PolyFax: A Toolkit for Characterizing Multi-language Software

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

Li Li
Monash University
Melbourne, Victoria, Australia
li.li@monash.edu

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

ABSTRACT

Today's software systems are mostly developed in multiple languages (i.e., multi-language software), yet tool support for understanding and assuring these systems is rare. To facilitate future research on multi-language software engineering, this paper presents PolyFax, a toolkit that offers automated means for dataset collection from GitHub and two analysis utilities—a vulnerability-fixing commit categorization tool (VCC) and a language interfacing mechanism identification/categorization tool (LIC). The VCC tool immediately assists with assessing the vulnerability proneness of a given multi-language project based on its version histories, while the LIC tool enables dissection of the most important aspect of the construction of multi-language systems. Application of PolyFax to 7,113 multi-language projects with 12.6 million commits showed its practical usefulness in terms of promising efficiency and accuracy.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

multi-language software, tool, language interfacing, vulnerability

ACM Reference Format:
Wen Li, Li Li, and Haipeng Cai. 2022. PolyFax: A Toolkit for Characterizing Multi-language Software. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3540250.3558925

1 INTRODUCTION

Large-scale studies of existing software projects, along with the corresponding code repositories (e.g., those on GitHub), have propelled significant progress in understanding hence improving modern software systems. Practical tool support for mining such projects and analyzing those systems can be greatly instrumental [31], as they allow researchers to focus more on the core research questions and insights. For instance, tools for automated data collection/crawling, filtering/cleaning, and common characterization analyses are essential for research based on mining open-source projects.

In fact, a large body of research aims to mine and study open-source repositories [5, 6, 8, 18, 19, 25, 28–30, 32], enabled or at least facilitated by the underlying characterization tools. However, few of these studies [30] provided a commonly reusable set of data collection and characterization utilities (e.g., for project profiling, complete commits/source retrieval, etc.). Most importantly, existing characterization tools (e.g., D2A [32] and VccFinder [28]) are largely limited to single-language projects. Tools dealing with typically multi-language programs (e.g., [7, 17] for Android apps with native code and [10–13] for distributed systems often built with various languages for different components) ended up only addressing part of those systems that is written in one language (e.g., Java).

Yet the majority of today's software systems are written in multiple languages (hence they are noted as *multi-language* software)—for example, a recent prior study [23] confirmed that more than 80% of open-source projects on GitHub are developed with more than one language. It is also found lately that multi-language software is notably prone to security vulnerabilities mainly induced by the interfacing between different languages used in a software project [22]—in fact, this proneness has found correspondence to cross-language vulnerabilities with severe consequences [24]. On the other hand, tools supporting studies of multi-language software (e.g., those for identifying language interfacing and assessing proneness to vulnerabilities across languages) are critically lacking.

To fill this gap, we present PolyFax, a toolkit for characterizing multi-language projects on GitHub and dissecting the construction of multi-language systems. PolyFax consists of three related tools/modules: a crawler, a scrubber, and two analyzers. The crawler retrieves project data per given criteria, including general properties and historical commits (i.e., commit logs, authors, code snippets) and sources. The scrubber supports data pre-processing to facilitate further analysis. As two instances of such analyses, PolyFax includes a tool for vulnerability-fixing commit categorization (VCC) and one for language-interfacing identification/categorization (LIC). The VCC tool classifies a given commit as one that potentially fixes a vulnerability of a particular class, based on fuzzy matching between the commit log and keywords/phases summarized from CWE [1]. The LIC tool identifies the mechanisms in which the different languages used in a multi-language system interface with each other.

To assess its efficiency and effectiveness, we used PolyFax to characterize 7,113 projects with 12.6 million commits. It finished crawling, scrubbing, and analyzing the 193.9GB data in 23.1, 1.1, and 17.2 hours (1.47 for VCC and 2.5 for LIC), respectively. Our evaluation of the two analyzers based on random sampling and cross-validation showed that they achieved 80%+ precision and recall. PolyFax is the technical enabler of a recent study on the vulnerability proneness of multi-language software [22] and expected to serve future studies of these systems. The VCC tool is also immediately applicable to single-language projects.

