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:
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 (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3540250.3549173

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, 35]. For instance, among other such analyses, dynamic dependence analysis [15, 25] (including one of its special forms, dynamic slicing [34]), has empowered a range of applications in software quality assurance (e.g., fault diagnosis [19, 24], security testing [32, 33, 44]). 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 [49] or over 15,000 [60]) and data sources (e.g., at major companies [38] or in the open-source world [54, 55]). The latest study confirmed the status quo: only 18% of the studied systems use one language [47].

In this context, *holistic* analysis of multilingual code is key to systematic quality assurance of real-world software systems [45]. 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` [31] 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 [52], the framework called, from its Java code via the Java native interface (JNI), the `Qualcomm Wi-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) [46], albeit the only few prior relevant works available all targeted that particular case (i.e., Java-C with JNI) [11, 36, 37, 43]. For instance, recently Li et al. [48] 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 [58].

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 [18, 20–23, 27] 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) [58]. To address this challenge, the state-of-the-art approach ORBS [13] and its follow-up works [14, 41, 42] 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.

*Haipeng Cai is the corresponding author.

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 [38]. 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`) [47] 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) [48].

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, 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, 42]), 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 [42]); (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 [41]—the original implementation of ORBS only works for backward slicing, mitigating the efficiency issue [42]), 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:

Table 1: Efficiency results of Orbs on real-world systems.

Subject	Language combination	Code size	#Qfin.	Time (hrs)
Pyrasite [1]	`python c++`	1,580	10	2.67
Affinity [2]	`java c++`	4,677	*0*	*24+*
Pyjnius [3]	`python java`	5,071	10	7.36
Snappy [4]	`java c++ shell`	14,615	*0*	*24+*
Pysonar2 [5]	`java python`	18,247	*0*	*24+*
Deap [6]	`python c`	22,491	7	22.2
sbe [7]	`java c++ c`	48,406	*0*	*24+*
brotli [8]	`c c# java javascript`	51,073	*0*	*24+*
Vertx-web [9]	`java python`	124,942	*0*	*24+*
Mongo [10]	`c++ javascript python`	178,735	*0*	*24+*

`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 (*#Qfin.*) 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.

Table 2: Detailed results on the finished cases of slicing. (Sb.: Subject; Sn.: Slicing criterion no.; Ss.: Slice size (SLOC); Sr.: Slice ratio (%); Th.: Time (hrs)

Sb.	Sn.	Ss.	Sr.	Th.
Pyrasite	1	132	11%	3.00
	2	188	15%	3.18
	3	188	15%	3.15
	4	129	11%	3.10
	5	129	11%	3.40
	6	118	10%	1.94
	7	118	10%	1.95
	8	118	10%	1.93
	9	135	11%	2.48
	10	135	11%	2.60
Pyjnius	1	2,962	83%	8.33
	2	2,961	83%	8.29
	3	2,521	71%	8.32
	4	2,962	83%	8.36
	5	2,612	73%	5.79
	6	2,540	71%	8.42
	7	2,977	84%	8.42
	8	2,973	83%	8.42
	9	2,341	66%	4.69
	10	2,973	83%	4.53
Deap	1	5,460	53%	19.63
	2	5,460	53%	19.84
	3	5,460	53%	23.99
	7	5,008	49%	21.99
	8	5,008	49%	22.60
	9	5,008	49%	23.28
	10	5,008	49%	23.75

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 recompilation 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., Deap is much larger than Pyjnius (22.5 verus 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

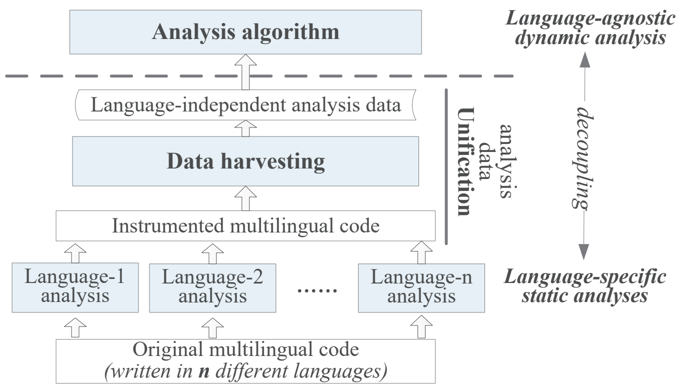
Analysis algorithm
Language-agnostic dynamic analysis
Language-independent analysis data
Data harvesting
analysis data Unification
decoupling
Instrumented multilingual code
Language-1 analysis
Language-2 analysis
......
Language-n analysis
Language-specific static analyses
Original multilingual code (written in n different languages)

Figure 1: Envisioned high-level design for better balancing language independence and analysis practicality.

