.1 (2002), pp. 247–265.

[6] Apache. *Apache Commons Compress*. https://github.com/apache/commons-compress. 2022.

[7] Steven Arzt, Tobias Kussmaul, and Eric Bodden. "Towards Cross-Platform Cross-Language Analysis with Soot". In: *State Of the Art in Program Analysis (SOAP)*. 2016, pp. 1–6.

[8] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *Network and Distributed System Security Symposium*. Vol. 19. 2019, pp. 1–15.

[9] aubio. *A library to label music and sounds*. https://github.com/aubio/aubio.git. 2019.

[10] Sora Bae, Sungho Lee, and Sukyoung Ryu. "Towards Understanding and Reasoning about Android Interoperations". In: *2019 IEEE/ACM 41st International Conference on Software Engineering*. 2019, pp. 223–233.

[11] Emery D Berger et al. "On the impact of programming languages on code quality: a reproduction study". In: *ACM Transactions on Programming Languages and Systems* 41.4 (2019), pp. 1–24.

[12] Guru Bhandari, Amara Naseer, and Leon Moonen. "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software". In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2021, pp. 30–39.

[13] David Binkley et al. "ORBS: Language-Independent Program Slicing". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 109–120.

[14] Tegawendé F Bissyandé et al. "Popularity, interoperability, and impact of programming languages in 100,000 open source projects". In: *2013 IEEE 37th annual computer software and applications conference*. 2013, pp. 303–312.

[15] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based greybox fuzzing as markov chain". In: *IEEE Transactions on Software Engineering* 45.5 (2017), pp. 489–506.

[16] Marcel Böhme et al. "Directed greybox fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2329–2344.

[17] Bounter. *Web*. https://tinyurl.com/4667pnkv. 2017.

[18] Fraser Brown et al. "Finding and Preventing Bugs in JavaScript Bindings". In: *2017 IEEE Symposium on Security and Privacy*. 2017, pp. 559–578.

[19] Achim D Brucker and Michael Herzberg. "On the Static Analysis of Hybrid Mobile Apps". In: *Engineering Secure Software and Systems*. 2016, pp. 72–88.

[20] Haipeng Cai. "Hybrid Program Dependence Approximation for Effective Dynamic Impact Prediction". In: *IEEE Transactions on Software Engineering* (2017).

[21] Haipeng Cai and Raul Santelices. "TracerJD: Generic trace-based dynamic dependence analysis with fine-grained logging". In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE. 2015, pp. 489–493.

[22] *categories of security vulnerabilities*. https://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf. 2020.

[23] Hongxu Chen et al. "Hawkeye: Towards a desired directed grey-box fuzzer". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2095–2108.

[24] Junjie Chen et al. "A survey of compiler testing". In: *ACM Computing Surveys* 53.1 (2020), pp. 1–36.

[25] Yaofei Chen et al. "An empirical study of programming language trends". In: *IEEE software* 22.3 (2005), pp. 72–79.

[26] Yuanliang Chen et al. "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers." In: *USENIX Security Symposium*. 2019, pp. 1967–1983.

[27] Yuting Chen, Ting Su, and Zhendong Su. "Deep differential testing of JVM implementations". In: *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE. 2019, pp. 1257–1268.

[28] Yuting Chen et al. "Coverage-directed differential testing of JVM implementations". In: *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 85–99.

[29] Jaeseung Choi et al. "Grey-box concolic testing on binary code". In: *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE. 2019, pp. 736–747.

[30] civetweb. *An embeddable web server with optional CGI, SSL and Lua support*. https://github.com/civetweb/civetweb. 2022.

[31] Code-Intelligence. *Coverage-guided, in-process fuzzing for the JVM*. https://github.com/CodeIntelligenceTesting/jazzer. 2022.

[32] Adam Cohen. "FuzzyWuzzy: Fuzzy string matching in python". In: *ChairNerd Blog* 22 (2011).

[33] PoC Consul and Felix Famoye. "Generalized Poisson regression model". In: *Communications in Statistics-Theory and Methods* 21.1 (1992), pp. 89–109.

[34] Juliet Corbin and Anselm Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[35] Chris Cummins et al. "Compiler fuzzing through deep learning". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, pp. 95–105.

[36] Cvxopt. *Web*. https://tinyurl.com/4k4v9646. 2016.

[37] Ali Davanian et al. "{DECAF++}: Elastic {Whole-System} dynamic taint analysis". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses*. 2019, pp. 31–45.

[38] dell. *A library to interface with the SMBIOS tables*. https://github.com/dell/libsmbios. 2007.