A demo video for PolyFax is here and tool package here [20].

*Haipeng Cai is the corresponding author.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3558925

2 ARCHITECTURE

Figure 1 gives an overview of PolyFax's architecture. As its primary input, PolyFax retrieves open-source projects from GitHub [4]; optionally, users can customize the configuration as another input to let PolyFax only collect samples of interests.

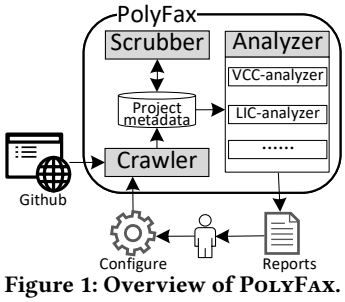

Figure 1: Overview of PolyFax.

With these inputs, PolyFax performs data analysis with *three modules*: **Crawler**, **Scrubber** and **Analyzer**. At first, the **Crawler** grabs repository profiles, clones the projects, and retrieves historical commits to the specified local storage. Then, the **Scrubber** performs pre-processing [16] of the textual information (e.g., project descriptions, commit logs) out of all the project metadata.

Finally, the **Analyzer** executes vulnerability-fixing commit categorization (VCC) and language interfacing mechanism categorization (LIC). VCC utilizes the FuzzyWuzzy technique [9] on commit logs to classify the commits into three high-level vulnerability categories (i.e., Porous defenses, Risky resource management, and Insecure interaction) [1]. LIC takes project sources as input and scans them with a finite state machine (FSM) modeled on summaries of language interaction patterns; it outputs a tuple of language interfacing mechanisms for each project. After all of these analyses complete, the **Analyzer** reports the results to users.

3 DESIGN AND IMPLEMENTATION

This section describes the design and implementation of PolyFax, elaborating its three modules: *crawler*, *scrubber*, and *analyzer*.

3.1 Crawler

The **Crawler** searches repositories using the GitHub API [2], filtering/grabbing repositories matching the criteria as configured. Specifically, it consists of three sub-modules, two **Profile Crawlers** (i.e., Crawler-by-Language and Crawler-by-Domain) and a **Commit Crawler**, as shown in Figure 2.

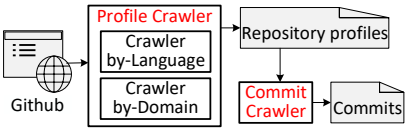

Figure 2: Overview of PolyFax.

3.1.1 **Profile Crawler.** A specific configuration defines the criteria of repository collection, specifying project attributes such as popularity, primary languages, size, creation date, and updated date. With these constraint values, the **Profile Crawler** constructs profile requests following the manual of GitHub's API [2].

Two profile crawlers are available in PolyFax for complementary purposes. **Crawler-by-Language** grabs repositories according to the user-specified languages while **Crawler-by-Domain** searches and grabs repositories based on a given functionality-domain list. When no languages or domains are specified, the crawler grabs projects with stars greater than 1,000 (configurable) by default.

For the over 70 different project properties available on GitHub, PolyFax retrieves 7 (i.e., repository id, stargazers count, languages, URL, pushed date, topics, and description). Users can customize to include more or less to meet varying analysis needs. As the output, a set of **repository profiles** is stored in a database.

3.1.2 **Commit Crawler.** With the repository profiles as input, **Commit Crawler** clones all the projects to the local storage, and then retrieves (simply using `git`) and parses all the commit information for each repository. This approach is much more efficient than grabbing the commits using GitHub APIs due to the rate limits of GitHub [2]. For each commit, PolyFax saves five primary features (i.e., commit identifier, author, date, related issue (if existed), and commit log); hence users can retrieve code changes and details of issues for the commits for further, in-depth analyses.

