Dissecting Real-World Cross-Language Bugs

HAORAN YANG, Washington State University, USA
HAIPENG CAI*, University at Buffalo, USA

Multilingual systems are prevalent and broadly impactful, but also complex due to the intricate interactions between the heterogeneous programming languages the systems are developed in. This complexity is further aggravated by the diversity of cross-language interoperability across different language combinations, resulting in additional, often stealthy *cross-language bugs*. Yet despite the growing number of tools aimed to discover cross-language bugs, a systematic understanding of such bugs is still lacking. To fill this gap, we conduct the first comprehensive study of cross-language bugs, characterizing them in 5 aspects including their *symptoms*, *locations*, *manifestation*, *root causes*, and *fixes*, as well as their *relationships*. Through careful identification and detailed analysis of 400 cross-language bugs in real-world multilingual projects classified from 54,356 relevant code commits in their GitHub repositories, we revealed not only bug characteristics of those five aspects but also *how they compare* between *two top language combinations* in the multilingual world (Python-C and Java-C). In addition to findings of the study as well as its enabling tools and datasets, we also provide practical recommendations regarding the prevention, detection, and patching of cross-language bugs.

CCS Concepts: • **Software and its engineering → Software defect analysis**.

Additional Key Words and Phrases: Multilingual code, cross-language bug, cross-language information flow

ACM Reference Format:
Haoran Yang and Haipeng Cai. 2025. Dissecting Real-World Cross-Language Bugs. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE058 (July 2025), 23 pages. https://doi.org/10.1145/3715777

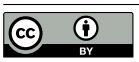
1 Introduction

Multilingual software development [49] (i.e., developing one software project with multiple programming languages that *interact*) is ubiquitous [9, 16, 37] and still on the rise [23, 30, 43, 44]. Thus, the vast and growing majority of real-world systems are *multilingual*, hence the paramount importance of assuring their quality [8]. While fostering a productive software process in various ways [1, 5] (e.g., combining the strengths of different languages [32, 45]), multilingual development also leads to great complexity of the resulting software [47]. This is not just the result of adding per-language complexities up, but particularly due to the intricate *interactions* between heterogeneous languages. The extra complexity tends to make multilingual software more buggy [2, 13, 20, 26, 28, 51].

Even worse, the multilingual world is diverse, both in terms of the large number and great variety of languages that can be used in tandem [16, 23] and concerning the varying ways in which the languages used may *interact* [21, 22, 29]. This diversity aggravates the complexity of multilingual software, explaining their bug-proneness, both qualitatively [31] and statistically [2, 20]. In fact, the bug-proneness often indicates actual/real bugs [13, 14, 17, 19, 24, 42], especially stealthy *bugs borne in faulty cross-language information flow induced by interactions between the used languages* (referred to as cross-language bugs).

*Haipeng Cai is the corresponding author.

Authors' Contact Information: Haoran Yang, Washington State University, Pullman, USA, haoran.yang2@wsu.edu; Haipeng Cai, University at Buffalo, Buffalo, USA, haipengc@buffalo.edu.



ACM 2994-970X/2025/7-ARTFSE058
https://doi.org/10.1145/3715777

Yet in contrast to the burgeoning tools each discovering only a few of them [10, 14, 19, 24, 25, 50, 51], cross-language bugs have not been comprehensively studied. Earlier relevant research does exist, which has examined the general characteristics of bugs induced by language use [52] and interoperation [3]. However, the extant works fall short in at least three folds. First, most current studies are concerned about *individual* languages each used separately from others, not really addressing *multilingual* software (i.e., with multiple languages used together with *interactions*) [4, 37, 52]. Second, studies that do address multilingual software are qualitative or statistical/probabilistic—they investigate bug-related human opinions (e.g.,developers' perception [1, 31]) or defect *proneness* (e.g., in terms of statistical associations between language combination and *likely* bug indicators [13, 17, 20]), not focusing on *factual* bugs in real-world multilingual code. Likewise, the most recent relevant works are either based on developers' discussions [48]—and the issues examined are not necessarily code-level bugs, or limited to high-level characteristics (e.g., open/reopen timing) of bugs in software using multiple languages [26–28]—the software is not necessarily multilingual per se and the bugs may not be induced by language interactions. Third, studies addressing language interoperability are closer to understanding cross-language bugs, yet they are mainly focused on cross-language API misuses [14, 42] rather than systematically investigating various aspects (e.g., symptoms, locations, manifestation, root cause, fixes, and their relationships) of those bugs.

In this paper, we conduct the first comprehensive study of real-world *cross-language* bugs (**CLB**s), focusing on those in two dominating language combinations [22] that have seen widespread presence and impact: Python-C (e.g., in machine-learning frameworks such as PyTorch/TensorFlow and NumPy/SciPy) and Java-C (e.g., in mobile/Web/database platforms such as Android, Tomcat, MySQL). For each language combination, we randomly sampled 200 CLBs, filtered and classified from a total of 865,680 commits in 2,428 repositories on GitHub. For each of the 400 confirmed CLBs, we then carefully examined the commit logs, the code changed and those impacted by the change, directly associated and indirectly linked bug/issue reports, and relevant official/external documentation. Both of the automated commit classification/labeling and manual CLB inspection are enabled by our novel static cross-language analyzer. The main goal of our study is to understand *where* and *why* CLBs occur as well as *how* to fix them, hence dissecting the *lifecycle* of CLBs. Accordingly, our study aims to answer the following questions on 5 aspects of CLBs:

- ***RQ1: What are the symptoms of CLBs?*** We identify how CLBs are externalized (as observed by developers or users) by examining the bug/issue reports.
- ***RQ2: Where do CLBs occur?*** We identify the location (in terms of structural/syntactic code contexts) where CLBs appear by forward tracking the cross-language information flow from the bug-fixing locations indicated in the associated CLB-fixing commit.
- ***RQ3: What are the manifestation characteristics of CLBs?*** We identify how CLBs are manifested, concerning the categories of functional semantics (e.g., memory access versus I/O) of the cross-language functions on the CLB's underlying information flow.
- ***RQ4: What are the root causes of CLBs?*** We identify the reasons why CLBs occur by backwardly tracking the cross-language information flow paths from the bug location (identified for RQ2) until the flow source (i.e., CLB origin).
- ***RQ5: How are CLBs fixed?*** We identify the CLB fixing strategies, immediately based on the bug fixes indicated by the associated CLB-fixing commit.

For each RQ, we seek answers both for each language combination and *between the two combinations (concerning the similarities and differences)*. Moreover, we examine the *lifecycle relationships between the 5 aspects* of CLBs via association analysis across the answers to these RQs.

Among a number of other novel findings, our study revealed that: **(1)** CLBs exhibit three primary symptoms: incorrect results/outputs, error/warning messages, and crashes/aborts, which are

common to both Python-C and Java-C integrations. Yet, performance bottlenecks were notably more prevalent in Java-C code, while memory leaks were more significant in Python-C code due to reference count misuse in cross-language memory management semantics; **(2)** CLBs commonly occur at assignments and function calls in both Python-C and Java-C code. However, they frequently manifested at cross-language API call sites in Python-C code and at conditional statements involving foreign data in Java-C code; **(3)** the studied CLBs were mostly manifested in code semantics related to foreign data parsing, foreign object handling, cross-language (data) conversion, and memory management (particularly reference counting) across the two languages; **(4)** the two primary root causes of CLBs were cross-language logic errors and boundary conditional errors, with the former far dominating over any other categories; **(5)** the primary CLB fixing strategies involved adding missing (e.g., boundary) checks and using the correct data types across languages. These strategies effectively addressed the prevalent logic and boundary errors in both language combinations. Finally, **(6)** the CLB symptom of memory leaks is strongly associated with the CLB root cause of reference count misuse, which is further strongly associated with the manifestation semantics of cross-language memory management; the symptom of incorrect result/output is strongly associated with CLB locations of return statements and conditional expressions.

Contributions. In summary, this work makes the following main contributions:

- To the best of our knowledge, this is the first comprehensive study of real-world factual/confirmed CLBs that *systematically examines their lifecycle aspects* (bug locations, root causes, manifestation, symptoms, fixing strategies), while *addressing two dominant language combinations*, Python-C and Java-C, including their *similarities and differences in each aspect*. Our novel empirical findings can help researchers and developers gain a systematic and in-depth understanding of such stealthy bugs hence guide their countermeasures. Based on these findings, we also offer actionable insights/recommendations on how to avoid, detect, test/debug, and repair CLBs, which can inform design of practical tooling support of respective capabilities.
- We contribute the first high-quality (i.e., accurate and diverse) dataset of CLBs and an automated tool (a novel static cross-language analyzer) to collect/analyze the dataset, which can assist with curating more of them. This dataset/tool can be used to benefit relevant future research.

2 Methodology

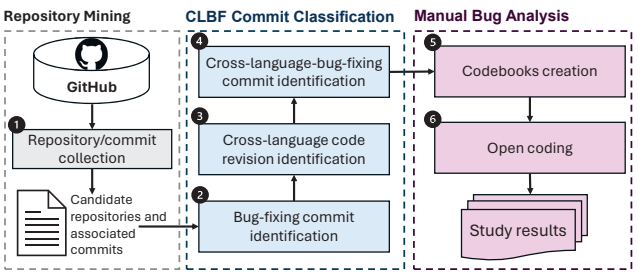


Fig. 1. Overview of our study methodology (process).

Our study is governed by the overall process outlined in Figure 1. Given the lack of existing datasets on CLBs, we start with CLB collection via repository mining (targeting Python-C and Java-C projects) on GitHub, the most popular source of open-source (multilingual) software [22]. Then, based on the resulting list of *candidate repositories* and their *associated commits*, we aim to curate CLBs from potential cross-language-bug-fixing (**CLBF**) commits identified using our CLBF commit classification (**CCC**) tool. CCC works in three steps: (1) *bug-fixing commit identification* to produce any commit aimed to fix a bug, (2) *cross-language code revision identification* to ensure the revision corresponding to a bug-fixing commit includes cross-language interactions, and (3) *cross-language-bug-fixing commit identification* to determine if the commit is intended to fix a CLB. Finally, guided by our RQs, we perform manual bug analysis to confirm and analyze CLBs to obtain *study results*, following an *open coding* process after *codebooks creation*.