[39] Daniel P Delorey, Charles D Knutson, and Christophe Giraud-Carrier. "Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects". In: *Second International Workshop on Public Data about Software Development*. 2007.

[40] Isil Dillig et al. "Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs". In: 2011, pp. 567–577.

[41] Sung Ta Dinh et al. “Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases”. In: *Network and Distributed System Security Symposium*. 2021.

[42] Jeff Donahue et al. “Decaf: A deep convolutional activation feature for generic visual recognition”. In: *International conference on machine learning*. PMLR. 2014, pp. 647–655.

[43] Eclipse Cyclone DDS. *An open-source implementation of the OMG DDS specification*. https://github.com/eclipse-cyclonedds/cyclonedds. 2022.

[44] Jiahao Fan et al. “AC/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 508–512.

[45] Alessandro Fin, Franco Fummi, and Graziano Pravadelli. “Amleto: A multi-language environment for functional test generation”. In: *Proceedings International Test Conference*. IEEE. 2001, pp. 821–829.

[46] Andrea Fioraldi et al. “AFL++: Combining incremental steps of fuzzing research”. In: *14th USENIX Workshop on Offensive Technologies*. 2020.

[47] fusesource. *A java library for using ANSI escape codes to format the console output*. https://github.com/fusesource/jansi. 2021.

[48] Shuitao Gan et al. “{GREYONE}: Data Flow Sensitive Fuzzing”. In: *USENIX Security Symposium*. 2020, pp. 2577–2594.

[49] GitHub. *The 2020 State of the OCTO——VERSE*. https://octoverse.github.com/. 2020.

[50] google. *A Coverage-Guided, Native Python Fuzzer*. https://github.com/google/atheris. 2022.

[51] google. *A multi-language, cross-platform library of cryptographic APIs*. https://github.com/google/tink. 2021.

[52] google. *Continuous Fuzzing Framework for Open Source Software*. https://github.com/google/oss-fuzz. 2022.

[53] Manel Grichi et al. “On the impact of interlanguage dependencies in multilanguage systems empirical case study on java native interface applications (JNI)”. In: *IEEE Transactions on Reliability* 70.1 (2020), pp. 428–440.

[54] Alex Groce et al. “An extensible, regular-expression-based tool for multi-language mutant generation”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion*. IEEE. 2018, pp. 25–28.

[55] Samuel Groß. “Fuzzil: Coverage guided fuzzing for javascript engines”. In: *Department of Informatics, Karlsruhe Institute of Technology* (2018).

[56] gRPC. *gRPC Tutorial*. https://grpc.io/docs. 2020.

[57] Emre Güler et al. “Cupid: Automatic fuzzer selection for collaborative fuzzing”. In: *Annual Computer Security Applications Conference*. 2020, pp. 360–372.

[58] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. “CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines”. In: *Network and Distributed System Security Symposium*. 2019.

[59] Mary Jean Harrold, Gregg Rothermel, and Alex Orso. “Representation and analysis of software”. In: *Lecture Notes* (2005).

[60] Mostafa Hassan et al. “MaxSMT-based type inference for Python 3”. In: *Computer Aided Verification: 30th International Conference*. Springer. 2018, pp. 12–19.

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

[62] Fauna Herawati et al. “ANTIBIOTIC CONSUMPTION AT A PEDIATRIC WARD AT A PUBLIC HOSPITAL IN INDONESIA”. In: *Asian Journal of Pharmaceutical and Clinical Research* 12.8 (2019), pp. 64–67.

[63] Christian Holler, Kim Herzig, Andreas Zeller, et al. “Fuzzing with Code Fragments.” In: *USENIX Security Symposium*. 2012, pp. 445–458.

[64] Sungjae Hwang et al. “JUSTGen: Effective Test Generation for Unspecified JNI Behaviors on JVMs”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE. 2021, pp. 1708–1718.

[65] igraph. *A C library for creating, manipulating and analysing graphs*. https://github.com/igraph/igraph. 2022.

[66] Immutables. *Web*. http://immutables.github.io/.

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

[68] ISC. *A Classic, full-featured and mostly standards-compliant DNS*. https://gitlab.isc.org/isc-projects/bind9. 2022.

[69] javaparser. *A set of libraries implementing a Java 1.0 - Java 15 Parser*. https://github.com/javaparser/javaparser. 2022.

[70] jhy. *Java HTML Parser*. https://github.com/jhy/jsoup. 2022.

[71] jna. *Java Native Access*. https://github.com/java-native-access/jna. 2020.