3.2 Scrubber

Usually, an insightful analysis is not readily feasible by just directly using the raw, potentially noisy (e.g., textual) information [16]. Hence, the Scrubber is responsible for data cleaning, taking the raw text (e.g., project descriptions, commit logs) as input. This pre-processing procedure transforms the text to accommodate a natural language processing (NLP) algorithm via four steps while leveraging NLTK for Python [3]: (a) remove all characters besides numbers, letters, and commas from the input text; (b) tokenize remaining text; (c) lemmatize each token; and (d) eliminate stop words. This process results in a set of words that capture the critical information for each text snippet.

3.3 Analyzer

The **Analyzer** analyzes the data collected from GitHub. Specifically in PolyFax, it includes two analysis tools: **VCC** and **LIC**.

3.3.1 **Vulnerability-fixing commit categorization (VCC).** We developed **VCC** based on the following assumption: if the log of a commit contains keywords/phrases indicating a class of vulnerabilities, then we regard the commit as aiming to fix those vulnerabilities. This is in the same spirit as prior work [29] identifying bug-fixing commits based on keyword search in commit logs.

Based on the assumption, **VCC** works in two steps: (1) Vulnerability keywords summarizing. By summarizing the top 25 most dangerous CWEs [1], three high-level categories [26] are obtained as Porous defenses (11 CWEs), Risky resource management (8 CWEs), and Insecure interaction (6 CWEs). We applied the Scrubber to the description for each category and extracted security-related keywords or phrases. (2) Vulnerability keywords matching. Based on the per-category keywords, we improved the FuzzyWuzzy technique [9] to classify commit logs as outlined in Algorithm 1.

The algorithm first retrieves these categories (line 2) and cleans the given commit with pre-processText (line 3), followed by computing a match score between each category and the commit (lines 5-21). Specifically, it retrieves (line 7) and traverse keywords/phrases in each category (lines 8-20). Next, the commit log is split into n-grams (lines 9-17) for a given phrase/keyword of length n and matched against the phrase with FuzzyWuzzy (line 18). For better precision, we use a minimal score of 90 as the threshold (lines 6) and take the the highest score for all phrases of a category (lines 19-20) as the score against that category (line 21). The best-matching category is eventually returned as the vulnerability category for the given commit (lines 22-23).

Algorithm 1: Identifying and classifying a vulnerability-fixing commit

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

3.3.2 **Language interface categorization (LIC).** Through manually checking respective languages' official documentation, we derived four basic language interfacing mechanisms:

(1) *Foreign function invocation (FFI).* With FFI, the host language provides a foreign function interface to bridge its own semantics and calling conventions and those of the guest language's (e.g., Java Native Interface (JNI) in Java).
(2) *Implicit invocation (IMI).* IMI is a cross-language interfacing mechanism based on inter-process communications (e.g., remote procedure call (RPC)).
(3) *Embodiment (EBD).* With this mechanism, the languages are interdependent and coexist with each other, with the code of one language often embedded in that of another language (e.g., the interfacing among {`css`, `html`, `javascript`}).
(4) *Hidden interaction (HIT).* With HIT, there is no explicit indication of direct interaction between languages, but there may be indirect data connection between different languages.

Then, we devised a rule-based classification model **C** based on pattern matching and finite state machine (FSM) as follows:

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

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

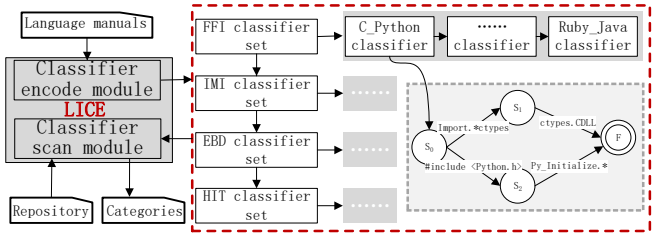

Figure 3: Language interfacing mechanism classification.