2.1 Repository Mining

Given the vast diversity of the multilingual world [23, 37], it is infeasible to cover many language combinations while aiming at a comprehensive and in-depth bug study in one paper. We start with Python-C and Java-C as they are the most popular and impactful language combinations [22, 49] where most of existing CLBs are reported [14, 18–20, 24]. For each candidate project, we use the GitHub API [11] to retrieve its set of languages used and the code size of each language unit. Based on this information, we collect projects per two criteria summarized and justified below:

- 50% or more of the project is written in Python/Java and C combined in terms of code size (SLOC)—such projects can be considered Python-C or Java-C software per earlier studies [14, 19, 24].
- The project's repository has n stars. While not a perfect indicator of the true quality of a project [7], #stars has been widely used as a popularity indicator to select software projects in many prior studies [6, 20, 22, 23, 34, 37, 38, 41]. Projects with a larger n tend to gain more attention, thus the bugs in them likely have broader impacts. Meanwhile, a larger n implies fewer projects to be collected. However, a greater number and diversity of projects would make our study findings more representative. With this trade-off in mind and per our preliminary repository mining outcome, we chose $n = 50$ to strike a good balance between the two competing concerns.

With these criteria, we obtained 4,185 (2,428 Python-C and 1,757 Java-C) repositories. Then, considering their entire history, we mined a total of 886,526 commits associated with these repositories.

2.2 CLBF Commit Classification (CCC)

Not all the projects and commits obtained in §2.1 will be relevant for our study. Relevant are CLBF commits each meeting 3 requirements outlined and justified below:

- **R1**: The commit is associated with (i.e., aimed for) *fixing a bug*—because we intend to retrieve bug cases from such commits.
- **R2**: The commit is made to a project revision (version) that is multilingual (i.e., having interactions between its programming languages)—the commit can be a CLBF commit only if it aims to fix a bug in truly multilingual software.
- **R3**: The bug to be fixed by the commit involves interactions between languages (i.e., cross-language behaviors)—we study CLBs which are induced by faulty cross-language behaviors.

Our CCC tool has three major steps, labeled as ②, ③, and ④ in Figure 1, to check a commit against R1, R2, and R3, respectively, as elaborated next. CCC follows this specific order of these steps because of their descending levels of overhead—② is the fastest and ④ is the most expensive. Thus, the order enables filtering out more irrelevant commits earlier and letting the fewest candidate CLBF commits enter the most costly step, hence the lowest overall cost.

2.2.1 Bug-Fixing Commit Identification. A CLBF commit must be a bug-fixing commit in the first place. Thus, we first check the commits mined so far against R1. For each commit, we start by looking for its associated issue (id) and dismiss those for which we can not find any. Otherwise, we further use a set of bug related keywords to check against the issue tags, including 'error', 'bug,' 'defect,' 'patch', 'mistake', 'fault', 'failure', 'fix', 'issue', 'incorrect', and 'flaw'. Then, a commit is considered a bug-fixing commit if its associated issue's tag name contains any of these keywords. Developers do not always associate a bug-fixing commit with an issue, though. Thus, CCC may miss some true positives here. However, our approach is precise (e.g., more than those based on matching commit messages against such keywords [36]). We prioritize precision over recall in our study as justified by our goal of studying cross-language bugs (that are actually so). By the end of this step, we obtained 54,356 bug-fixing commits.

2.2.2 Cross-Language Code Revision Identification. Now we check each bug-fixing commit against R2: the underlying bug may not be a CLB unless the commit is made to a project revision/version that is indeed multilingual (i.e., with cross-language interactions). Moreover, checking such interactions at the project level may not suffice because the language use and multilingual nature of a project may change across revisions [20, 23], as we indeed encountered.

We developed an automaton-based module in CCC to detect whether a project revision has language interactions via common Python-C or Java-C FFIs (e.g., `Python C Extension` and `JNI`), by extending PolyFax [21], a language interfacing mechanism detector. The automaton transitions between states based on the presence of respective FFI's characteristics within the code. The automaton's transition to its final state indicates the use of a specific FFI, thereby confirming that the code revision involves multiple interacting languages. After this step, we have 25,120 commits.

2.2.3 Cross-Language-Bug-Fixing Commit Identification. For each commit (that satisfies R1 and R2) to be further associated with a *cross-language* bug, the changed code should be part of the cross-language behaviors of the revision. Such behaviors can be modeled by cross-language information flows. Moreover, in an FFI-based Python-C or Java-C revision, language interactions are realized via cross-language APIs. Thus, a bug-fixing commit in a multilingual project (revision) is a CLBF commit if at least one of the changed code lines has a (transitive) dependence (backward or forward) relationship with code entities of the other language via at least one of the cross-language APIs.

However, a static information flow/dependence analysis tool widely applicable to real-world Python-C or Java-C programs is not available [51]. The closest available one is Joern [46], which supports intraprocedural control flow analysis for Python, Java, and C. Based on these capabilities, we extended Joern to build a whole-program cross-language interprocedural control flow graph (**CICFG**) for a given project (revision). A CICFG is directed graph where each node represents a statement in any of two language units and each edge represents control flow between two nodes.

To construct the CICFG, CCC first obtains control flow graph (CFG) of each method in each language unit as Joern immediately provides. Next, CCC builds the intra-language call graph for each language unit by traversing per-method CFGs of that unit, then builds cross-language call graph from the two intra-language call graphs and call edges across the two languages. To identify the cross-language call edges, CCC uses calling-relation rules defined per the respective FFI (e.g., Python C Extension or JNI) according to their official specifications (e.g., cross-language function naming/calling conventions). Finally, based on the cross-language call graph and per-method CFGs, CCC builds the CICFG following classical ICFG construction approach [40]. To facilitate forward/backward traversal on it, CICFG is annotated with function names and line numbers.

CCC checks if a commit is CLBF as described earlier, but using CICFG to approximate cross-language dependencies (information flow). The resulting imprecision is resolved by our final confirmation during the manual bug analysis. The CICFG may also be incomplete (e.g., due to the lack of pointer/reference analysis in construction), which is acceptable for our bug collection purposes. After this step of CCC, we have 5,941 (likely CLBF) commits.

2.3 Manual Bug Analysis

To answer our RQs, we manually examine the CLBs associated with the confirmed, associated CLBF commits via case studies. The answers would involve terms/phrases that we had no full prior knowledge about. Thus, we adopted an open coding approach for each RQ, first deriving a codebook and then applying it to characterize each bug in each of the 5 aspects—both by the authors following a common inter-agreement and consensus procedure.

Random sampling for codebook. To create the codebooks, we first randomly sampled 364 out of the 5,941 commits for analysis. This sample size is statistically significant at 95% confidence level (CL) and 5% margin of error (ME) with respect to the population (i.e., the 5,941 commits).

Derive codebooks. We need a codebook for each RQ. To that end, each author independently curated an initial catalog (e.g., of root cause) based on the 364 commits. Subsequently, they reconciled any differences until agreeing upon a finalized codebook. In particular, each participant (1) meticulously reviewed the commits, (2) evaluated whether each commit fell within the existing categories, and (3) forged a fresh category if necessary. When formulating a new category, the initial step involved defining a label to encapsulate the bug aspect (e.g., root cause) presented in the commit. Following this, we crafted detailed descriptions for this new category, and identified that bug aspect that would appropriately fall under this category. The commit was then incorporated into the codebook, serving as an illustrative example for this new category.

Random sampling for study. Given the great time cost of manually examining even a single commit, we randomly sampled commits (other than the 364 for creating per-RQ codebooks) until we had 200 Python-C and 200 Java-C CLB cases, 400 *confirmed* CLBs in total for the bug analysis. This sample size is statistically significant at 95% CL and 5% ME.

As during the sampling for codebook creation, the confirmation of a potential CLBF commit (identified by CCC) as corresponding to a true CLB is subsumed in the sampling here too. The confirmation is to verify that the underlying CICFG path actually bears relevant information flow.

Coding process. For each RQ, we applied the respective codebooks to answer it against each of the 400 CLBs. To ensure the reliability of this process, we utilized a negotiated agreement approach, which is often adopted when the main goal is to generate novel insights [33], as needed in study.

- For RQ1, we identify CLB symptoms per the issue/bug reports and any related documents they link to, developer-user discussions, bug-inducing code, and expected outputs, etc.
- For RQ2, we start with a forward traversal on the CICFG from the bug-fixing location, carefully examining the information flow (data/control dependencies), to pinpoint the bug location and name it per the respective codebook.
- For RQ3, starting from the bug location, a backward traversal on the CICFG is conducted until reaching the bug's source/origin in the other language. We identify cross-language (e.g., foreign and native) functions encountered during the traversal and classify the semantics of each as per its code (along with any comments and project documentation), language interfacing definitions, and the official specifications of foreign functions.
- For RQ4, a deeper examination of the backward cross-language information flows from the bug location back to its source/origin is carried out to identify the bug's root cause, which is named per the respective codebook.
- For RQ5, the code changes in the associated CLBF commit are examined. The fixing strategy is summarized based on the semantics impact of the bug-fixing changes on the bug location.

3 Study Results

3.1 RQ1: Symptoms of CLBs

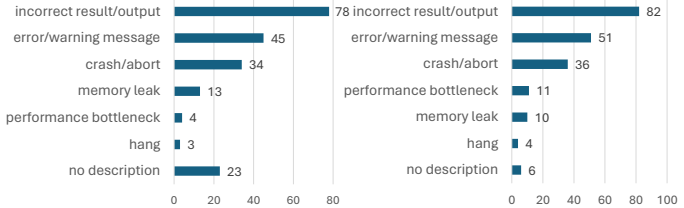


Fig. 2. Distribution of symptoms (vertical axes) of CLBs (counted on horizontal axes) in Python-C (left) and Java-C (right) projects.