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 ORBS 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 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 [26] to send at runtime the variable value in a language-agnostic format to an analysis server through interprocess communication (IPC). We used Soot [39] and LLVM [40] 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 [50, 51, 56, 57]. 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 [40] 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 [50, 57, 59] 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 [50, 51, 53, 56, 57] to cross-language analysis are mostly *static* while relying on substantial language-specific modeling and/or engineering. Recently proposed dynamic cross-language analysis [28] 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 [29, 30], 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

[1] 2021. https://github.com/lmacken/pyrasite.
[2] 2021. https://github.com/OpenHFT/Java-Thread-Affinity.
[3] 2021. https://github.com/kivy/pyjnius.
[4] 2021. https://github.com/xerial/snappy-java.
[5] 2021. https://github.com/yinwang0/pysonar2.
[6] 2021. https://github.com/DEAP/deap.
[7] 2021. https://github.com/real-logic/simple-binary-encoding.
[8] 2021. https://github.com/google/brotli.
[9] 2021. https://github.com/vert-x3/vertx-web.
[10] 2021. https://github.com/mongodb/mongo.
[11] Mouna Abidi, Md Saidur Rahman, Moses Openja, and Foutse Khomh. 2021. Are multi-language design smells fault-prone? An empirical study. *TOSEM* 30, 3 (2021), 1–56.
[12] Steven Arzt, Tobias Kussmaul, and Eric Bodden. 2016. Towards cross-platform cross-language analysis with Soot. In *SOAP*. 1–6.
[13] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-independent program slicing. In *ESEC/FSE*. 109–120.
[14] David Binkley, Nicolas Gold, Syed Islam, Jens Krinke, and Shin Yoo. 2017. Tree-oriented vs. line-oriented Observation-Based Slicing. In *SCAM*. 21–30.
[15] Haipeng Cai. 2018. Hybrid Program Dependence Approximation for Effective Dynamic Impact Prediction. *TSE* 44 (2018), 334–364.
[16] Haipeng Cai and Xiaoqin Fu. 2021. D2ABS: A framework for dynamic dependence analysis of distributed programs. *TSE* (2021).
[17] Haipeng Cai, Siyuan Jiang, Raul Santelices, Ying jie Zhang, and Yiji Zhang. 2014. SensA: Sensitivity Analysis for Quantitative Change-impact Prediction. In *SCAM*. 165–174.
[18] Haipeng Cai and Raul Santelices. 2014. Diver: Precise dynamic impact analysis using dependence-based trace pruning. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 343–348.
[19] Haipeng Cai and Raul Santelices. 2015. Abstracting program dependencies using the method dependence graph. In *QRS*. IEEE, 49–58.
[20] Haipeng Cai and Raul Santelices. 2015. A Comprehensive Study of the Predictive Accuracy of Dynamic Change-Impact Analysis. *Journal of Systems and Software* 103 (2015), 248–265.
[21] Haipeng Cai and Raul Santelices. 2015. A framework for cost-effective dependence-based dynamic impact analysis. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 231–240.
[22] Haipeng Cai and Raul Santelices. 2015. TracerJD: Generic trace-based dynamic dependence analysis with fine-grained logging. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 489–493.
[23] Haipeng Cai and Raul Santelices. 2016. Method-Level Program Dependence Abstraction and Its Application to Impact Analysis. *Journal of Systems and Software (JSS)* 122 (2016), 311–326.
[24] Haipeng Cai, Raul Santelices, and Siyuan Jiang. 2016. Prioritizing Change Impacts via Semantic Dependence Quantification. *IEEE Transactions on Reliability* 65, 3 (2016), 1114–1132.
[25] Haipeng Cai, Raul Santelices, and Douglas Thain. 2016. DiaPro: Unifying Dynamic Impact Analyses for Improved and Variable Cost-Effectiveness. *TOSEM* 25, 2 (2016).
[26] Haipeng Cai, Raul Santelices, and Tianyu Xu. 2014. Estimating the accuracy of dynamic change-impact analysis using sensitivity analysis. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*. IEEE, 48–57.
[27] Haipeng Cai and Douglas Thain. 2016. DistIA: a cost-effective dynamic impact analysis for distributed programs. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. 344–355.
[28] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression test selection across JVM boundaries. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 809–820.
[29] Catarina Costa, Jair Figueiredo, Leonardo Murta, and Anita Sarma. 2016. TIPMerge: recommending experts for integrating changes across branches. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 523–534.