Algorithm 2: Classifying a project by language interfacing mechanisms

```
Input: P: a multi-language project repository
Output: L_P: the set of interfacing mechanism labels for P
1  Function classifyProject (P)
2      AC ← getClassifiers ()      /* Convene all the classifiers in LICE */
3      R ← compileRegex (AC)               /* Compile all regexs in AC */
4      R_C ← createMap (AC)   /* Create a map from regexs to classifiers */
5      foreach file in P do
6          R_M ← scanRegex (R, file)           /* Obtain matched regexs */
7          P_C ← pickClassifier (R_C, R_M)   /* Fetch relevant classifiers */
8          foreach C in P_C do
9              if classifyMatch (C, R_M) then
10                 L_P.insert (C.label)        /* One mechanism recognized */
11     if L_P == ∅ then
12         L_P.insert ("HIT")   /* Not FFI, IMI, or EBD, so defaulted to HIT */
13     return L_P
14 Function classifyMatch (C, R_M)
15     S_Q ← initStateQueue (C) /* Initialize with the initial state of C */
16     foreach r_m in R_M do
17         qlen ← S_Q.length
18         for k ← 0 to qlen − 1 do
19             S ← S_Q[k]
20             N_S ← nextState (S, r_m)   /* State transition on input r_m */
21             if N_S == NULL then
22                 continue
23             if isFinalState (C, N_S) then
24                 return TRUE                 /* Reached a final state */
25             else
26                 S_Q.push (N_S)   /* Save context for a matched pattern */
27     return FALSE
```

Based on this model, we developed a language interfacing classification engine (LICE), as shown in Figure 3. LICE consists of two collaborating modules: *Classifier encode* and *Classifier scan*.

(1) *Classifier encode module (CEM).* CEM aims to construct a chain of classifier set (`FFI`, `IMI`, `EBD`, and `HIT`). For the `FFI` classifier set, we manually summarized the interfacing code patterns for top languages [14]; and 20 `FFI` classifiers were finally constructed (e.g., `c_java`, `c_python`). For IMI, we implemented 7 classifiers by investigating code patterns based on standard components that support remote calls (e.g., D-bus [27], gRPC [15]). The `EBD` classifier set consists of one classifier for languages {javascript, css, html} as only these three languages are interdependent and exist in the top language selections. The `HIT` set includes one classifier for the projects without explicit code patterns.
(2) *Classifier scan module (CSM).* With a repository as input, CSM scans the source files one by one and works in a best-effort fashion to obtain all language interfacing types. As shown in Algorithm 2, after compiling regexs in all the available classifiers (lines 2-4), PolyFax finds matched patterns $\mathbb{R} = \Phi(\mathbb{I})$ (line 6) in each file (line 5) and picks relevant classifiers (line 7). If a classifier accepts all the matched patterns (regexs), the corresponding mechanism is recognized (line 8-10). To determine the acceptance, PolyFax runs the nFSM as a non-deterministic finite automaton against those regexs (lines 15-27). Importantly, it maintains a matching context (via the state queue S_Q) to obtain all possibly accepted regex sequences.

4 EVALUATION

We evaluate PolyFax through the following two research questions:

(1) **RQ1** What is the efficiency of PolyFax?
(2) **RQ2** What is the accuracy of PolyFax?

Table 1: Time cost of individual modules on analysis of 7,113 projects with 12.6 million commits.

Module	Crawler	Scrubber	VCC	LIC
Time cost (in hours)	23.1	1.1	14.7	2.5

Table 2: PolyFax's time costs with growing sample sizes (T-* denotes "the time cost (in minutes) of").

Subset No.	#Projects	Size (GB)	#Commits (K)	T-VCC	T-LIC
1	1,113	20.2	1,134	77.1	18.2
2	2,000	50.3	3,150	173.3	39.6
3	4,000	121.4	8,316	404.5	96.5

Experiment Setup. In the experiments, we ran the **Crawler** in PolyFax without languages or domains specified; the default star count for repositories was configured in [1000, 15000], and only **multi-language** projects were saved. Both **Crawlers** and **Analyzers** in PolyFax worked in single process. All experiments were conducted on a 64-bit Ubuntu 18.04 with a 32-core CPU (AMD Ryzen Threadripper 3970X) and 256 GB memory.