[72] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. "A Large Scale Study of Multiple Programming Languages and Code Quality". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. 2016, pp. 563–573.

[73] Goh Kondoh and Tamiya Onodera. "Finding bugs in Java native interface programs". In: *Proceedings of the 2008 international symposium on Software testing and analysis*. 2008, pp. 109–118.

[74] Jacob Kreindl et al. "Multi-Language Dynamic Taint Analysis in a Polyglot Virtual Machine". In: *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*. 2020, pp. 15–29.

[75] Patrick Lam et al. "The Soot framework for Java program analysis: a retrospective". In: *Cetus Users and Compiler Infastructure Workshop*. 15 35. 2011.

[76] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization*. IEEE. 2004, pp. 75–86.

[77] Seongmin Lee et al. "Evaluating Lexical Approximation of Program Dependence". In: *Journal of Systems and Software* 160 (2020), p. 110459.

[78] Sungho Lee. "JNI program analysis with automatically extracted C semantic summary". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 448–451.

[79] Sungho Lee, Julian Dolby, and Sukyoung Ryu. "HybriDroid: static analysis framework for Android hybrid applications". In: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 2016, pp. 250–261.

[80] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. "Broadening Horizons of Multilingual Static Analysis: Semantic Summary Extraction from C Code for JNI Program Analysis". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 2020, pp. 127–137.

[81] Legrandin. *An self-contained Python package of low-level cryptographic primitives*. https://github.com/Legrandin/pycryptodome. 2018.

[82] Penghui Li et al. "On the Feasibility of Automated Built-in Function Modeling for PHP Symbolic Execution". In: *Proceedings of the Web Conference*. 2021, pp. 58–69.

[83] Siliang Li and Gang Tan. "Finding Bugs in Exceptional Situations of JNI Programs". In: *Proceedings of the 16th ACM conference on Computer and communications security*. 2009, pp. 442–452.

[84] Wen Li, Li Li, and Haipeng Cai. "On the vulnerability proneness of multilingual code". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 847–859.

[85] Wen Li, Li Li, and Haipeng Cai. "PolyFax: a toolkit for characterizing multi-language software". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 1662–1666.

[86] Wen Li et al. "{PolyCruise}: A {Cross-Language} Dynamic Information Flow Analysis". In: *USENIX Security Symposium*. 2022, pp. 2513–2530.

[87] Wen Li et al. "How are Multilingual Systems Constructed: Characterizing Language Use and Selection in Open-Source Multilingual Software". In: *ACM Transactions on Software Engineering and Methodology* (2023).

[88] Wen Li et al. "PCA: memory leak detection using partial call-path analysis". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1621–1625.

[89] Wen Li et al. "PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems". In: *USENIX Security Symposium*. 2023, pp. 1379–1396. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/li-wen.

[90] Wen Li et al. "PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 1645–1659.

[91] Wen Li et al. "Understanding Language Selection in Multi-Language Software Projects on GitHub". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings*. 2021, pp. 256–257.

[92] Jie Liang et al. "PATA: Fuzzing with Path Aware Taint Analysis". In: *IEEE Symposium on Security and Privacy*. 2022, pp. 154–170.

[93] Libgit2.org. *pygit*. https://tinyurl.com/2hpw3pwt. 2014.

[94] Baozheng Liu et al. "{FANS}: Fuzzing Android Native System Services via Automated Interface Analysis". In: *USENIX Security Symposium*. 2020, pp. 307–323.

[95] LLVM. *LibFuzzer: A library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html. 2020.

[96] luben. *JNI bindings for Zstd native library*. https://github.com/luben/zstd-jni. 2021.

[97] lukarao. *PyLLVM*. https://tinyurl.com/yckr397m. 2020.

[98] Chenyang Lyu et al. "{MOPT}: Optimized mutation scheduling for fuzzers". In: *USENIX Security Symposium*. 2019, pp. 1949–1966.

[99] M.Zalewski. *Technical "whitepaper" for afl-fuzz*. https://lcamtuf.coredump.cx/afl/technical_details.txt. 2014.

[100] Valentin JM Manes et al. "Fuzzing: Art, science, and engineering". In: *arXiv preprint arXiv:1812.00140* (2018).

[101] Bob Martin et al. "2011 CWE/SANS top 25 most dangerous software errors". In: (2011).

[102] Philip Mayer and Alexander Bauer. "An empirical analysis of the utilization of multiple programming languages in open source projects". In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. 2015, pp. 1–10.

[103] 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". In: *Journal of Software Engineering Research and Development* 5.1 (2017), pp. 1–33.

