On the Vulnerability Proneness of Multilingual Code

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

Software construction using multiple languages has long been a norm, yet it is still unclear if multilingual code construction has significant security implications and real security consequences. This paper aims to address this question with a large-scale study of popular multi-language projects on GitHub and their evolution histories, enabled by our novel techniques for multilingual code characterization. We found statistically significant associations between the proneness of multilingual code to vulnerabilities (in general and of specific categories) and its language selection. We also found this association is correlated with that of the language interfacing mechanism, *not* that of individual languages. We validated our statistical findings with in-depth case studies on actual vulnerabilities, explained via the mechanism and language selection. Our results call for immediate actions to assess and defend against multilingual vulnerabilities, for which we provide practical recommendations.

CCS CONCEPTS

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

KEYWORDS

multi-language software, multilingual code, software security, cross-language vulnerability, language interfacing, regression analysis

ACM Reference Format:
Wen Li, Li Li, and Haipeng Cai. 2022. On the Vulnerability Proneness of Multilingual Code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549173

1 INTRODUCTION

Studies have shown that software written in multiple languages (i.e., *multi-language* software) is dominant [21, 41, 49, 53, 76, 86]. Intuitively, this prevalence has to do with the benefits of combining the best functional capabilities of different languages [3, 16, 55]. Yet the decisions on language selection may not fully depend on functionality considerations: As earlier works initially suggested [4, 40, 46], the decisions may come with security [35, 91] consequences.

Cyber threats rooted in code vulnerabilities [34, 62], as introduced during the construction of software, have been on the rise [30]. Despite the dominance of multilingual construction in modern software practice for years, it is still not clear *whether multilingual construction has significant security implications and real security consequences in terms of vulnerabilities in the code*. In particular, we do not know *whether certain language combinations are more associated with code proneness to vulnerabilities than others* and, if so, *what contributes to the greater or weaker associations*. It also remains unknown *whether the association, if exists, indicates actual vulnerabilities*. Answers to these questions are essential for understanding and defending the holistic security of multi-language software.

Among a number of relevant-looking works [13, 16, 43, 55, 76, 89, 93], the majority addressed *individual* languages. Of the few works on *multi-language* software, most were focused on prevalence [54, 86] and good/bad practices of developers [3], or only limited to JNI (Java-C) programs [40, 46, 91]. Grichi et al. [35] reported likely greater security risks of multilingual code than single-language ones, but still for JNI code only and based on only 10 projects. Individual languages were found to have little association with *bug* proneness [10, 76]. Yet it is unclear if the same holds between *vulnerability* proneness and selections of multiple languages. Relevant works [13, 21, 45, 53] looked at high-level companionship among languages, but not code-level interfacing between languages, nor were they concerned with security vulnerabilities.

In this paper, we conduct a *security-focused characterization study of multi-language software*, targeting those in the open source world while with a focus on *vulnerability proneness* through the lens of *language selection* and *language interfacing mechanism*. Motivated by the foregoing questions, our study has three **specific aims**:

- Elucidate the security relevance of language selection in terms of the quantitative association between such selections and the resulting code's vulnerability proneness;
- Explain/justify the relevance via such proneness of individual languages and language interfacing mechanisms;
- Validate/concretize statistical findings about the vulnerability proneness by connecting it to actual vulnerabilities.

To that end, we (1) developed a new taxonomy and two novel tools for multilingual code analysis, hence (2) conducted extensive statistical analyses of 4,001 popular multi-language projects on GitHub and 20.37 million commits in their 3-year evolution history, and (3) manually inspected 50 projects and 500 commits for each, both randomly sampled. With these facilities, we answered three questions (justified by the specific aims) with key findings as follows:

- **RQ1: How is language selection related to the vulnerability proneness of multilingual code that uses the languages?** We found that language selection was overall strongly associated with the proneness and the association was even stronger for specific categories of vulnerabilities. Some language selections (e.g., `c++ python`) were much more prone to vulnerabilities overall

*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/10.1145/3540250.3549173

than others (e.g., `c c++ python`), and the same selection (e.g., `c shell`) can be much more prone to one vulnerability category (e.g., *risky resource management*) than to others (e.g., *insecure interaction*). The most prone selections were `c c++ shell` and `c python`.