4.1 RQ1: Efficiency of PolyFax

With the default configuration, PolyFax took a total of 41.4 hours to finish the whole process of data collection and analysis, during which 7,113 projects and 12.6 million commits were collected [21]. The time costs of individual modules are shown in Table 1. By contrast, PolyFax is much more efficient since it could take about three months to grab 12.6 million commits by GitHub API [2].

To further evaluate the relationship between the efficiency of the two analyzers and the sample size, We divided the 7,113 samples into three subsets and evaluated the time cost on the subsets as shown in Table 2. From the results, we can see that the time cost of VCC grows almost linearly as the number of projects or the number of commits increases; a similar correlation can be found between LIC and commit or project count. This shows that PolyFax is capable of (scalable for) large-scale repository mining.

4.2 RQ2: Accuracy of PolyFax

To evaluate the effectiveness of PolyFax, we adopted a strategy of random sampling followed by cross-validation.

Evaluation of VCC. To evaluate the accuracy of VCC, we randomly sampled 50 projects and 500 commits per project from the dataset and constructed ground truth manually for gauging the precision and recall of VCC. Specifically, the authors independently labeled the sampled commits following three steps: (1) read the commit log, (2) check the associated code snippet, and (3) check the issue comments if they exist.

It is worth noting that each ground-truth vulnerability-fixing commit corresponds to an *actual/confirmed vulnerability* rather than just keyword/phrase matches. After all the authors completed independent labeling, they cross-validated and accepted the label for each commit when all agreed. For cases with initial disagreement, dedicated discussions were held to reach final decisions.

Table 3 summarizes the evaluation results. While not complicated, our tool achieved a quite competitive level of accuracy compared to the state-of-the-art peer tool D2A [32], which only reported 53% accuracy (based on a small manual study of only 57 commits in total)—although we cannot make strong claims here since we did not compare both tools on the same dataset. Moreover, D2A only targets C/C++ projects while PolyFax is language-independent; hence it can be applied more broadly.

Table 3: Cross-validation results of the VCC tool

Category	% Commits	Precision	Recall
Porous defenses	43%	85%	89%
Risky resource management	48%	83%	81%
Insecure interaction	9%	91%	83%

Table 4: Cross-validation results of the LIC tool

Category	%Projects	%Precision	%Recall
FFI	28%	85%	89%
IMI	69%	78%	82%
EBD	35%	96%	90%
HIT	11%	81%	84%

Evaluation of LIC. Per its design, LIC can identify a set of language interfacing mechanisms for each input repository. For instance, given a repository with language selection {`java`, `c`, `python`}, LIC may classify it as {`FFI, II`} because `java` interacts with `c` through JNI while `c` interacts with `python` through D-bus. To evaluate LIC's precision, we randomly sampled 150 projects and conducted a cross-validation procedure to measure the precision and recall based on manual ground truth. According to the implementation described in Section 3.3.2, the evaluation results are based on the samples' top languages [14]. Table 4 presents the precision and recall of **LIC** on the samples; since **LICE** summarized all the possible interfacing mechanisms between top languages according to the respective official manuals, it achieved high precision and recall. Specifically, the precision ranged from a minimum of 78% for IMI up to 96% for EBD, and the recall ranged from 82% to 90%.

4.3 Discussion

PolyFax offers a series of useful features, including repository crawling, commit classification, and language interfacing categorization. Its precision and recall indicate its potential of being applicable for multiple purposes. For example, the VCC can be used for empirical analysis as wells for providing abundant training data for machine learning (or deep learning) based vulnerability detectors since the code snippets, issues, or even CVEs of the commits can be retrieved from the results of VCC. Moreover, it is not limited to particular languages due to its language-independent nature.

For another example, results of the LIC may inform the design of a cross-language vulnerability detector—algorithms specific to each interfacing mechanism will be more precise than generic ones for arbitrary interfacing mechanisms. For instance, for FFI, the algorithm may identify vulnerabilities with more precise data flow analysis based on foreign/native function calls.