As Figure 2 shows, we found six types of CLB symptoms. 'Incorrect result/output' is the predominant symptom, while 'hang' and 'performance bottleneck' are the least common. The top 3 symptoms are as follows:

Sym1: Incorrect result/output. Incorrect results/outputs refer to situations where a program fails to produce the intended outcome or behaves unexpectedly due to issues in the cross-language information flow, often in response to user inputs. As shown in Figure 3, this issue report describes the symptom (**Sym**) where cross-language code does not utilize the expected 16-bit color depth for drawing on an image, instead using only the lower 8-bits. This bug leads to a discrepancy between the expected and actual color output.

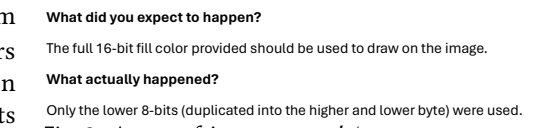


Fig. 3. A case of *incorrect result/output* symptom.

Sym2: Error/warning messages. The program provides explicit notifications that it encounters a situation it cannot handle or when an operation fails. As shown in Figure 4, the issue report depicts the symptom where an error occurred because the function expected a string type for the filename, but a Path object was provided instead. This bug leads to an error message arising.

Fig. 4. A case of *error/warning message* symptom.

Sym3: Crash/abort. A program crash/exit refers to an unexpected termination of the program during its execution, which prevents it from completing its intended tasks. As shown in Figure 5, the Python program experiences a crash specifically when using the `psutil.Popen()` function on the FreeBSD 12 operating system. This bug causes the Python interpreter to terminate unexpectedly.

While Sym2 may seem to be subsumed by Sym3 (e.g., the messages are collected from crashes), we treat them separately as they represent distinct aspects of bug manifestation, and understanding both provides a more comprehensive picture of how CLBs surface in practice. In particular, (1) error/warning messages (Sym2) are diagnostic outputs that indicate potential issues without necessarily leading to program termination—thus they are not always collected from crashes/aborts. In contrast, Sym3 (crash/abort) represents cases where the program terminates unexpectedly, which is typically a more severe outcome requiring immediate attention. Also, (2) while Sym2 may precede Sym3, this is not always the case. For example, certain CLBs, such as memory corruption, can directly result in a crash without producing any warning messages. Conversely, many error messages are indicative of recoverable states or issues that do not escalate into crashes. Thus, (3) by analyzing Sym2 and Sym3 separately, we capture the spectrum of bug symptoms more comprehensively: including diagnostic signals (Sym2) that may aid in early detection to severe consequences (Sym3) that highlight system vulnerabilities. This separation also allows us to study how error reporting mechanisms vary across languages and systems, a key characteristic of cross-language interactions.

Between Python-C and Java-C CLBs, it is evident that the top three symptoms are common to them: Sym1, Sym2, and Sym3, in that order. This trio of symptoms not only tops the list but also collectively constitutes over 75% of the reported issues in each language combination, indicating a significant prevalence.

Fig. 5. A case of *crash* symptom.

Memory leaks appear to be more problematic in Python-C projects compared to Java-C. This is related to differences in garbage collection mechanisms between Java and Python. In Python-C projects, memory leaks can be particularly problematic due to the way Python's garbage collection interacts with manually managed memory in C. Also, performance bottlenecks are notably more significant in Java-C projects. The JNI requires frequent context switches between the JVM and native C environments, leading to performance bottlenecks due to overheads in data marshaling,

memory management synchronization, and error handling. In both Python-C and Java-C projects, hangs are relatively uncommon compared to other symptoms. This rarity suggests that both language combinations tend to effectively manage scenarios that typically lead to hangs, such as resource deadlock or inefficient loop conditions in cross-language interoperations.

The CLBs exhibit three top symptoms: incorrect result/output, error/warning messages and crash/abort, which are common to Python-C and Java-C. Yet performance bottlenecks are notably more prevalent in Java-C code and memory leaks are more significant in Python-C code.

3.2 RQ2: Locations of CLBs

The studied CLBs were mainly introduced at 9 locations (**Loc**s) within a function shown in Figure 6. The following are the top 3 CLB locations within Python-C and Java-C projects:

Loc1: Assignment. The CLB happens when a foreign object is assigned to a local variable. In these cases, the assignments are usually used to either map cross-language data or convert data formats between two languages. The goal of such assignments is often to ensure interoperability and correct data handling. The assignment statement is the location where CLBs occur most frequently in Python-C and the second most frequent location for CLBs in Java-C.

Bug location patterns in Python-C and Java-C assignments exhibit similar characteristics, including: (1) *Initialization for foreign data type interaction*, which is critical for setting up variables involved in cross-language operations. Errors at this stage can propagate, causing downstream issues. (2) *Calculation with foreign data types*, which involves assigning computed values to foreign variables or vice versa. Errors in this process can lead to computational inaccuracies, type mismatches, or memory corruption. (3) *Foreign function return value assignment*, which captures the output from functions written in a foreign language. Proper handling of these return values is essential to ensure their correct interpretation and usage in the calling language.

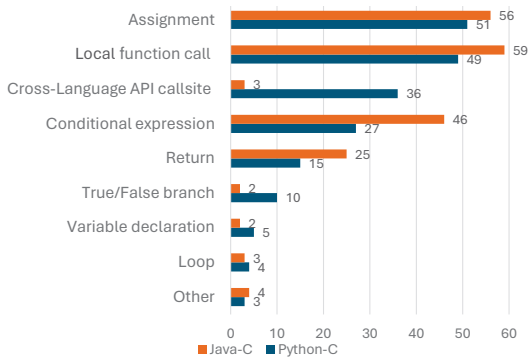


Fig. 6. Distribution of CLB locations.

Loc2: Local function call. The CLB occurred at a callsite targeting a local function, but the callee is not a cross-language function/API—although the function call is on the cross-language information flow responsible for the CLB. Typically, the callee is misused or itself faulty. Local function calls primarily involve invoking functions to process data from a different language. Such callsites are second most frequent CLB location in Python-C and the most frequent in Java-C. In both Java-C and Python-C contexts, foreign objects are rarely directly used as arguments in local function calls. Instead, they are typically converted into local objects or processed through some computation before being passed to the function. This conversion ensures compatibility with the function's expected argument types and leverages the local language's type safety mechanisms.

Loc3: Cross-Language API callsite. The CLB was introduced right at a cross-language API callsite. First, at the cross-language API callsite, data types from foreign languages are converted into native types through FFI (e.g., convert `PyLong` to long via `PyLong_AsLong`). Second, high-level languages, such as Java or Python, typically offer richer standard libraries, as well as support for advanced features like garbage collection. To maintain consistency with high-level languages, the C language invokes corresponding APIs, such as using `Py_DecRef()` to manage reference count. The cross-language API callsite is the location where CLBs occur third most frequently in Python-C.

There is a notable difference in return value handling between Python-C and Java-C. In Python-C, some API calls return values (e.g., `PyInt_AsLong`, shown in Figure 12), while others do not (e.g., `Py_INCREF`, shown in Figure 13). Most JNI functions do not return values, as they often perform operations that manipulate memory or state directly through their arguments. Therefore, in Python-C, the usage of arguments at cross-language API callsites is similar to that of local function calls, typically involving the conversion of foreign objects to local variables. In contrast, Java-C often requires the direct involvement of foreign objects as arguments in these API calls.

Loc4: Conditional statement. In these cases, the CLB occurred in a conditional expression (e.g., a predicate) at a conditional statement (e.g., if). These conditional statements are typically used to manage decision-making processes that involve cross-language information flow. The goal of such conditional expressions is often to ensure the integrity of cross-language data flow and avoid logic errors. The conditional statement is the location where CLBs occur third most frequently in Java-C.

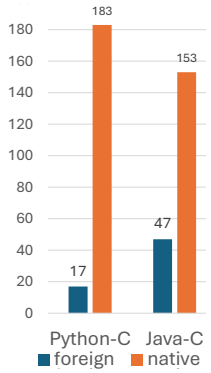


Fig. 7. Distribution of CLBs across language units.

In Python-C and Java-C projects, a common pattern is omitting critical conditions within conditional statements. This oversight often leads to the processing of invalid or unexpected data, as vital checks are neglected. For instance, overlooking NULL pointer checks or other crucial state verifications can precipitate system crashes or generate erroneous outputs. Notably, Python-C projects exhibit a higher incidence of operator errors compared to Java-C projects. This disparity largely stems from the fundamental differences in operator handling and type conversion between Python and C. Python's dynamic typing and flexible operator usage can cause confusion when translated to C's stricter and more explicit syntax. This misalignment increases the likelihood of using incorrect operators, leading to logical errors.

As shown in Figure 6, assignment and function call are the dominating locations where CLBs occurred. Among the CLB locations, the cross-language API call site shows significant quantitative disparity between the Python-C and Java-C projects. Within the Python-C context, this location is ranked third (out of ten), whereas Java-C is ranked to ninth (out of ten).

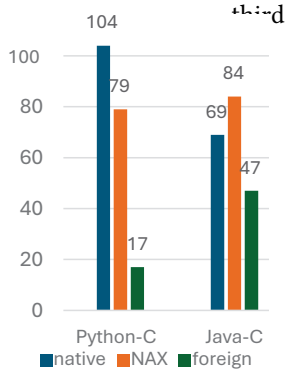


Fig. 8. Distribution of CLB locations within 3 kinds of cross-language functions.

Figure 7 reveals a predominance of CLBs in native code for both Python-C and Java-C projects, indicating hightened complexity in native language components. This distribution stems from two primary factors. First, native code manages intricate data translations between languages with disparate type systems and memory models, necessitating complex conversions and integrity checks that increase CLB risk. Second, native components act as critical bridges between languages, implementing error-prone interfaces that require low-level system calls and meticulous resource management. These findings underscore the challenges inherent in cross-language integration, particularly in data handling and interface implementation, where the native language bears the brunt of synchronizing divergent programming paradigms.