[30] Catarina Costa, Jair Figueiredo, Anita Sarma, and Leonardo Murta. 2016. TIPMerge: recommending developers for merging branches. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 998–1002. Tool Demonstration.
[31] Cowbody Adventure. 2021. Android malware: com.tinker.gameone. https://github.com/ashishb/android-malware/tree/master/feabme.
[32] Xiaoqin Fu and Haipeng Cai. 2019. A Dynamic Taint Analyzer for Distributed Systems. In *ESEC/FSE*. 1115–1119. https://doi.org/10.1145/3338906.3341179
[33] Xiaoqin Fu and Haipeng Cai. 2021. FlowDist: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems. In *30th USENIX Security Symposium (USENIX Security)*. 2093–2110.
[34] Xiaoqin Fu, Haipeng Cai, and Li Li. 2020. Dads: Dynamic Slicing Continuously-Running Distributed Programs with Budget Constraints. In *ESEC/FSE*. 1566–1570.
[35] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li. 2021. Seads: Scalable and Cost-Effective Dynamic Dependence Analysis of Distributed Systems via Reinforcement Learning. *TOSEM* 30, 1 (2021), 10:1–10:45.
[36] 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.
[37] Sungjae Hwang, Sungho Lee, Jihoon Kim, and Sukyoung Ryu. 2021. JUSTGen: Effective Test Generation for Unspecified JNI Behaviors on JVMs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1708–1718.
[38] Capers Jones. 2010. *Software engineering best practices*. McGraw-Hill, Inc.
[39] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. Soot - a Java Bytecode Optimization Framework. In *Cetus Users and Compiler Infrastructure Workshop*.
[40] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.
[41] Seongmin Lee, David Binkley, Robert Feldt, Nicolas Gold, and Shin Yoo. 2021. Observation-based approximate dependency modeling and its use for program slicing. *Journal of Systems and Software* 179 (2021), 110988.
[42] Seongmin Lee, David Binkley, Nicolas Gold, Syed Islam, Jens Krinke, and Shin Yoo. 2018. MOBS: multi-operator observation-based slicing using lexical approximation of program dependence. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 302–303.
[43] 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 *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 127–137.
[44] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: Memory Leak Detection using Partial Call-Path Analysis. In *ESEC/FSE*. 1621–1625. https://doi.org/10.1145/3368089.3417923
[45] Wen Li, Li Li, and Haipeng Cai. 2022. On the Vulnerability Proneness of Multilingual Code. In *ESEC/FSE*.
[46] 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)*.
[47] 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*. IEEE, 256–257.
[48] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A Cross-Language Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA, 2513–2530.
[49] Philip Mayer and Alexander Bauer. 2015. An empirical analysis of the utilization of multiple programming languages in open source projects. In *EASE*. ACM, 1–10.
[50] Philip Mayer and Andreas Schroeder. 2012. Cross-language code analysis and refactoring. In *Proceedings of IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE, 94–103.
[51] Daniel L Moise and Kenny Wong. 2005. Extracting and Representing Cross-Language Dependencies in Diverse Software Systems. In *Proceedings of the 12th Working Conference on Reverse Engineering*. 209–218.
[52] National Vulnerability Database. 2021. CVE-2016-6691. https://nvd.nist.gov/vuln/detail/CVE-2016-6691.
[53] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. 2015. Cross-language program slicing for dynamic web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 369–380.
[54] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. 2017. A Large-scale Study of Programming Languages and Code Quality in GitHub. *Commun. ACM* 60, 10 (2017), 91–100.
[55] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in GitHub. In *ESEC/FSE*. ACM, 155–165.
[56] Miloš Savić, Gordana Rakić, Zoran Budimac, and Mirjana Ivanović. 2014. A language-independent approach to the extraction of dependencies between source code entities. *Information and Software Technology* 56, 10 (2014), 1268–1288.
[57] Dennis Strein, Hans Kratz, and Welf Lowe. 2006. Cross-language program analysis and refactoring. In *Proceedings of IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE, 207–216.
[58] The NumPy team. 2021. NumPy–the fundamental package needed for scientific computing with Python. https://github.com/numpy/numpy.
[59] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. 2000. A meta-model for language-independent refactoring. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*. IEEE, 154–164.
[60] Federico Tomassetti and Marco Torchiano. 2014. An empirical assessment of polyglot-ism in GitHub. In *EASE*. ACM, 1–4.