- **RQ2 What are the underlying factors contributing to the stronger or weaker vulnerability proneness of a language selection than others?** We identified and examined two intuitive contributors: the individual languages selected, and the mechanisms of interfacing across languages. We found that the interfacing mechanisms, especially those via code-level function calls (either explicitly or implicitly), were overall strongly associated with the multilingual code's proneness to both vulnerabilities in general and to particular vulnerability categories. Some mechanisms (e.g., `foreign function invocation`) were clearly more prone than others (e.g., `embodiment`), and the most prone mechanism was `implicit invocation`. We found overall even stronger association of individual languages among the studied selections with such proneness. The most prone were `java` and `c`. However, the proneness of language selections was correlated with that of the mechanism of interfacing across the selected languages, *not* with the proneness of the individual languages in the selections.
- **RQ3 Does the vulnerability proneness of multilingual code indicate actual vulnerabilities in the code?** The statistical proneness was validated and concretized in sizable random manual sampling to strongly indicate actual vulnerabilities. Extensive in-depth case studies further demonstrated the presence of various kinds of real-world vulnerabilities in the sampled multilingual code. All of the vulnerabilities were explainable by the underlying language selection and interfacing mechanism.

Importantly, we define/quantify the *proneness of multilingual code* as #likely vulnerability-fixing commits in the code's version history. Then, we define the *proneness of an individual language, a language selection, or an interfacing mechanism* as their statistical association with the proneness of the underlying code—and quantify these proneness metrics via the association significance/coefficients. This paper only addresses such associations, *not* causality—we never intend to claim multilingual code vulnerabilities are "caused by or due to" the individual languages, the holistic language selections, or the language interfacing mechanisms used in the code.

Contributions and significance. Our novel empirical results provide not only the overall strong statistical grounds for the vulnerability proneness of multilingual code, but also offer extensive evidence on connections between the proneness and actual vulnerabilities. Based on these findings, we provided practical recommendations to researchers, developers, and tool builders, which facilitate understanding, analyzing, and defending against vulnerabilities in multilingual code. We also contribute an automated vulnerability classifier of commits more accurate than state-of-the-art peer tools, the first automated language interfacing mechanism detector based on the first taxonomy of such mechanisms, and a real-world vulnerability dataset of a greater size, accuracy, and diversity (covering more projects) than peer datasets. These new tools and dataset immediately support the practical adoption of our suggestions.

Open science. Source code and datasets are all available in our artifact and have been made publicly accessible.

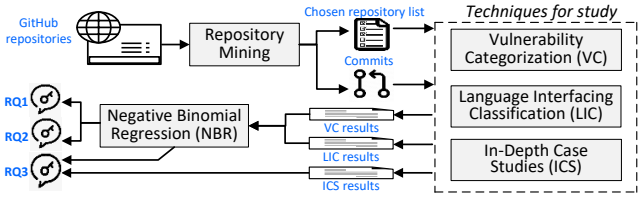

Figure 1: Overview of our study process.

2 METHODOLOGY

2.1 Study Overview

Figure 1 illustrates the overall process flow of our study. As its primary input, the process takes the repositories of open-source projects on GitHub [1]. From this source, we mined repository metadata hence selected multilingual projects that met a few criteria, and then retrieved commit information for each project.

Based on this dataset, we performed three main analyses: vulnerability categorization (VC), language interfacing classification (LIC), and in-depth case studies (ICS). VC identifies and categorizes security vulnerabilities based on commits; further, with the VC results, a negative binomial regression (NBR) analysis [7] examines the relationships between vulnerability proneness and language selection to answer **RQ1**. Then, LIC analyzes each project's code to recognize and classify its language interfacing mechanism. We also applied NBR over the same set of projects as for RQ1 to discover the association of those mechanisms and individual languages with the project vulnerability proneness to answer **RQ2**. Finally, we randomly sampled a subset of the projects to examine their vulnerability specifics through extensive manual inspections, concretizing the statistical findings obtained from RQ1 and RQ2, to answer **RQ3**.

2.2 Repository Mining

We mined repositories using the GitHub API [2] while applying four filters (selection criteria) as summarized and justified below:

- The repository has 1,000 or more stars—a popularity indicator and threshold used in peer prior studies [15, 22, 67, 76, 79]; these projects gained more attention, thus the (multi-language) software practice embodied in them potentially has greater influence.
- The repository has been updated in the latest 6 months—a reasonable indicator of activeness; more active projects is more likely to be representative of *ongoing* practices.
- The repository has been in maintenance for at least three years until 2020 or later—a reasonably long version history for identifying vulnerability proneness based on commits.
- The language selection did not change during the studied evolution period (i.e., three years)—essential for validly analyzing the effects of language selection on vulnerability proneness.