Figure 8 shows that CLBs primarily occur in native functions within Python-C and non-API cross-language (NAX) in Java-C. A NAX function is defined as a helper function that supports the operation of cross-language APIs. These functions facilitate the underlying mechanisms necessary for the API's cross-language communication and integration. They also assist in the preparation, conversion, or management of data to ensure compatibility and efficient interaction between languages. For example, a NAX function might involve a function in C that is directly called by Java through JNI, but this function is not part of any public API, and it is typically used internally by the application for specific cross-language tasks

```
c1- if (!PyArg_ParseTuple(args, "O&O&O&O&iidi",
            NI_ObjectToInputArray, &input,
                        NI_ObjectToOptionalInputArray, &zoom,
                        NI_ObjectToOptionalInputArray, &shift,
                        NI_ObjectToOutputArray, &output,
                        &order, &mode, &cval, &nprepad))
c2+ if (!PyArg_ParseTuple(args, "O&O&O&O&iidi",
                        NI_ObjectToInputArray, &input,
                        NI_ObjectToOptionalInputArray, &zoom,
                        NI_ObjectToOptionalInputArray, &shift,
                        NI_ObjectToOutputArray, &output,
                        &order, &mode, &cval, &nprepad, &grid_mode))
c3    goto exit;
```

Fig. 9. A Python-C case exhibiting the CLB manifestation semantics of cross-language data parsing.

```
c1 jint fd = (*env)->GetIntField(env, jfd, fdClassDescriptorFieldID);
c2- FILE *file = fdopen(dup(fd), "rb");
c3+ int dupFd = dup(fd);
c4+ FILE *file = fdopen(dupFd, "rb");
...
c5- return createGifHandle(GifFileIn, Error, ftell(file), fileRewind,
                           env, justDecodeMetaData);
c6+ struct stat st;
c7+ if (fstat(dupFd, &st) != 0)
c8+   st.st_size = -1;
c9+ return createGifHandle(GifFileIn, Error, ftell(file), fileRewind,
                           env, justDecodeMetaData, st.st_size);
```

Fig. 10. A Java-C case exhibiting the CLB manifestation semantics of cross-language object handling.

(e.g., handling low-level memory or resource management). While these functions are not exposed as formal APIs, they still operate in cross-language contexts. Here we highlight the importance of NAX functions as they are a critical but often overlooked source of bugs. These functions complicate the debugging process because they are not part of the formal API and might not be well-documented or exposed in typical error reporting mechanisms. As shown in Figure 8, native functions in both Python-C and Java-C projects exhibit a large number of CLBs, potentially due to the complexity these functions often entail. The presence of a substantial number of CLBs in NAX functions suggests challenges in managing cross-language operations that do not involve direct API calls but still require language interfacing. This could involve data marshaling, memory management, or concurrency controls, which are inherently complex and prone to CLBs. Foreign functions have the fewest CLBs in both project types, which indicates that they are less susceptible to cross-language errors.

CLBs occur at two main locations, assignment and function call on cross-language information flows, that are common to Python-C and Java-C projects. Differently, CLBs frequently occur at cross-language API call sites in Python-C code, versus at conditional expressions in Java-C code. There is a significant challenge in managing CLBs in the native language.

3.3 RQ3: Manifestation Characteristics

To understand how CLBs are manifested, we mainly examined the manifestation characteristic (**MC**), specifically the type of cross-language function/API involved in terms of its functional semantics. Due to the various interfacing mechanisms used between Python-C and Java-C projects, the naming rule for the cross-language API's semantics category also varies accordingly. As shown in Figure 11, there are 10 distinct cross-language API types that appear on the information flows underlying respective Python-C and Java-C bugs. We focus on the 4 most prevalent categories for the two language combinations studied, as the other categories are minor.

MC1: Cross-language data parsing. Parsing arguments and building values involves translating strings into data structures according to specific rules or syntax. In our study, many CLBs arose from inconsistencies in how cross-language APIs parsed data from foreign languages or converted these into formats understandable by the native language. This issue is predominant in Python-C projects but absent in Java-C ones. Figure 9 illustrates the use of the `PyArg_ParseTuple` function in Python-C to interpret parameters passed to C, highlighting a critical semantics facet of Python-C interfacing that facilitates data exchange and manipulation across languages.

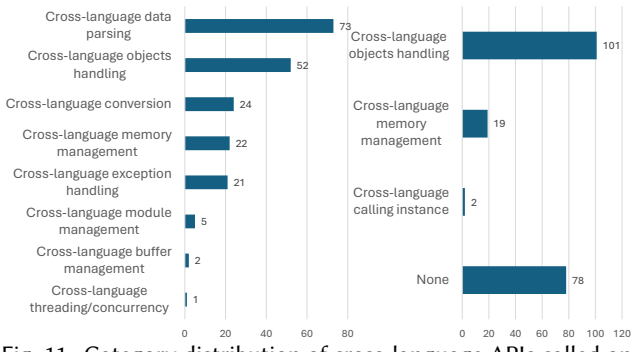


Fig. 11. Category distribution of cross-language APIs called on the cross-language information flow paths underlying CLBs in Python-C (left) and Java-C (right) projects.

MC2: Cross-language object handling. This category includes those aimed at manipulating (e.g., creating, accessing, and managing) foreign (i.e., Python and Java) objects. These CLBs involve incorrect object creation, misuse of object methods, or issues with object lifetime management. Our result points to difficulties hence bug-proneness in managing objects across languages. This semantics category constitutes nearly half of the cases in Java-C projects and also ranks as the second most prevalent in Python-C projects. As shown in Figure 10, the JNI function is designed to open a file descriptor from a Java object and manipulate it in native code. The foreign function `GetIntField` (at line c1) extracts the integer value of the descriptor field, which demonstrates object field manipulation across languages. Then the native part of the function proceeds to duplicate the file descriptor (`fd`) obtained from the Java object and associates it with a file stream for reading.

MC3: Cross-language conversion. This semantics category involves the translation of data types and structures between different programming languages. It ensures that the data exchanged between languages maintains its integrity and relevance. In our study, CLBs related to this semantics frequently emerged from challenges associated with correctly transforming data from one language's data type system to another's. In Python-C projects, this semantics category ranks third, but it is missing within Java-C projects. Figure 12 shows that the original implementation employs the Python API `PyInt_AsLong` to convert a Python object `tmp` directly into a long integer in C. This method assumes that the Python object strictly represents an integer. However, if `tmp` is a floating-point number, using `PyInt_AsLong` leads to incorrect results.

```
c1 if (tmp == NULL) {
c2   return -1;}
c3- seconds_offset = PyInt_AsLong(tmp);
c4+ seconds_offset = (int)PyFloat_AsDouble(tmp);
```

Fig. 12. A Python-C case exhibiting the CLB semantics of cross-language conversion.

MC4: Cross-language memory management. Reference counting is a fundamental aspect of memory management in Python and Java. It helps ensure that memory is efficiently allocated and deallocated. The CLBs underlain by the cross-language APIs of this category often arise from improper increments or decrements of reference counts, leading to memory leaks or premature disposal of objects. The #cases within this semantics category do not vary significantly between Python-C and Java-C projects. As shown in Figure 13, the initial implementation risked such issues by potentially over-incrementing the reference count of dtype->singleton without corresponding decrements. Incorrect handling of reference counts can lead to memory leaks or dangling pointers.

In Python-C projects, prevalent cross-language data parsing issues arise due to the intrinsic mechanisms of data transfer between Python and C. Python frequently utilizes the CPython API for functions like `PyArg_ParseTuple`, which breaks down Python arguments (`args` and `kwargs`) into C-compatible types. This is necessitated by Python's dynamic typing and high-level structures lacking direct C equivalents, requiring explicit parsing and validation at language boundaries. Conversely, Java-C interactions often employ JNI for direct object handling, facilitating straightforward access to Java objects and method calls from C without needing additional type conversions.

```
c1  if (dtype == NULL) {
c2      goto fail;
c3  }
c4- Py_INCREF(dtype->singleton);
c5  otype = dtype->singleton;
c6+ Py_INCREF(otype);
c7+ Py_DECREF(dtype);
```

Fig. 13. A Python-C case exhibiting the CLB location of cross-Language API callsite.

The higher incidence of conversion-related bugs in Python-C can be attributed to Python's dynamic nature, which requires constant runtime type checks and conversions when interfacing with the statically typed C. In contrast, Java's static typing aligns more closely with C, allowing JNI to manage type conversions seamlessly, reducing the necessity for external cross-language conversion functions. This alignment results in many Java-C interactions being categorized under 'None', indicating a lack of specific cross-language function/API usage, reflecting JNI's inherent capability to handle type conversions and data passing effectively.

Moreover, the frequent use of NAX functions in Python-C, alongside native and foreign functions, suggests complex interactions that necessitate bespoke handling, often involving specific data

```
py1 import iteration_utilities
py2 minmax = iteration_utilities.minmax
py3 ...
py4 minmax([T((T(1), T(5)))],key=lambda x: x.value[0])
-----------------------------------------------------
c1  static PyObject*
c2  PyIU_MinMax(...,PyObject *args,PyObject *kwargs){
c3    ...
c4    if (!PyArg_ParseTupleAndKeywords(PyIU_global_0tuple,
      kwargs, ..., kwlist, &keyfunc, &defaultitem))
c5    ...
c6    if (keyfunc != NULL) {
c7-     funcargs = PyTuple_New(0);
c8+     funcargs = PyTuple_New(1);
c9  ...
```

Control Flow
Data Flow

Fig. 15. A Python-C CLB case rooted in 'logic error' [39].

```
java1  return getCurrentPosition(gifInfoPtr);
-----------------------------------------------------------------------
c1  Java_pl_droidsonroids_gif_GifInfoHandle_getCurrentPosition
    (JNIEnv *__unused env, jclass __unused handleClass, jlong gifInfo){
c2    GifInfo *const info = ((GifInfo *) (intptr_t) gifInfo);
        ...
c3-   const uint_fast32_t idx = info->currentIndex;
c4    if (info->gifFilePtr->ImageCount == 1) {
c5      return 0; }
c6    uint_fast32_t i;
c7    uint32_t sum = 0;
c8-   for (i = 0; i < idx; i++){
c9+   const uint_fast32_t maxFrameIndex = info->currentIndex
      == 0 ? info->gifFilePtr->ImageCount : info->currentIndex;
c10+  for (i = 0; i < maxFrameIndex; i++){
c11     sum += info->controlBlock[i].DelayTime; }
```

Fig. 16. A Java-C CLB case rooted in 'logic error' with incorrect algorithm implementation

transformations or memory management for effective Python-C integration. In Java-C projects, the dominance of native functions points to a reliance on direct JNI interactions, with less frequent use of NAX functions, indicating a more standardized approach to managing cross-language tasks.