[104] Leo A Meyerovich and Ariel S Rabkin. "Empirical analysis of programming language adoption". In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 2013, pp. 1–18.

[105] Matthew B Miles, A Michael Huberman, and Johnny Saldaña. *Qualitative data analysis: A methods sourcebook*. Sage publications, 2018.

[106] Amir M Mir et al. "Type4Py: Practical deep similarity learning-based type inference for Python". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 2241–2252.

[107] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. "A Multilanguage Static Analysis of Python Programs with Native C Extensions". In: *Static Analysis Symposium*. 2021.

[108] mozilla. *An allowed-list-based HTML sanitizing library*. https://github.com/mozilla/bleach. 2022.

[109] msgpack. *An efficient binary serialization format*. https://github.com/msgpack/msgpack-python. 2021.

[110] ninia. *Java Embedded Python*. https://github.com/ninia/jep. 2018.

[111] Yu Nong and Haipeng Cai. "A preliminary study on open-source memory vulnerability detectors". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*. IEEE. 2020, pp. 557–561.

[112] NumPy.org. *NumPy*. https://github.com/numpy/. 2018.

[113] OK.ru. *A library for building high performance Java servers*. https://github.com/odnoklassniki/one-nio. 2020.

[114] Oracle. *JNI 6.0*. https://tinyurl.com/2p94tvhp. 2009.

[115] Sebastian Österlund et al. "Collabfuzz: A framework for collaborative fuzzing". In: *Proceedings of the 14th European Workshop on Systems Security*. 2021, pp. 1–7.

[116] OWASP. *A JSON encoder in Java*. https://github.com/OWASP/json-sanitizer. 2017.

[117] Shankara Pailoor, Andrew Aday, and Suman Jana. "Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation". In: *USENIX Security Symposium*. 2018, pp. 729–743.

[118] Jibesh Patra and Michael Pradel. "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data". In: *TU Darmstadt, Department of Computer Science, Tech. Rep.* (2016).

[119] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "T-Fuzz: fuzzing by program transformation". In: *2018 IEEE Symposium on Security and Privacy*. IEEE. 2018, pp. 697–710.

[120] Yun Peng et al. "Static inference meets deep learning: a hybrid type inference approach for python". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 2019–2030.

[121] Havoc Pennington. "D-Bus Tutorial". In: (2020).

[122] Raffaele Perego, Salvatore Orlando, and P Palmerini. "Enhancing the apriori algorithm for frequent set counting". In: *International Conference on Data Warehousing and Knowledge Discovery*. 2001, pp. 71–82.

[123] Henning Perl et al. "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 426–437.

[124] pillow. *Python Imaging Library*. https://github.com/python-pillow/Pillow. 2019.

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

[126] Psycopg. *Web*. https://tinyurl.com/27buex7n. 2018.

[127] pycurl. *A Python Interface To The cURL library*. https://github.com/pycurl/pycurl. 2022.

[128] pydata. *A a collection of fast NumPy array functions.* https://github.com/pydata/bottleneck. 2020.

[129] Pygame. *Web.* https://tinyurl.com/nhhdtj4f. 2020.

[130] pygments. *A generic syntax highlighter written in Python.* https://github.com/pygments/pygments. 2022.

[131] PyTables. *Web.* https://tinyurl.com/ywbr9jm3. 2019.

[132] Pytest. *Web.* https://tinyurl.com/45sajhuw. 2021.

[133] Python. *3.9.2 docs.* https://docs.python.org/3.9. 2020.

[134] Python. *CPython Repository.* https://github.com/python/cpython. 2022.

[135] Python. "Extending Python with C or C++". In: (2020).

[136] Python. *Python 3.8 Abstract Syntax Trees.* https://docs.python.org/3.8/library/ast.html. 2022.

[137] Python. *Unit test.* https://tinyurl.com/2fewd6ca. 2021.

[138] PyTorch. *Web.* https://tinyurl.com/5n8pfv9m. 2016.

[139] Foyzur Rahman and Premkumar Devanbu. "How, and why, process metrics are better". In: *2013 35th International Conference on Software Engineering.* IEEE. 2013, pp. 432–441.

[140] Sebastian Raschka. *Mlxtend: (machine learning extensions), a Python library of useful tools for the day-to-day data science tasks.* http://rasbt.github.io/mlxtend. 2020.

[141] Sanjay Rawat et al. "VUzzer: Application-aware Evolutionary Fuzzing." In: *Network and Distributed System Security Symposium.* Vol. 17. 2017, pp. 1–14.