With these filters, our data sampling worked in two steps. First, we sampled projects that satisfy the first two criteria until we obtained 10,000 satisfying projects—this number represents a reasonably large scale. Second, we filtered these 10,000 projects using the other two criteria while dismissing single-language projects.

Specifically, for the second step, we ensured to only include unique projects—no project is a fork of another in the dataset. We also ignored those that are not actual software development projects

(e.g., course projects, teaching/tutorial code)—many on GitHub are not [42]. For each project, we retrieved the language URL for its repository, with which the GitHub API enables us to query the detailed language information, including the set of languages selected (referred to as *language combination* or *language selection*), of the project. Languages with <1% of total code size were removed from each project's language selection, and then projects using <2 languages left were dismissed.

The above two steps resulted in 4,001 *multi-language* projects which were eventually used in our study; these projects cover a variety of language selections unevenly distributed (e.g., `css`, `html`, `javascript`:10.4%, `c`, `c++`, `shell`: 4.8%, `c`, `python`:2.3%). We then retrieved all the commits of each chosen project in its 3-year version history, resulting in 20.37 million commits in total.

2.3 Vulnerability Categorization (VC)

To examine the security relevance of language selection (as a way to justify such selections), we needed to know, for each project, the overall vulnerability proneness and the proneness with respect to particular vulnerability categories. To that end, we developed a new tool that identifies and categorizes vulnerabilities based on commits, and applied it to each project in our sample set.

In a well-known list compiled by MITRE and the SANS Institute, the top 25 most dangerous CWEs [56] are categorized into three high-level categories [52]: *Porous defenses* (11 CWEs), *Risky resource management* (8 CWEs), and *Insecure interaction* (6 CWEs). For each category, we applied common text pre-processing [38] (e.g., tokenize, lemmatize, remove stop words) to the description of each CWE in the category to extract security-related keywords or phrases. For instance, we extracted the phrase "integer overflow" for CWE-190 in *Risky resource management*. Table 1 lists such keywords/phrases per category. We then applied the same process to changed code in each commit as a natural language token sequence.

Next, we used the *FuzzyWuzzy* technique [17] to match the keywords/phrases that characterize each of the three vulnerability categories against the tokens in a commit including its log (i.e., commit message) and code. Compared to other popular approaches (e.g., direct keyword searching, regex matching), FuzzyWuzzy does not require exact string matching but reports the similarity between the patterns and input strings; hence it strikes a better balance between precision and recall. For instance, in our preliminary experiments, direct keyword searching caused too many false negatives, due to exact string matching with respect to diverse/irregular wording/styles of commit messages (between vulnerability-relevant and irrelevant ones); Regex matching was relatively more effective, but not using any wildcards degenerated it to direct keyword searching, while extensively using wildcards caused too many false positives.

As shown in Algorithm 1, we first retrieve these categories (line 2) and apply the same text pre-processing to the given commit (line 3), followed by computing a match score between each category and the commit (lines 5–21). For each category, we retrieve (line 7) and traverse its keywords/phrases (lines 8–20). Next, for each phrase/keyword of length n, the pre-processed commit text is split into n-grams (lines 9–17), which are matched against the phrase with FuzzyWuzzy (line 18). For better precision, we use a minimal score of 90 as threshold (lines 6) and take the highest score overall all phrases of a category (lines 19–20) as the score against that