The CLBs studied were mostly manifested in code semantics on data parsing, objects handling, data conversion, memory management (reference counting in particular), all across languages.

3.4 RQ4: Root Cause of CLBs

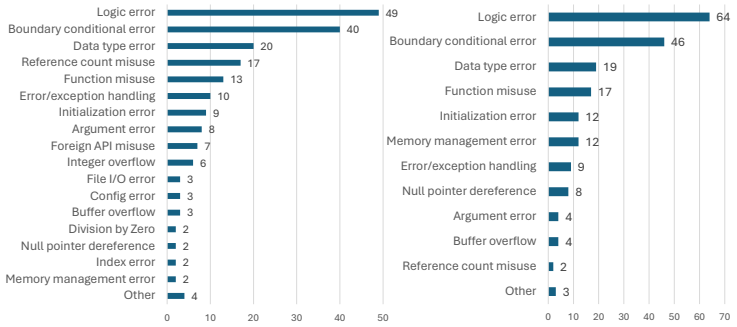


Fig. 14. Distribution of CLB root causes in Python-C (left) and Java-C (right).

We traced the data/control flow to determine the root causes (**RC**s) of CLBs, and identified 18 (Python-C) and 12 (Java-C) root cause types, as shown in Figure 14. Our results show that the most prevalent issues are related to logic errors, boundary conditional errors, and data type errors. The "other" category in our analysis encapsulates some issues that are difficult to classify. These are related to general concerns such as thread management issues, compilation errors, and compatibility errors. Subsequently, we presented definitions for these primary categories, conducting an in-depth analysis of their implications. Furthermore, we illustrated the concepts by including exemplar bugs representing instances within each category:

RC1: Logic error. A logic error is a type of runtime error that causes a program to deviate from its intended behavior through the cross-language information flow. Logic errors frequently stem from misconceptions or flawed presumptions made by the programmer during the development process. Our study shows that logic error incorporates an incorrect algorithm—a situation where the algorithm devised to resolve a problem is either flawed or incomplete.

Illustration. Figure 15 illustrates a logic error due to incorrect algorithmic implementation across languages. The Python code aims to obtain the `minmax` of a given list of custom objects `T` through a lambda key function. The `minmax` function is implemented in C, which invokes the key function that requires an argument (line py4). However, the C code prepares a tuple of size 0 for the argument, which is not consistent with

```
java1  int encryptFinal(...) throws ...{
java2    ...
java3    NativeCrypto.GCMEncrypt(...,iv.length,...);
---------------------------------------------------------------
c1  JNIEXPORT jint JNICALL
    Java_jdk_crypto_jniprovider_NativeCrypto_GCMEncrypt
    (JNIEnv * env, jclass obj, ..., jint keyLen,...){
c4    ...
c5    if (first_time_gcm == 0) {
c6      ...
c7 -    evp_gcm_cipher = EVP_aes_128_gcm();
c8 -  }
c9 +  }
c10+  switch(keyLen) {
c11+      case 16:
c12+      evp_gcm_cipher = EVP_aes_128_gcm();
c13+      break;
c14+    case 24:
c15+      evp_gcm_cipher = EVP_aes_192_gcm();
c16+      break;
c17+    case 32:
c18+      evp_gcm_cipher = EVP_aes_256_gcm();
c19+      break;
c20+ }...
```

Control Flow
Data Flow

Fig. 17. A Java-C CLB case with the root cause of 'boundary conditional error' [12].

the lambda function taking one argument, hence causing the logic error. To fix this, the variable `funcargs` is assigned a tuple object of size 1.

Pattern. The main pattern of logic errors is incorrect algorithm implementation in both Python-C and Java-C. These errors primarily stem from a misunderstanding of how to correctly implement and handle algorithmic calculations in a cross-language context. The algorithm needs to account for all possible values, including edge cases where variables come from the other language. For instance, the algorithm failed to properly calculate because it did not account for the special case of a native variable being 0, leading to an incorrect result. In these cases, developers frequently misinterpret the logic necessary to handle specific scenarios, leading to flawed implementations.

As shown in Figure 16, the Java code aims to obtain the current position of a GIF image using the `getCurrentPosition` method. This method is implemented in C via the JNI function, where it calculates the total delay time by iterating over frames. However, the C loop condition did not account for `currentIndex` being 0, leading to incorrect calculations. By modifying the C code to use `maxFrameIndex`, which correctly reflects the frame count needed, the calculation includes the appropriate number of frames up to `currentIndex`. This cross-language adjustment ensures that the Java method integrates seamlessly with the C, maintaining accurate delay time summation.

In both Python-C and Java-C cases, logic errors often arise from an incorrect understanding of how to handle data states and edge cases in a cross-language context. This includes misinterpreting algorithm requirements and failing to account for all possible values and conditions. In Python-C, these errors frequently involve dynamic typing and flexible data handling. In Java-C, these errors more often involve logical conditions and algorithmic calculations.

RC2: Boundary conditional error. A boundary conditional error occurs when a condition at the boundary of an interval or at the extreme ends is not correctly handled on the cross-language information flow. These errors often manifest in scenarios like array indexing (accessing an element outside the bounds of an array), looping constructs (incorrect loop termination conditions), and numerical ranges (performing calculations at the limits of numerical ranges).

Illustration. In Figure 17, the RC2-induced CLB arises in the interaction between Java and C when assigning the encryption implementation in `EVP_aes_128_gcm()` to the `evp_gcm_cipher` pointer. The Java method `encryptFinal` calls the JNI function, which initially sets `evp_gcm_cipher` to `EVP_aes_128_gcm()` without considering the key length. The cause of this error is the lack of a mechanism to select the appropriate AES-GCM algorithm version based on the provided key length during the JNI call. This oversight leads to a boundary condition error as the fixed assignment to `EVP_aes_128_gcm()` does not accommodate different key lengths (16, 24, or 32 bytes).

To fix this bug, a switch-case structure was introduced in the JNI function. This addition ensures that before assigning the encryption algorithm to the `evp_gcm_cipher` pointer, the function evaluates the key length based on the data passed from the Java method and selects the corresponding AES-GCM algorithm (`EVP_aes_128_gcm()`, `EVP_aes_192_gcm()`, or `EVP_aes_256_gcm()`).

Pattern. Both Python-C and Java-C top patterns concern value/size validations. This involves verifying that values meet expected criteria and sizes are within acceptable ranges before proceeding with operations. This error pattern often occurs when checking specific states (e.g., system state, array state, and initialization state). Figure 17 provided an example showing a JNI function that validates the `keyLen` parameter, which determines the length of the encryption key.

```
C1  static inline void *opj_aligned_alloc_n(size_t alignment, size_t size)
c2  void* ptr;
c3  assert( (alignment != 0U) && ((alignment & (alignment - 1U)) == 0U));
c4+ if (size == 0U) {
c5+   return NULL;}
c6  #if defined(HAVE_POSIX_MEMALIGN)
c7    if (posix_memalign (&ptr, alignment, size)){
c8      ptr = NULL;} ...
```

Fig. 18. A Java-C CLB case with the root cause of 'boundary conditional error' with validation in values.

In Figure 18, the memory allocation in the C function `opj_aligned_alloc_n` is part of a larger system interfacing with Java. This function includes an additional check to ensure that the `size` parameter is not zero before proceeding with memory allocation.

Table 1. Comparison of CLB fixing strategy between Python-C and Java-C.

CLB Fixing Strategy	Python-C		Java-C	
	Count	Percentage	Count	Percentage
Apply appropriate error and exception handling	17	8.5%	15	7.5%
Apply proper memory management and reference counting	36	18.0%	22	11.0%
Correct logical mistakes in conditions or loops	25	12.5%	31	15.5%
Ensure thread safety and synchronization	2	1.0%	4	2.0%
Improve code structure, comments and error messages	19	9.5%	17	8.5%
Incorporate necessary checks or input validations	38	19.0%	36	18.0%
Optimize performance and memory usage	6	3.0%	6	3.0%
Update and fix function calls and APIs	23	11.5%	14	7.0%
Utilize appropriate data types and casting	29	14.5%	38	19.0%
Other	5	2.5%	17	8.5%

This cross-language interaction starts from the Java method `internalEncodeImageToJ2K`, which calls the JNI function `Java_org_openJpeg_OpenJPEGJavaEncoder_internalEncodeImageToJ2K`, eventually leading to the C function. The `size` parameter must be validated to avoid undefined behavior when calling memory allocation functions like `realloc`. If `size` is zero, `realloc` can exhibit implementation-defined behavior, potentially causing unexpected results such as returning NULL or reallocating to a minimum block size. This validation is crucial here to ensure consistent and predictable behavior across the Java and C boundary.

Boundary conditional errors in cross-language interactions occur due to differences in memory management, type systems, and error handling between languages. In Python-C, manual memory management and type flexibility in C contrast with Python's automatic memory management and dynamic typing, leading to discrepancies because C's lack of automatic checks can result in unhandled null pointers or type mismatches. In Java-C, Java's strict type checking and automatic garbage collection differ from C's flexible type system and manual memory handling, causing issues in state validation and pointer use.

