Language-Agnostic Dynamic Analysis of Multilingual Code: Promises, Pitfalls, and Prospects

Haoran Yang
Washington State University
Pullman, WA, USA
haoran.yang2@wsu.edu

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

Haipeng Cai∗
Washington State University
Pullman, WA, USA
haipeng.cai@wsu.edu

ABSTRACT
Analyzing multilingual code holistically is key to systematic quality assurance of real-world software which is mostly developed in multiple computer languages. Toward such analyses, state-of-the-art approaches propose an almost-fully language-agnostic methodology and apply it to dynamic dependence analysis/slicing of multilingual code, showing great promises. We investigated this methodology through a technical analysis followed by a replication study applying it to 10 real-world multilingual projects of diverse language combinations. Our results revealed critical practicality (i.e., having the levels of efficiency/scalability, precision, and extensibility to various language combinations for practical use) challenges to the methodology. Based on the results, we reflect on the underlying pitfalls of the language-agnostic design that leads to such challenges. Finally, looking forward to the prospects of dynamic analysis for multilingual code, we identify a new research direction towards better practicality and precision while not sacrificing extensibility much, as supported by preliminary results. The key takeaway is that pursuing fully language-agnostic analysis may be both impractical and unnecessary, and striving for a better balance between language independence and practicality may be more fruitful.

CCS CONCEPTS
- Software and its engineering → Dynamic analysis.

KEYWORDS
multi-language software, multilingual code, dynamic analysis

ACM Reference Format:

1 INTRODUCTION
Software failures are consequential and costly. A fundamental approach to assuring software quality hence mitigating these failures is to verify program behaviors via dynamic analysis [16, 17, 37]. For instance, among other such analyses, dynamic dependence analysis [15, 26] (including one of its special forms, dynamic slicing [36]), has empowered a range of applications in software quality assurance (e.g., fault diagnosis [20, 25], security testing [33, 35, 47]). Meanwhile, most (80+) real-world software today is multilingual (i.e., the program is written in multiple languages), according to recent studies regardless of the sample size (e.g., around 1,000 [52] or over 15,000 [63]) and data sources (e.g., at major companies [41] or in the open-source world [57, 58]). The latest study confirmed the status quo: only 18% of the studied systems use one language [50].

In this context, holistic analysis of multilingual code is key to systematic quality assurance of real-world software systems [48]. To understand this critical need, consider a few specific cases. In several samples of Android malware [12], the main app logic in one language invoked malicious code in another language. For instance, the game malware com.tinker.gameone [32] retrieves the user’s Facebook credential through its C# code, and passes the private data to an untrustworthy remote server in its Java code. Such issues also have been found in the Android framework itself. For example, as reported in CVE-2016-6691 [55], the framework called, from its Java code via the Java native interface (JNI), the QualComWi-Fi gbk2utf module in C++ which had GBK encoding errors.

Yet cross-language bugs are not limited to one language combination (e.g, Java-C) or one interfacing mechanism (e.g., JNI) [49], albeit the only few prior relevant works available all targeted that particular case (i.e., Java-C with JNI) [11, 39, 40, 46]. For instance, recently Li et al. [51] demonstrated multiple cases of high-severity security vulnerabilities of different kinds that happen across Python and C code in popular open-source projects such as NumPy [61].

While these examples are about security defects, cross-language correctness defects would happen the same way. The root cause is common: the defects originated in the code written in one language (i.e., one language unit) propagated to and were only exhibited in a different language unit. It would be difficult for single-language techniques/tools [19, 21–24, 28] to find these defects as their underlying analyses are not holistic—they dismiss cross-language dependencies and behaviors. Manual approaches (e.g., code review) are not always practical because humans can get easily lost in complex, large codebases like that of NumPy (one million SLOC) [61]. To address this challenge, the state-of-the-art approach ORBS [13] and its follow-up works [14, 44, 45] propose and promote language-agnostic dynamic analysis for multilingual code, focusing on (dynamic) program slicing as a demonstrating case. Here being language-agnostic means total language independence—the analysis is designed without assuming (i.e., independently of) any specific knowledge about the particular languages used in the multilingual software.
Given the general diversity (in terms of varying language combinations used) of multilingual code, the promises of the language-agnostic methodology advocated in these prior approaches are highly meritorious, both intuitively (e.g., it would work for any language combinations) and based on their evaluation results. Yet as we arguably and empirically show, there are also major pitfalls underneath this methodology that risk practicality. Ultimately, the sensible pursuit should be on the balance between the language independence of the analysis design and the practicality of the analysis with respect to real-world multilingual software.