Algorithm 1: Identifying and Clasifying 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 ← preprocessText (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
```

category (line 21). The best matched category is finally returned as the vulnerability category of the commit (lines 22–23).

Any commit was identified as a vulnerability-fixing commit if it was categorized into one of the three categories with score>90. The key assumption here is that if a commit includes phrases/keywords relevant to a category, in its log or code, then the commit represents an attempt to *fix* the corresponding type of vulnerabilities. Given that accurate vulnerability detection/categorization remains an open challenge, we used this approximate, yet efficient and language-agnostic approach under this assumption, in the same spirit as prior work [76] identifying bug-fixing commits based on single keyword search but in commit logs only. Eventually out of the total of 20.37 million commits, we detected 141.38K as vulnerability-fixing, of which 36%, 48%, 16% fall in the three categories, respectively, as shown in Table 1.

To evaluate our approach, we randomly sampled 50 projects and 500 commits for each, and measured the precision and recall based on manual ground truth. The results are listed in the last two columns of Table 1. In producing the ground truth, the authors independently labeled the sampled commits, by (1) reading the commit log, (2) checking the associated code snippet, and (3) checking the issue comments whenever available. Then, 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. It is worth noting that each ground-truth vulnerability-fixing commit corresponds to an *actual/confirmed vulnerability* rather than just keyword/phrase matches. While not complicated, our technique achieved a quite competitive level of accuracy compared to the state-of-the-art peer tools—e.g., D2A [94] which only reported 53% accuracy (based on a small manual study of only 57 commits in total) and VCCFinder [70] which was even less accurate. We cannot make strong claims here though since we did not compare these tools on the same dataset—it is not immediately feasible to compare them against our tool on multi-language projects as they both target C/C++ projects. Meanwhile, our tool Lice is language-independent hence more widely applicable.

Table 1: Characteristics and our vulnerability categorization precision (*Prec*) and recall (*Rec*) of the 20.37 million commits (*Cts*)

Category	Security Vulnerability Description	Search Keywords/Phrases	%Cts	Prec	Rec
Porous defenses	these are the vulnerabilities that are related to defensive techniques that are often misused, abused, or just plain ignored.	missing authentication, missing authorization, hard coded credential, missing encryption, unnecessary privilege, user-controlled key, authorization bypass, broken cryptographic, excessive authentication, privilege escalation, etc.	36%	79%	81%
Risky resource management	the creation, usage, transfer, or destruction of important system resources is not properly managed.	deadlock, data race, data leak, buffer overflow, stack overflow, memory leak, exposed danger, integer overflow, memory corruption, untrusted control, etc.	48%	83%	86%
Insecure interaction	data is sent and received between separate components, modules, programs, processes, threads, or systems in insecure ways.	sql injection, command injection, request forgery, reflected xss, unrestricted upload, CSRF, unintended proxy, unintended intermediary, incomplete blacklist, origin validation error, etc.	16%	81%	88%

2.4 Language Interfacing Classification (LIC)

To gain an in-depth understanding of the security relevance of language selection, we had to determine the language interfacing mechanism of each project. We thus developed a taxonomy of such mechanisms through extensive manual search via code and relevant documentation reviews. As per this taxonomy, we then developed a new tool for automated interfacing classification.

2.4.1 Taxonomy. In our taxonomy, we define four mechanisms:

Foreign function invocation (FFI) A primary way of interfacing between mainstream programming language is FFI [46, 85], with which the host language provides a foreign function interface to match its semantics and calling conventions with those of the guest language. For instance, Java (the host) provides Java Native Interface (JNI) to support interaction with native code written in C (the guest). Two prominent features of FFI are that (1) the language interfacing follows standard definitions as documented (e.g., JNI [65], Python extension [73]) and (2) the interactions are implemented with *explicit* function invocations.

Implicit invocation (IMI) We define IMI as a mechanism with which different language components interact implicitly via inter-process communication (IPC)—for instance, socket-based message passing. This is often seen in multi-language distributed systems.

Embodiment (EBD) The involved languages do not interact by explicit/implicit invocations but via one embodying the other—the actual interaction often occurs within the underlying runtime system (e.g., a web browser engine) that executes the multi-language program. Typically, these languages are interdependent on each other and even co-exist. For instance, interfacing in the language selection {`css`, `html`, `javascript`} is via EBD as seen in Web apps.

Hidden interaction (HIT) In a selection with this mechanism, there are no any code-level evidence of connection, even implicit ones, between the languages. The interaction is often realized through external data sharing. For instance, a Python component downloads Web data as inputs of an analyzer written in C.

2.4.2 Classification Technique. For our study, we developed a language interfacing classification engine (Lice), as described below.

As shown in Figure 2, Lice takes a project repository as its input and it outputs the mechanism labels for the project. It works as a chain of classifier sets, where each set focuses on one of the four mechanisms and each classifier focuses on a unique pair of languages. Underlying each classifier **C** is a custom finite state machine (cFSM), as illustrated for `C-Python` in the figure and defined below:

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

where s_0 is the initial state; F is the final state set; S is the state set; R is the input set: a set of patterns each represented by a regex; δ is the state-transition function and Φ is a regex engine. Given a sequence

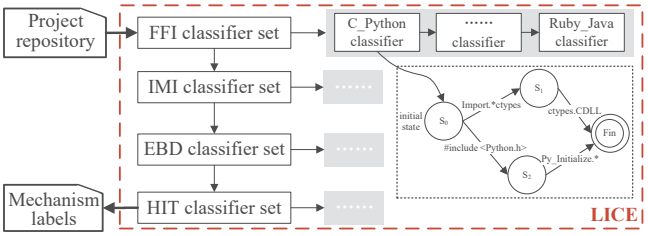

Figure 2: The proposed interfacing classification technique.

of inputs $\mathbb{I} = \{I_0, I_1, ..., I_n\}$, Lice applies the engine to obtain a set of matched patterns $\mathbb{R} = \Phi(\mathbb{I})$. Iff $\delta^*(s_0, \mathbb{R}) \in \mathrm{F}$, $\mathbb{I}$ is accepted by **C** and the resulting label is that of the mechanism focused on by the classifier set that includes **C**; otherwise, Lice moves to the next classifier set—the last (HIT) set must accept $\mathbb{I}$.

First, we manually built each classifier in Lice. We started with the top 12 languages in the top language selections found in our study (results for RQ1). Then, for each of the C_{12}^2=66 language pairs, we summarized the code (e.g., relevant function call) patterns of interfacing between the two languages each as a regex by extensively inspecting related real-world programs and official documentation (e.g., that on JNI). Next, we crafted the cFSM from the resulting regexs and their relationships—given the diversity of specific use cases of interfacing between the two languages, usually multiple patterns need to be observed in particular sequences to accurately recognize a mechanism, which justifies our use of automaton for classification. In the end, we found 20 of the 66 pairs used FFI hence created 20 classifiers for the FFI set.

Similarly, for the IMI classifier set, we created 7 classifiers by inspecting relevant artifacts (e.g., D-bus [69], gRPC [36]). From our top language selections, we only found one `javascript, css, html` using EBD; thus we created one classifier for the EBD set. Finally, the HIT set also includes one classifier which trivially accepts any input and labels it as "HIT". Following our procedure, Lice can be easily extended to include additional classifiers. Further details on the current 29 classifiers can be found in our artifact package.

When applying it to a project, Lice produces the mechanism label(s) by analyzing the source files as $\mathbb{I}$, according to Algorithm 2. After compiling regexs in all the available classifiers (lines 2–4), Lice 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, Lice 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.

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

