**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

---

**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](https://example.com/video) and tool package [here](https://example.com/package).
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.

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 interface 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.

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 FuzzyWuzz 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 FuzzyWuzz (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: nCat: the vulnerability category of Cmmt
1 Function classifyCommit(Cmmt)
2 Cmmt ← pre-processText(Cmmt) /* Tokenize, stemmatize, etc. */
3 VC ← initialCategory() /* Categories with keywords/phrases */
4 Cmmt ← VC
5 foreach Cat in VC do
6 Score ← 0 /* The minimum match score as the threshold */
7 foreach Phrase in PhraseList do
8 if Phrase is a cross-language interfacing
9 then
10 maxScoreCat ← Score
11 Numer of tokens in n-gamms in Cmmt
12 Index ← 0
13 while Index ≤ Np do
14 xGramSet ← xGramSet appending xGramStr
15 Index ← Index + Np /* Split Cmmt into n-gamms */
16 xGramStr ← xGramSet
17 /* Match Phrase against Cmmt's n-gamms with FuzzyWuzzy */
18 Result ← FuzzyWuzzy.extractOne(Phrase, xGramSet)
19 if Result score > Score then
20 Score ← Result score /* One mechanism recognized */
21 return nCat.

3.3.2 Language interface categorization (LICE). 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:

\[ C = (S_0, F, \delta, S, R, \Phi), \quad S_0, F \subseteq S, \delta^S : S \times R^* \rightarrow S \]

In the model, $S_0$ and $F$ represent the initial and end state respectively; $S$ is the state set; $R$ is the pattern set; $\delta$ is the state transition function and $\Phi$ is a regular expression engine. Given a sequence of inputs $I = \{I_0, I_1, ..., I_n\}$, $C$ obtains a set of matched rules $R = \Phi(I); \quad \delta^S(S_0, R) = F$ then we say $I$ is classified by $C$.

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-jar, 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 $R = \Phi(I)$ (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?
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.

Table 3 summarizes the evaluation results. While not comparing both tools on the same dataset, we adopted a similar methodology to compare PolyFax’s merits in efficiency and effectiveness against real-world open-source projects on GitHub. PolyFax is open source and publicly available.

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