[142] Baishakhi Ray et al. "A large scale study of programming languages and code quality in GitHub". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 2014, pp. 155–165.

[143] Jesse Ruderman. "Introducing jsfunfuzz". In: *http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz* 20 (2007), pp. 25–29.

[144] José Fragoso Santos et al. "Gillian, part i: a multi-language platform for symbolic execution". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2020, pp. 927–942.

[145] Hossain Shahriar and Mohammad Zulkernine. "Injecting comments to detect JavaScript code injection attacks". In: *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops.* IEEE. 2011, pp. 104–109.

[146] Qingkai Shi et al. "Pinpoint: Fast and precise sparse value flow analysis for million lines of code". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018, pp. 693–706.

[147] *SourceForge: The Complete Open-Source and Business Software Platform*. https://sourceforge.net. 2020.

[148] SQLAlchemy. *The Python SQL Toolkit and Object Relational Mapper*. https://github.com/sqlalchemy/sqlalchemy. 2022.

[149] Robert Swiecki. "Honggfuzz". In: *Available online at: http://code.google.com/p/honggfuzz* (2016).

[150] Gang Tan and Greg Morrisett. "ILEA: Inter-language analysis across Java and C". In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*. 2007, pp. 39–56.

[151] Federico Tomassetti and Marco Torchiano. "An empirical assessment of polyglot-ism in GitHub". In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. 2014, pp. 1–4.

[152] tytso. *The filesystem utilities for use with the ext2 filesystem*. https://github.com/tytso/e2fsprogs. 2022.

[153] ultrajson. *An ultra fast JSON encoder and decoder*. https://github.com/ultrajson/ultrajson. 2020.

[154] urllib3. *A HTTP client for Python*. https://github.com/urllib3/urllib3. 2022.

[155] Bogdan Vasilescu, Alexander Serebrenik, and Mark GJ van den Brand. "The Babel of software development: Linguistic diversity in Open Source". In: *International Conference on Social Informatics*. Springer. 2013, pp. 391–404.

[156] Vasudev Vikram, Rohan Padhye, and Koushik Sen. "Growing a Test Corpus with Bonsai Fuzzing". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering*. 2021.

[157] Jinghan Wang, Chengyu Song, and Heng Yin. "Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing". In: *Network and Distributed System Security Symposium*. 2021.

[158] Junjie Wang et al. "Skyfire: Data-driven seed generation for fuzzing". In: *2017 IEEE Symposium on Security and Privacy*. IEEE. 2017, pp. 579–594.

[159] Junjie Wang et al. "Superion: Grammar-aware greybox fuzzing". In: *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE. 2019, pp. 724–735.

[160] Fengguo Wei et al. "JN-SAF: Precise and Efficient NDK/JNI-Aware Inter-Language Static Analysis Framework for Security Vetting of Android Applications with Na-

tive Code". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 1137–1150.

[161] Zhaogui Xu et al. "Python predictive analysis for bug detection". In: *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 2016, pp. 121–132.

[162] Lei Xue et al. "NDroid: Toward tracking information flows across multiple Android contexts". In: *IEEE Transactions on Information Forensics and Security* 14.3 (2018), pp. 814–828.

[163] yaml. *A full-featured YAML processing framework for Python*. https://github.com/yaml/pyyaml. 2022.

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

[165] Haoran Yang, Wen Li, and Haipeng Cai. "Language-Agnostic Dynamic Analysis of Multilingual Code: Promises, Pitfalls, and Prospects". In: *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Ideas, Visions and Reflections*. 2022, pp. 1621–1626.

[166] Wei You et al. "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery". In: *2019 IEEE Symposium on Security and Privacy*. IEEE. 2019, pp. 769–786.

[167] Wei You et al. "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2139–2154.

[168] Tai Yue et al. "EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit". In: *USENIX Security Symposium*. 2020, pp. 2307–2324.

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

[170] Jie Zhang et al. "A Study of Programming Languages and Their Bug Resolution Characteristics". In: *IEEE Transactions on Software Engineering* (2019).

[171] Yunhui Zheng et al. "D2A: a dataset built for AI-based vulnerability detection methods using differential analysis". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*. IEEE. 2021, pp. 111–120.

[172] Yaqin Zhou and Asankhaya Sharma. "Automated identification of security issues from commit messages and bug reports". In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 2017, pp. 914–919.

[173] Peiyuan Zong et al. “Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning”. In: *USENIX Security Symposium*. 2020, pp. 2255–2269.

[174] zxing. *A multi-format 1D/2D barcode image processing library*. https://github.com/zxing/zxing. 2022.