In this paper, we reflect on the language-agnostic methodology as demonstrated in Orbs [13], the core in the line of works around it. We first briefly revisit how it works and the promises it holds (§2), followed by discussing the pitfalls as illustrated through a replication study of Orbs against 10 randomly chosen multilingual projects on GitHub (§3). We offer insights into our empirical findings and lay out a new research direction towards alternative tradeoffs between language independence and practicality that lead to more practical solutions (§4), as we look forward to the prospects of language-agnostic dynamic analysis of multilingual code.

Open science. Our artifact for this paper is available on figshare.

2 THE PROMISES

The state-of-the-art multilingual analysis, Orbs [13], achieves the greatest language-independence to date—it instruments at the given query (i.e., slicing criterion, which includes a code line number and a variable on that line), and the rest of the analysis is language agnostic. It works by tentatively deleting some other code lines, recompiling and executing the remaining code, and checking if the variable’s value changes—if so, those lines are deleted. This process is repeated until no more lines can be deleted, and the remaining code lines are considered the dynamic slice of the query.

Indeed, per its inner workings, Orbs only requires probing for the run-time values of criterion variables in the enclosing language unit. Other than this language-specific step, the analysis does not assume any knowledge about (the syntax or semantics) the languages involved in the multilingual code under analysis. This language-agnostic design holds great promises, because multilingual software is diverse and complex. Prior studies on successful projects in top companies reported that there were 2,500 languages in use and most applications were written in 2 to 15 languages [41]. Later studies based on open-source projects found that more than half of the samples used two or more languages. Most recently, further studies showed that multilingual code uses a variety of language combinations (e.g., java c++, python shell, javascript ruby php) [50] and diverse mechanisms for interfacing between different language units (e.g., one unit calling another via explicit calls to foreign functions, one unit embedding another) [51].

With these levels of diversity and complexity, it is clearly desirable to have an analysis be agnostic of the underlying languages of a given multilingual program, as it implies that the analysis can be perfectly generalized to any given multilingual software without additional (e.g., language-specific engineering) effort. The original evaluation experiments for Orbs [13] consolidated the promises—it worked reasonably well for not only small benchmarks (of a few hundred lines of code), but also with (four source files chosen from) a real-world multilingual project Bash (a Unix shell).

In sum, as in non-code-based approaches (e.g., entirely dropping any code analysis) [18], the language-agnostic methodology demonstrated via Orbs appeared to be highly promising.

3 THE PITFALLS

Despite its appealing promises, the language-agnostic design instantiated in Orbs [13] could face practicality challenges with large-scale, real-world multilingual systems. The largest-scale real-world case studied in the original Orbs evaluation only considered a quite small portion (four source files) of the project, rather than the holistic system. As a result, the complexity dealt with may not be representative of that of a whole, real-world multilingual system.

3.1 Technical Analysis

Technically, the design may suffer from a few limitations that make it impractical: (1) since the code lines to remove must be deleted together and lines are grouped speculatively [13] (despite aids of simple heuristics [14, 45]), it can take numerous trials, resulting in a long time to delete even one line (e.g., up to 1 minute per line for a small program of 2KLOC [45]); (2) every single trial requires a complete recompilation and then re-execution of the entire software, another potential source of overhead and inefficiency; (3) it only works with source code, because it relies on deleting the code at source level and (re)building the source after deletion; and (4) it is semi-automated as it requires users to write multiple scripts that fit the inner workings of the analysis for each system under analysis. As a result, the technique is not applicable where recompilation is infeasible (e.g., source code is unavailable or incomplete).

The fact that the deleted lines are grouped speculatively has another potential consequence—these lines may not be maximally removable for each instance of the line-deletion operation. In particular, since the grouping is heuristic and tentative while having to be done scrupulously to reduce the possibility of (re)compilation failure, there may often be code lines that could be deleted but are not comprehensively identified for deletion. The consequence is that the resulting slice may include many code lines that should not be in the slice (i.e., they should have been deleted since the criterion is not dependent on them). In other words, the language-agnostic methodology of Orbs may result in an excessive rate of false positives (i.e., great impression).

Above all, the greatest barrier with the language-agnostic methodology in Orbs may be its efficiency and scalability. Follow-up works achieved valuable improvements (e.g., enabling forward slicing [44]—the original implementation of Orbs only works for backward slicing, mitigating the efficiency issue [45]), but the practicality (efficiency/scalability wise) challenge remains due to the unchanged nature of the language-agnostic methodology.

3.2 Empirical Analysis

To validate the above dissection and understand the gap, we performed a replication study on Orbs using the artifact shared by the authors in their paper [13].