Python-C/Java-C cases highlight the necessity of validating input values and sizes to prevent boundary conditional errors, ensuring indices and dimensions are within acceptable ranges before proceeding with operations. In Python-C, errors often arise from insufficient checks on array dimensions and indices, leading to out-of-bounds access, whereas in Java-C, the focus is on ensuring valid memory allocation sizes to prevent undefined behavior in low-level operations like realloc.

The two primary root causes of CLBs were logic errors and boundary conditional errors in language interoperations. The number of bugs rooted in logic errors far dominated over other root causes.

3.5 RQ5: Fixing Strategies of CLBs

We looked into the kinds of code changes involved in the fixes (i.e., *how* the fixes were made). Table 1 presents a comparative overview of CLB-fixing strategies employed in Python-C and Java-C projects. Below, we elaborate the top 2 fixing strategies (**FS**) across both language combinations:

FS1: Incorporate necessary checks or input validations.These checks/validations, often prior to essential operations, enhance the robustness and reliability of cross-language behaviors. By performing validations against null pointers, container boundaries, and logically invalid values, critical operations such as pointer dereferencing, array indexing, and arithmetic operations are safeguarded. Such checks prevent erroneous data and operations from propagating through the system, ensuring that cross-language interactions are managed securely and consistently.

As shown in Figure 10, the JNI function is designed to open a file descriptor from a Java object and manipulate it in native code. The foreign function `GetIntField` (at line c1) extracts the integer value of the descriptor field, demonstrating object field manipulation across languages. Subsequently, the native function duplicates the file descriptor (`fd`) obtained from the Java object and associates it with a file stream for reading. The code changes in lines c7 and c8 exemplify the implementation of an input validation strategy. This approach can be adapted across various programming languages that support system calls and error handling mechanisms, including Python, Java, and C.

```
c1-npy_intp ndmin = 0;
c2+int ndmin = 0; ...
c3-ndmin_obj = PyDict_GetItem(kws, npy_ma_str_ndmin);
c4-if (ndmin_obj) {
c5+   ndmin = PyLong_AsLong(ndmin_obj);
c6+   if (error_converting(ndmin)) {
c7      long t = PyLong_AsLong(ndmin_obj);
c8      if (error_converting(t)) {
c9        goto clean_type;} ...
c10+ ndmin = t;}
```

Fig. 19. A CLB fixed with the FS2 strategy.

Pattern. This fixing strategy has two top patterns: 1) Null check/validation, which involves validating pointers or object references in C to ensure they are not null before performing operations. For example, in Figure 9, the function in C starts by parsing the input arguments using foreign function `PyArg_ParseTuple`. If the parsing fails, the function returns NULL, indicating an error in input validation. 2) Type check/validation, ensuring that variables or inputs are of expected types, such as integers, arrays, or specific objects (common in Python, C and Java). For instance, the function `instanceof` is used for checking if an object is an instance of a list or other collection interfaces. But in Python, it uses `isinstance(variable, list)` to check if a variable is a list.

FS2: Utilize appropriate data types and casting. In cross-language programming, using appropriate data types and performing correct type casting are crucial to maintaining data integrity and preventing bugs. Mismatched data types between languages can lead to data corruption and unexpected behaviors. This strategy ensures that data passed between languages is stored and processed using the correct types, and strong type conversions (casting) are applied where necessary. For example, using an integer variable in the native language (C) to hold floating-point data passed from the foreign language (Python) led to data-corruption bugs. Such measures prevent errors arising from incorrect type handling and ensure operations are performed on the correct data representations, maintaining the integrity and reliability of cross-language interactions.

As illustrated in Figure 20, the original code utilized `uint_fast16_t` type for the `widthOverflow` and `heightOverflow` variables, which were assigned values passed from Java to C. However, if the Java side passes values that exceed the storage capacity of 16-bit integers, this leads to overflow issues. The modification involved using `int_fast32_t` instead of `uint_fast16_t` for calculating dimensional overflows in C to provide a much larger range for data handling. This change addresses CLBs related to data type mismatches, ensuring safer handling of variable values.

```
c1- uint_fast16_t widthOverflow =
    gifFilePtr->Image.Width - gifFilePtr->SWidth;
c2- uint_fast16_t heightOverflow =
    gifFilePtr->Image.Height - gifFilePtr->SHeight;
c3+ int_fast32_t widthOverflow =
    gifFilePtr->Image.Width - gifFilePtr->SWidth;
c4+ int_fast32_t heightOverflow =
    gifFilePtr->Image.Height - gifFilePtr->SHeight;
```

Fig. 20. A Java-C case exhibiting the CLB location of assignment.

Pattern. The top pattern of this fixing strategy is type mismatch. In Python-C, dynamic typing in Python often leads to implicit type conversion issues when interfacing with C, which requires explicit type definitions. For example, in Figure 19, the original implementation directly assigns the result of foreign function `PyLong_AsLong(ndmin_obj)`, which is used for converting the int type in Python to the long type in C, to `ndmin`, which is of type `npy_intp`. This assignment can lead to data type mismatches, potentially causing runtime errors or misinterpretations of the variable's value in C. The modified code assigns the result to a long variable t, which is a more appropriate type for holding the conversion result from the foreign function `PyLong_AsLong`. This pattern is most commonly associated with the misalignment in type expectations, where Python's flexibility contrasts with C's rigidity, leading to type safety violations.

In Java-C, the fix strategy often involves utilizing appropriate data types and careful casting to bridge the gap between Java's strict type system and C's more permissive casting rules. For example, errors occur when casting Java objects to C pointers, such as casting a Java Integer to a C int* or handling Java arrays as C arrays without proper conversion. Developers can align Java's strict typing with C by employing correct data types and explicit casting methods.

Two top CLB fixing strategies are adding missing checks and using correct data types, both against cross-language data flow, together accounting for 37% (Java-C) and 33.5% (Python-C) of CLBs.

3.6 Associations between CLB Lifecycle Aspects

Table 2. Association between root causes and symptoms

CLB Root Cause	CLB Symptom	Support	Confidence	Lift
Memory management error	Memory leak	2%	57%	10.88
Reference count misuse	Memory leak	2%	56%	10.71
Null pointer dereference	Crash	2%	70%	4
Buffer overflow	Error/warning message	1%	71%	3.01
Initialization error	Error/warning message	3%	55%	2.32
Argument error	Incorrect result/output	2%	58%	1.62
Logic error	Incorrect result/output	14%	54%	1.49
Boundary conditional error	Incorrect result/output	9%	42%	1.18
Error/Exception handling	Incorrect result/output	2%	42%	1.17

To understand the lifecycle mechanisms of CLBs, we analyze statistical associations between each pair of the 5 CLB aspects. We compute frequent *if-then* associations which consist of an antecedent (*if*, e.g., root cause) and a consequent (*then*, e.g., symptom), using the Apriori algorithm [35]. Below, we only discuss strong associations (with $lift$>1) discovered.

Table 2 shows the relationship between the root causes and symptoms of CLBs. The super strong association with $lift$>10 suggests that when a memory leak resulting from a CLB is observed, it is very likely that the root cause lies in memory management errors or reference count misuse. Meanwhile, when a CLB of these two root causes occurs, memory leaks are very likely observed as symptoms. The results also reveal that when crashes are encountered, null pointer dereference is often found as the root cause. In contrast, when CLBs due to buffer overflow or initialization errors occurr, only minor symptoms such as error messages or warnings are often observed. On the other hand, when incorrect results/outputs are encountered, it may not be easy to link the symptom with a specific root cause: it is significantly associated with 4 different root causes.

Table 3. Association between root causes and manifest. semantics

CLB Root Cause	CLB Manifestation Semantics	Support	Confidence	Lift
Reference count misuse	Cross-language memory management	3%	63%	6.10
Argument error	Cross-language data parsing	2%	50%	2.74
Null pointer dereference	Cross-language objects handling	2%	70%	1.83
Memory management error	Cross-language objects handling	2%	64%	1.68
Boundary conditional error	Cross-language objects handling	11%	49%	1.29
Error/exception handling	Cross-language objects handling	2%	47%	1.24
Initialization error	Cross-language objects handling	2%	45%	1.18

Table 3 shows the association between the root cause and manifestation semantics of CLBs. The strong association with a $lift > 6$ indicates that when issues in cross-language memory management are manifested, it is very likely that the root cause is reference count misuse. Conversely, when a CLB occurs due to reference count misuse, cross-language memory management problems are highly likely to be the semantics issue encountered. The results also reveal that when cross-language data parsing errors are encountered, argument errors are often identified as the root cause. On the other hand, when CLBs stem from null pointer dereference, memory management error, boundary conditional error, error/exception handling, or initialization error, they are frequently associated with cross-language object handling issues. This suggests that cross-language object handling problems are significantly linked to multiple root causes, making it more challenging to pinpoint a specific root cause based solely on the semantics issues observed.

Table 4. Association between locations and symptoms

CLB Location	CLB Symptom	Support	Confidence	Lift
Return	Incorrect result/output	4%	43%	1.26
Conditional expression	Incorrect result/output	8%	42%	1.26

Table 4 shows how locations relate to symptoms in CLBs. The results suggest that when incorrect results/outputs are observed, the bug may be located at either return site or conditional expressions. This means that while incorrect results/outputs are likely symptoms when bugs are found in these locations, it is challenging to pinpoint the exact bug location based solely on the symptom, as these symptoms are significantly associated with both locations.

The CLBs exhibit strong associations between reference count misuse and memory leaks, as well as between memory management errors and memory leaks. When cross-language memory management issues occur, the root cause is often improper reference count handling. CLBs occur at return statements and conditional expressions, commonly leading to incorrect results/outputs.

4 Discussion

4.1 Uniqueness of CLBs

In our paper, we report findings on CLB characteristics without always prefacing them with qualifiers such as “cross-language information flow” or “due to language interaction.” Nonetheless,

all our results are fundamentally tied to the definition of CLBs—as bugs induced by faulty cross-language information flow (see §1). We discuss the uniqueness of CLBs in two perspectives below.