5 CONCLUSION

We presented PolyFax, a novel toolkit for characterizing multi-language software. It offers the capabilities of mining the repositories of open-source multi-language projects. It also includes two analysis tools, for vulnerability-fixing commit categorization and language interfacing mechanism identification/categorization, respectively. We empirically demonstrated PolyFax's merits in efficiency and effectiveness against real-world open-source projects on GitHub. PolyFax is open source and publicly available.

ACKNOWLEDGMENT

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

REFERENCES

[1] 2020. categories of security vulnerabilities. https://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf.

[2] 2020. GitHub Developer: provides APIs to retrive or query repositories in GitHub. https://developer.github.com/v3.

[3] 2020. NLTK: platform for building Python programs to work with human language data. https://www.nltk.org.

[4] 2021. GitHub: a US-based global company, provides hosting for software development version control using Git. https://github.com/.

[5] 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.

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

[7] Haipeng Cai and Barbara Ryder. 2017. DroidFax: A Toolkit for Systematic Characterization of Android Applications. In *International Conference on Software Maintenance and Evolution (ICSME)*. 643–647. https://doi.org/10.1109/ICSME.2017.35

[8] Casey Casalnuovo, Yagnik Suchak, Baishakhi Ray, and Cindy Rubio-González. 2017. Gitcproc: A tool for processing and classifying github commits. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 396–399.

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

[10] Xiaoqin Fu and Haipeng Cai. 2019. A Dynamic Taint Analyzer for Distributed Systems. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1115–1119. https://doi.org/10.1145/3338906.3341179

[11] 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.

[12] Xiaoqin Fu, Haipeng Cai, and Li Li. 2020. Dads: Dynamic Slicing Continuously-Running Distributed Programs with Budget Constraints. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1566–1570.

[13] Xiaoqin Fu, Boxiang Lin, and Haipeng Cai. 2022. DistFax: A Toolkit for Measuring Interprocess Communications and Quality of Distributed Systems. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 51–55. https://doi.org/10.1145/3510454.3516859

[14] GitHub. 2020. The 2020 State of the OCTO——VERSE. https://octoverse.github.com/#project-spotlight-tensorflow. (2020).

[15] gRPC. 2020. gRPC Tutorial. https://grpc.io/docs/. (2020).

[16] Emma Haddi, Xiaohui Liu, and Yong Shi. 2013. The role of text pre-processing in sentiment analysis. *Procedia Computer Science* 17 (2013), 26–32.

[17] John Jenkins and Haipeng Cai. 2018. ICC-inspect: Supporting runtime inspection of Android inter-component communications. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 80–83.

[18] Siim Karus and Harald Gall. 2011. A study of language usage evolution in open source software. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 13–22.

[19] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2201–2215.

[20] Wen Li. 2022. PolyFax code repository. https://github.com/Daybreak2019/PolyFax. (2022).

[21] Wen Li. 2022. PolyFax dataset. https://hub.docker.com/repository/docker/daybreak2019/fse22_vpomc. (2022).

[22] Wen Li, Li Li, and Haipeng Cai. 2022. On the Vulnerability Proneness of Multilingual Code. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[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)*. IEEE, 256–257.

[24] 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.

[25] 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.

[26] MITRE. 2020. Common Weakness Enumeration. http://cwe.mitre.org/.

[27] Havoc Pennington. 2020. D-Bus Tutorial. https://dbus.freedesktop.org/doc/dbus-tutorial.html. (2020).

[28] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 426–437.

[29] 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.

[30] Alex Villazón, Haiyang Sun, Andrea Rosà, Eduardo Rosales, Daniele Bonetta, Isabella Defilippis, Sergio Oporto, and Walter Binder. 2019. NAB: automated large-scale multi-language dynamic program analysis in public code repositories. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 9–10.

[31] Haoran Yang, Wen Li, and Haipeng Cai. 2022. Language-Agnostic Dynamic Analysis of Multilingual Code: Promises, Pitfalls, and Prospects. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[32] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: a dataset built for AI-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 111–120.