Dataset: We targeted open-source multilingual projects on GitHub that primarily used two or all of three programming languages:
Python, C/C++, and Java, because they are widely considered mainstream languages and commonly ranked among the top-5 lists by various sources (e.g., [? ]). Among all such projects, we sampled those that are popular (i.e., with 1,000 or more stars) and active (i.e., updated within the last six months). We also dismissed projects where the language unit in any of the three targeted languages accounts for less than 1% of total project code size. Then, from the resulting sample, we randomly selected 10 projects that cover all possible combinations of the three primary languages, as outlined in Table 1. The first column gives the project name and link.

**Metrics:** As per our technical analysis of the pitfalls, we mainly examine the efficiency of Orbs in terms of the slicing time cost. For each slicing criterion, we set a *timeout of 24 hours*, which is a reasonably large budget that a developer possibly affords in practice. In addition, concerning the practical usefulness of the resulting slices, we also look at the slice size—generally the smaller slices are more desirable because developers may not afford inspecting a very-large slice, especially given that Orbs itself does not provide additional guidance (e.g., inspection priorities or ranking of statements in a slice) for the post-slicing analysis.

**Procedure:** We have applied Orbs to the 10 chosen multilingual systems, on a Ubuntu 18.04.5 LTS server with Intel(R) Xeon(R) CPU E7- 4870 2.40GHz and 512GB RAM.

For each subject, we randomly picked one test to exercise it and 10 queries (i.e., slicing criteria) to compute dynamic slices for, such that each language unit contains the number of queries that is proportional to the code size of the unit. For a given criterion, if Orbs does not finish the slicing within 24 hours, we terminated it and considered the case a *timeout/failure*.

**Results:** The overall efficiency results are summarized in Table 1. The languages for which at least one query was picked are listed in the second column, and the total code size of each subject in the third. The fourth column (*#Qin.* indicates the number of queries with which Orbs successfully finished the slicing in 24 hours.

As shown, only 3 (relatively small) subjects saw some queries finished within the timeout, and Orbs timed out for any query of the other subjects. For the only 27 (out of 100 total) queries it returned a slice for, the average cost was 9.5 hours per query.

Table 2 outlines the further details on the 27 successfully finished cases, including the slicing criterion (SC) no. (2nd column), the slice size—the number of source lines of code (SLOC) in the slice (3rd column), and the number of hours (hrs) spent on computing each slice (last column). The **slice ratio**—the ratio of the slice size to the total number of executed lines in the subject execution underlying the slicing (4th column)—provides another perspective into the slice size with respect to the worst-case slicing results (i.e., all the executed lines are considered part of the slice).

As in the original Orbs evaluation, we did not have the ground-truth slicing results to compute precision and recall. Yet the numbers of Table 2 show that Orbs is very likely to be excessively imprecise—it produced more than half of the executed code lines in all of the slices for the two relative large subjects.

### 3.3 Key Insights

Overall, the empirical results appeared to corroborate the results of our technical analysis (§3.1): the language-agnostic design instantiated in Orbs suffered critical efficiency/scalability barriers and was subject to excessive imprecision.

Taking a closer look into the results, we observed that in all the failure (timeout) cases, Orbs was stuck in unfruitful cycles between recomputation and line deletion (because the deletion causes failures to compile). The underlying reason, as outlined earlier, was that Orbs made heuristic attempts in identifying the group of code lines to delete without even fully knowing about the syntactic (not to mention semantic) relationships among those lines. As a result, the majority of such attempts failed as the remaining program with those lines deleted failed to compile.

Meanwhile, in the small percentage of cases in which it finished the slicing within 24 hours, Orbs often identified excessively large groups of code lines to delete. In particular, when heuristically forming the group of code lines to delete, the deletion-line grouping step often ended up also including the lines that have no dependence relationships with the slicing criterion. The result was the excessively-large dynamic slices, as seen especially in the cases of Pyjnius. Apparently, there was no consistent correlation between the degree of this imprecision and the total code size of the multilingual system—e.g., Deep is much larger than Pyjnius (22.5 versus 5.1 KLOC), but the former saw much smaller slices produced by Orbs (50% versus 80%) in terms of slice ratio.

In short, this replication study led us to the following **insights**:

1. The need for almost no knowledge about any language makes Orbs almost fully language-agnostic, yet that lack of knowledge also led to totally uninformed hence opportunistic line deletion, a core step in the design of the language-agnostic methodology; thus, (2) a more practical design would need to strike a better balance between language independence and efficiency/scalability by utilizing slightly more knowledge about each language.