4.1.1 Comparison to Single-Language Bugs. Some aspects of CLBs—particularly symptoms and root causes—may seem generic and like those of monolingual bugs. Yet there are notable differences.

Symptoms. Our study shows that CLB symptoms like incorrect outputs, error/warning messages, and crashes are frequently reported. While these manifestations are common to all software bugs, in cross-language contexts they are direct consequences of interoperability issues such as data type mismatches and incompatible memory management. Although performance bottlenecks and memory errors are also potential symptoms, they tend to emerge only under specific conditions. Moreover, error messages in cross-language systems often include low-level details from native libraries, making them harder to interpret, and crashes can result from memory management conflicts between, for example, garbage-collected and non-garbage-collected languages. These factors mean that while the symptoms may look similar to those in single-language bugs, their underlying causes are distinctly tied to cross-language interactions.

Root Causes. Although common root causes—such as logic errors, memory management issues, and API misuse—are present in both CLBs and monolingual bugs, their origins in CLBs are uniquely rooted in faulty language interactions. In a single-language environment, data type errors typically arise from mishandled type casting within a consistent type system, often caught by the compiler or interpreter. In contrast, cross-language contexts require converting data types between different systems, which can lead to truncation, precision loss, or misinterpretation. For example, passing a 64-bit integer from C to a Java method expecting a 32-bit integer results in data loss and incorrect calculations in our study. In short, underlying reasons of CLB root causes are unique: incorrect data handling, type mismatches, and semantic misunderstandings between different languages.

4.1.2 Comparison to Monolingual Cross-Component Bugs. Certain CLB characteristics, such as boundary checks and input validation, also apply to cross-component interactions in monolingual systems. However, CLBs exhibit unique traits due to differences in semantics, runtime behavior, and data representations across languages.

In cross-language systems, semantic mismatches among type systems, memory models, and exception propagation models often lead to bugs that are rare in monolingual contexts. The use of foreign function interfaces (FFIs), bindings, or bridges adds layers of complexity that require explicit handling of data conversion. This often results in errors like data loss or misinterpretation that are less common in single-language applications. Furthermore, differing memory management paradigms—such as automatic garbage collection in Java or Python versus manual allocation in C—can exacerbate issues like memory leaks, double frees, or dangling pointers.

Also, error propagation is more challenging in cross-language environments, where a unified exception model is absent. For instance, in our study, errors originating in native C code invoked from Java is not properly communicated back to the Java layer. Even standard practices like boundary checking can falter; for example, a Java method passes an integer array to C under the false assumption that it is null-terminated, leading to unexpected behavior.

Overall, CLBs differ from both single-language bugs and monolingual cross-component bugs in two key respects: (1) they are more susceptible to interface-related failures due to challenges in data marshaling, type coercion, and error propagation, and (2) they involve both conceptual and implementation-level errors uniquely arising from flawed language interactions.

4.2 Implications and Actionable Insights

4.2.1 CLB Prevention/Avoidance. Our insights offer practical strategies for enhancing CLB prevention and avoidance measures, as exemplified below.

Be aware of various error-prone code. In §3.1, the diversity of symptoms related to cross-language development presents significant challenges, underscoring the importance of CLB prevention and avoidance measures to maintain system integrity and improve cross-language interaction efficiency. Our results (§3.2, §3.4) reveal that the nature of these bugs can vary significantly depending on the language combination. In Python-C, CLBs frequently occur at cross-language API callsites (*Loc3*). This location is the third most common for CLBs in Python-C projects but ranks much lower in Java-C projects. Conversely, in Java-C integrations, conditional statements (*Loc4*) are more prone to CLBs, ranking third in frequency for Java-C projects. To mitigate CLBs, developers should understand the specific challenges associated with the particular language combination.

Despite these differences, certain bug locations and root causes are common across both Python-C and Java-C. Assignments (*Loc1*) and local function calls (*Loc2*) are the top two locations where CLBs occur in both language combinations. These are critical points where cross-language information flows, and any mismatch or mismanagement can lead to bugs. The most dominant root cause of CLBs in both Python-C and Java-C projects is logic errors (*RC1*). Additionally, boundary conditional errors (*RC2*) are another common root cause in both language combinations. To mitigate CLBs, developers should be mindful of these common challenges in cross-language development.

Need for cross-language coding assistance/recommendations. Our analysis in Sections 3.2 and 3.4 underscores cross-language development's complexities and error-prone areas. Developers face significant hurdles when working with language combinations like Python-C and Java-C. In addition to keeping in mind the common challenges in cross-language development, developers must also understand the specific challenges associated with their chosen language combination. These findings highlight the critical need for coding assistance and recommendations to mitigate the burden on developers, thereby reducing CLBs and enhancing multilingual software reliability.

Note that while reference count misuse and null/type checking are well-known strategies for error prevention, our contribution lies in systematically categorizing these issues as they manifest specifically in *cross-language interactions*. For example, we highlight patterns of reference count mismanagement that are unique to the boundary between Java and C and between Python and C, where the contrasting memory management paradigms (automatic vs. manual) exacerbate the problem. Also, our recommendations extend beyond generic best practices by addressing cross-language-specific root causes. For instance, we identified subtle mismatches in language semantics (e.g., data encoding differences or type conversion misbehaviors) that are unlikely or much less likely to arise in monolingual contexts.

4.2.2 CLB Detection. Our findings on the locations and root causes of CLBs provide concrete guidance on developing CLB detection/testing/debugging solutions.

Use symptoms to detect root cause and location. Our result (§3.6) provides critical insights into how specific symptoms can quickly indicate underlying root causes and pinpoint bug locations within CLBs. For example, the result reveals that memory leaks are highly associated with root causes such as memory management errors and reference count misuse. This strong association suggests that when a memory leak is detected, it is highly probable that it stems from improper memory management practices or incorrect handling of reference counts. Consequently, CLB detection tools should prioritize monitoring memory-related operations and reference counting mechanisms. Furthermore, the association between bugs located in return statements and conditional expressions with incorrect results or outputs underscores these code regions as critical hotspots for CLBs. Developers can streamline the CLB detection process by systematically utilizing symptom indicators to trace back to probable root causes and specific bug locations.

Utilize data flow analysis for CLB localization. We utilized the tool CCC to go through the cross-language commits via backward and forward searches across language boundaries. This

approach, though effective in tracing the control flow paths from one language context to another, often proved insufficient for pinpointing the bug location and root cause solely through control flow analysis. We found that a significant portion of the bug detection process necessitated manual analysis of data flows to uncover the underlying source of issues. Leveraging data flow for efficient bug detection presents a promising way of addressing the intricate challenges posed by cross-language software systems. The experience has underscored the limitations of relying solely on control flow analysis to identify the root causes, especially when crossing language boundaries.

4.2.3 CLB Fixing/Patching. Our findings in §3.5 provide immediate guidance on fixing/patching CLBs through the common fixing approaches.

Start with the common strategies for CLB patching. In §3.5, we revealed two common fixing approaches developers adopted for CLBs. This result provides clear guidance on what kinds of code changes may be effective for writing CLB patches. For instance, developers may enhance the robustness of cross-language interactions by adding null pointer checks or validating input types before performing critical operations across the languages.

Leverage automated tools to support common patching strategies. Given the prevalence of input validation and data type management in CLB fixes, there is a significant opportunity to develop and utilize automated tools that assist developers with these issues. Tools that can automatically insert necessary checks or validate data types across language boundaries can streamline the bug-fixing process and reduce the likelihood of introducing new bugs. For example, integrating static analysis tools that help fix type mismatches or potential null pointer dereferences in cross-language code can efficiently resolve CLBs.

4.3 Limitations and Threats to Validity

Our study results are subject to manual labeling/analysis errors and human biases. To reduce this threat, we followed a rigorous coding process for obtaining the results for each RQ. Moreover, we adopted inter-rater and/or negotiated agreement protocols to resolve any diverged decisions, which help reduce potential errors and human biases via cross-checking.

Despite justifiable repository mining criteria, the subject Python-C and Java-C projects chosen may not be representative of all existing real-world Python-C and Java-C software. Accordingly, the CLBs we analyzed do not necessarily represent all CLBs in these two language combinations. To mitigate this threat, we considered a sizable number of projects of various domains while using random sampling with statistically significant sample sizes. Also, by implementing comprehensive criteria (e.g., language composition, star count) and processes (e.g., CLBF commit classification, manual verification), we mitigate the inclusion of noisy (e.g., tutorials, demos, and other non-representative) repositories. Our multi-faceted approach ensures that the analyzed projects are substantial, actively developed, and relevant to real-world cross-language software development.

4.4 Extension to Other Language Combinations

As we revealed, different language combinations have different characteristics (e.g., symptoms, locations, root causes). Yet, the framework and methodology in our study can be extended to other language combinations, though several challenges would need to be addressed in doing so.

Extension. First, our core methodology of identifying and analyzing CLBs via an automated tool combined with manual analysis is flexible and can be adapted to other language pairs. The key principles of our approach—bug identification via approximate automation, followed by CLB confirmation and deeper analysis of CLB patterns—apply to other language interoperability issues.

Second, while our tool is so far implemented for Java-C and Python-C, a similar tool could be built for other cross-language environments. The process of analyzing CLBs would remain similar,

but the tool would need to be adjusted for the specific inter-language calling conventions, memory models, and data marshaling practices of each language combination. Our tool is based on Joern [**?**] [46], which supports a number of languages, which facilitates the tool extension.

Challenges. First, as we showed, collecting the CLB dataset is challenging. To study another language combination, we would need to overcome data-collection challenges (e.g., for some combinations, there may not be as much resource available to collect a good number of CLBs).

Second, each language combination comes with its own set of interoperability mechanisms, such as foreign function interfaces (FFIs), binding libraries, and calling conventions. Adapting our tool would require retooling to accommodate these differences in the way languages interoperate.

5 Related Work

Several studies [4, 37, 52] are not really concerned about multilingual code—instead, they mainly address single-language software despite looking at a variety of languages. In contrast, we study cross-language defects that have been confirmed as bugs at code level. Yang et al. [48] manually inspected 586 Stack-Overflow posts related to those issues, while dissecting challenges behind the issues and summarizing current solutions. The study is based on developers' discussions rather than multilingual code. Li et al. investigated characteristics of bug resolution (e.g., bug open time and reopen rate) in 54 Apache projects [28] and those of bugs themselves (e.g., code change complexity) in 3 machine-learning frameworks [27]. Like ours, these two studies are based on code artifacts. Yet neither the bugs nor the projects were necessarily *multilingual*—they were defined by involving multiple languages without concerning language interactions. Also, unlike these works, we address code-level root cause, location, manifestation, and fixing strategies of multilingual code bugs without being limited to specific software domains.

Sultana et al. [42] conducted a case study of interoperability issues in 20 C/Fortran applications, focusing on the declaration, data sharing, and parameter passing in language interfacing, rather than code bugs induced by C-Fortran interactions. Similarly, Hu et al. [14] identified 9 common bug patterns concerning Python/C API usage. Our study concerns bugs induced by faulty cross-language information flow, including but not limited to those caused by API misuses.

Hwang et al. [15] target JVM-specific bugs related to JNI compliance by generating *synthetic* tests to expose deviations in JNI call handling. In contrast, our study examines real-world, application-level bugs in Java-C software using JNI, beyond JVM behavior such as specification compliance issues. Moreover, our study is grounded in analyzing *actual* bugs, which enables us to analyze a broader range of symptoms, root causes, and manifestation patterns as they occur in practice.

6 Conclusion

We presented the first comprehensive study of real-world ross-language bugs. Using our tool along with manual analysis, we collected and dissected the first set of 400 bugs carefully labeled from real-world Python-C and Java-C repositories on GitHub. We thus identified the symptoms, locations, manifestation characteristics, root causes, and fixing strategies of those bugs, discovering both the common and different bug patterns between the two language combinations. Based on our findings, we also provide actionable insights into how to prevent, detect, and fix cross-language bugs.

7 Data Availability

We publicly released all of our code and datasets at https://anonymous.4open.science/r/artifact-B457/.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under Grant CCF-2146233 and CCF-2505223, and in part by Office of Naval Research (ONR) under Grant N000142212111.

References

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

[2] Mouna Abidi, Md Saidur Rahman, Moses Openja, and Foutse Khomh. 2021. Are multi-language design smells fault-prone? an empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–56.

[3] Sora Bae, Sungho Lee, and Sukyoung Ryu. 2019. Towards understanding and reasoning about Android interoperations. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 223–233.

[4] Emery D Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the impact of programming languages on code quality: A reproduction study. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 4 (2019), 1–24.

[5] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. 2013. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *IEEE 37th annual computer software and applications conference*. 303–312.

[6] Francesca Del Bonifro, Maurizio Gabbrielli, and Stefano Zacchiroli. 2021. Content-based textual file type detection at scale. In *13th International Conference on Machine Learning and Computing*. 485–492.

[7] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[8] David Chisnall. 2013. The challenge of cross-language interoperability. *Commun. ACM* 56, 12 (2013), 50–56.

[9] Daniel P Delorey, Charles D Knutson, and Christophe Giraud-Carrier. 2007. 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 (WoPDaSD'07)*. 1–5.

[10] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. 2021. Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases. In *The Network and Distributed System Security (NDSS) Symposium*. 1–15.

[11] GitHub, Inc. 2023. GitHub REST API: provide APIs to retrive or query repositories in GitHub. https://docs.github.com/en/rest Accessed: 2025-02-02.

[12] Charlie Gracie. 2018. commit of ibmruntimes/openj9-openjdk-jdk8.

[13] Manel Grichi, Mouna Abidi, Fehmi Jaafar, Ellis E Eghan, and Bram Adams. 2020. On the impact of interlanguage dependencies in multilanguage systems empirical case study on Java native interface applications (JNI). *IEEE Transactions on Reliability* 70, 1 (2020), 428–440.

[14] Mingzhe Hu and Yu Zhang. 2023. An empirical study of the Python/C API on evolution and bug patterns. *Journal of Software: Evolution and Process* 35, 2 (2023), e2507:1–22.

[15] Sungjae Hwang, Sungho Lee, Jihoon Kim, and Sukyoung Ryu. 2021. Justgen: Effective test generation for unspecified JNI behaviors on JVMs. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1708–1718.

[16] Capers Jones. 2010. *Software engineering best practices: lessons from successful projects in the top companies*. McGraw-Hill Education.

[17] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. 2016. A large scale study of multiple programming languages and code quality. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 563–573.

[18] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. 2020. 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*. 127–137.

[19] Wen Li, Ming Jiang, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A cross-language dynamic information flow analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2513–2530.

[20] Wen Li, Li Li, and Haipeng Cai. 2022. 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*. 847–859.

[21] Wen Li, Li Li, and Haipeng Cai. 2022. PolyFax: A toolkit for characterizing multi-language software. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tool Demos*. 1662–1666.

[22] Wen Li, Austin Marino, Haoran Yang, Na Meng, Li Li, and Haipeng Cai. 2024. How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 3 (2024), 1–46.

[23] Wen Li, Na Meng, Li Li, and Haipeng Cai. 2021. Understanding language selection in multi-language software projects on GitHub. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings*

(ICSE-Companion). 256–257.
[24] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. PolyFuzz: Holistic greybox fuzzing of multi-language systems. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1379–1396.
[25] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. PyRTFuzz: Detecting bugs in Python runtimes via two-level collaborative fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1645–1659.
[26] Zengyang Li, Jiabao Ji, Peng Liang, Ran Mo, and Hui Liu. 2024. An exploratory study on just-in-time multi-programming-language bug prediction. *Information and Software Technology* 175, 1 (2024), 107524.
[27] Zengyang Li, Sicheng Wang, Wenshuo Wang, Peng Liang, Ran Mo, and Bing Li. 2023. Understanding bugs in multi-language deep learning frameworks. In *International Conference on Program Comprehension*. 328–338.
[28] Zengyang Li, Wenshuo Wang, Sicheng Wang, Peng Liang, and Ran Mo. 2023. Understanding resolution of multi-Language bugs: An empirical study on Apache projects. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.
[29] Philip Mayer. 2017. A taxonomy of cross-language linking mechanisms in open source frameworks. *Computing* 99, 7 (2017), 701–724.
[30] Philip Mayer and Alexander Bauer. 2015. 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*. 1–10.
[31] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development* 5, 1 (2017), 1.
[32] Leo A Meyerovich and Ariel S Rabkin. 2013. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 1–18.
[33] Elizabeth R Morrissey. 1974. Sources of error in the coding of questionnaire data. *Sociological methods & research* 3, 2 (1974), 209–232.
[34] Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. 2016. User-perceived source code quality estimation based on static analysis metrics. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 100–107.
[35] Raffaele Perego, Salvatore Orlando, and P Palmerini. 2001. Enhancing the apriori algorithm for frequent set counting. In *International Conference on Data Warehousing and Knowledge Discovery*. 71–82.
[36] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*. 428–439.
[37] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. 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*. 155–165.
[38] Simone Romano, Maria Caulo, Matteo Buompastore, Leonardo Guerra, Anas Mounsif, Michele Telesca, Maria Teresa Baldassarre, and Giuseppe Scanniello. 2021. G-Repo: a tool to support MSR studies on GitHub. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 551–555.
[39] Michael Seifert. 2019. commit of MSeifert04/iteration_utilities.
[40] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. 2001. Interprocedural control dependence. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 10, 2 (2001), 209–254.
[41] Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Phuong T Nguyen. 2020. A multinomial Naïve Bayesian (MNB) network to automatically recommend topics for GitHub repositories. In *Proceedings of the Evaluation and Assessment in Software Engineering*. 71–80.
[42] Nawrin Sultana, Justin Middleton, Jeffrey Overbey, and Munawar Hafiz. 2016. Understanding and fixing multiple language interoperability issues: the C/Fortran case. In *Proceedings of the 38th International Conference on Software Engineering*. 772–783.
[43] Federico Tomassetti and Marco Torchiano. 2014. An empirical assessment of polyglot-ism in GitHub. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. 1–4.
[44] Sergi Valverde and Ricard V Solé. 2015. Punctuated equilibrium in the large-scale evolution of programming languages. *Journal of The Royal Society Interface* 12, 107 (2015), 20150249.
[45] Bogdan Vasilescu, Alexander Serebrenik, and Mark GJ van den Brand. 2013. The Babel of software development: Linguistic diversity in Open Source. In *International Conference on Social Informatics*. 391–404.
[46] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy*. 590–604.

[47] Haoran Yang, Wen Li, and Haipeng Cai. 2022. Language-agnostic dynamic analysis of multilingual code: Promises, pitfalls, and prospects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1621–1626.

[48] Haoran Yang, Weile Lian, Shaowei Wang, and Haipeng Cai. 2023. Demystifying issues, challenges, and solutions for multilingual software development. In *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1840–1852.

[49] Haoran Yang, Yu Nong, Shaowei Wang, and Haipeng Cai. 2024. Multi-language software development: issues, challenges, and solutions. *IEEE Transactions on Software Engineering* 50, 03 (2024), 512–533.

[50] Haoran Yang, Yu Nong, Tao Zhang, Xiapu Luo, and Haipeng Cai. 2024. Learning to detect and localize multilingual bugs. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2190–2213.

[51] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. 2023. Declarative static analysis for multilingual programs using CodeQL. *Software: Practice and Experience* 53, 7 (2023), 1472–1495.

[52] Jie Zhang, Feng Li, Dan Hao, Meng Wang, Hao Tang, Lu Zhang, and Mark Harman. 2021. A study of programming languages and their bug resolution characteristics. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2684–2697.

Received 2024-09-13; accepted 2025-01-14