### 4 THE PROSPECTS

Following the insights obtained from our technical and empirical analyses (§3.3), we believe it is necessary to explore other tradeoffs
between the language independence and practicality (in terms of efficiency/scalability primarily but also concerning practically useful levels of precision). Note that language independence does matter for a multilingual code analysis, because the more independent the analysis is of the underlying languages, the more extensible/generalizable the analysis is to accommodate the diverse language combinations in real-world multilingual systems. Thus, a total relaxation of the tradeoff with the (almost fully) language-agnostic methodology as demonstrated in Otas by entirely compromising language independence to favor practicality is not a viable solution.

In general, we envision a new methodology for dynamic analysis of multilingual code (as for distributed systems [34, 38]) to decouple analysis (e.g., dependence computation or slicing) algorithm from data harvesting (i.e., the process of collecting the program data needed by the analysis). An overview of this decoupling design is depicted in Figure 1. The key idea and rationale is that (1) the data harvesting is realized via minimal, language-specific static analyses, relying on as little knowledge about each particular language of the multilingual code as possible, but the harvested data is language-independent in terms of its format and semantics, and (2) the analysis algorithm itself is that of an entirely language-agnostic dynamic analysis, as enabled by the language independence of the data harvested. In this way, we will overcome semantics disparity induced by language heterogeneity through minimal language-specific effort, so as to reach the practicality goal at the sweet spot in balancing language independence and practicality. Conceptually, the language-specific (static) analyses and the language-agnostic (dynamic) analysis are bridged through an analysis data unification layer in between where data harvesting will actually happen at runtime.

The key insight underlying this proposed design is that minimizing language-specific analysis hence maximizing language-independence and analysis-extensibility (yet not losing scalability) can be achieved by decoupling analysis algorithms from specific language semantics through harvesting language-independent data. As a proof of concept of this design, we built a cross-language dynamic data dependence analyzer for Java-C programs on top of an earlier work SensA [17]. We instrumented at every statement where a variable is defined or used as in [27] to send at runtime the variable value in a language-agnostic format to an analysis server through interprocess communication (IPC). We used Soot [42] and LLVM [43] for probing and identifying variable definitions and uses in the Java and C unit, respectively. We then ran the instrumented code twice, one normally to get the original execution and the other with statements of interest being voided (i.e., operations there changed to "no operation"). Once the analysis data is collected by the server, it computes dependencies through differencing the original and voided executions. Our experiments on a number of Java-C programs showed that the decoupling design worked successfully—it correctly computed all dynamic data dependencies across the two heterogeneous language units. The key here is that decoupling the analysis data collection and the core analysis algorithm is realized via IPC—which is by nature language-independent.

5 RELATED WORK

Previous studies suggested that unifying or abstracting language semantics is not scalable because it relies on heavyweight per-language engineering [53, 54, 59, 60]. Converting code in different languages into a uniform intermediate representation (IR) suffers from misinterpretation/misconversion issues due to language-semantics disparity. Also, the IR conversion for a given language is not always practical, because it requires vast engineering effort [12]; these issues are further aggravated by the evolution of each language—for instance, while LLVM [43] aims at a uniform IR for several languages, only a couple of front ends (e.g., for C/C++) received regular maintenance while those needed for the IR conversion for other languages did not hence are not practically usable. Meanwhile, a common or meta model [53, 60, 62] is not amenable to dynamic analysis, since code represented in such models (e.g., the uniform IR) cannot be executed anymore, nor are they able to represent execution information of the original code.

Earlier approaches [53, 54, 56, 59, 60] to cross-language analysis are mostly static while relying on substantial language-specific modeling and/or engineering. Recently proposed dynamic cross-language analysis [29] captures coarse-grained (file-level) dependencies by modifying OS kernel for regression test selection. Extracting co-change patterns to derive file-level dependencies achieves language independence by avoiding code analysis [30, 31], which is difficult to extend for finer granularity.

6 CONCLUSION

As the growing majority of today’s software systems are built using multiple languages, holistic analysis of multilingual code is essential for systematic software quality assurance. We revisited the promises of a state-of-the-art methodology for dynamic analysis of multilingual code that promotes such analyses be language-agnostic. While conceptually appealing and promising, this methodology may suffer technical limitations that impede its practical use against real-world multilingual software systems. We thus proceeded with an empirical analysis to demonstrate such pitfalls of the language-agnostic methodology. Following the insights distilled from our study, we envisioned a new methodology towards more practical dynamic analysis of multilingual software.

ACKNOWLEDGMENT

We thank our reviewers for constructive comments. This research was supported by NSF (CCF-2146233) and ONR (N000142212111).
REFERENCES


Steven Arzt, Tobias Kussmaul, and Eric Bodden. 2016. Towards cross-platform