```
Function classifyProject (P)
    AC ← getClassifiers ()        /* Convene all the classifiers in LICE */
    R ← compileRegex (AC)                  /* Compile all regexs in AC */
    R_C ← createMap (AC)     /* Create a map from regexs to classifiers */
    foreach file in P do
        R_M ← scanRegex (R, file)              /* Obtain matched regexs */
        P_C ← pickClassifier (R_C, R_M)    /* Fetch relevant classifiers */
        foreach C in P_C do
            if classifyMatch (C, R_M) then
                L_P.insert (C.label)           /* One mechanism recognized */
    if L_P==∅ then
        L_P.insert ("HIT")   /* Not FFI, IMI, or EBD, so defaulted to HIT */
    return L_P
Function classifyMatch (C, R_M)
    S_Q ← initStateQueue (C) /* Initialize with the initial state of C */
    foreach r_m in R_M do
        qlen ← S_Q.length
        for k ← 0 to qlen − 1 do
            S ← S_Q[k]
            N_S ← nextState (S, r_m)   /* State transition on input r_m */
            if N_S == NULL then
                continue
            if isFinalState (C, N_S) then
                return TRUE                    /* Reached a final state */
            else
                S_Q.push (N_S)   /* Save context for a matched pattern */
    return FALSE
```

Table 2: Evaluation results for LICE

Category	%Projects	Precision	Recall
FFI	35.67%	85%	89%
IMI	78.91%	78%	82%
EBD	32.18%	96%	90%
HIT	5.36%	81%	84%
Average		85%	82%

As shown, LICE may return a hybrid mechanism. For instance, given a repository with language selection {java, c, python}, LICE will classify it as {FFI, IMI}: Java interfaces with C through JNI while C interacts with Python through D-bus. Following a similar procedure to that for evaluating our vulnerability categorization technique, we evaluated LICE against 150 randomly sampled projects with manual ground truth. The results are summarized in Table 2.

2.5 Statistical Methods

We use *NBR* to model the number of vulnerability-fixing commits (as a *response*) against a set of factors (as *predictors*) related to multi-language software projects to study the relationship between vulnerability proneness and several indicator factors (e.g, language selection), as inspired by peer work [76]. We chose *NBR* as it can process data with *over-dispersion* [7, 76], a property of our datasets.

In this regression analysis for RQ1, each (*language selection*, *project*) pair is considered a sample from the population of multi-language software. Any of the project predictors is likely to influence the response. We consider the following predictors each as an independent (control) variable of the model: project age (#days since creation), language selection size (#languages selected), and the language selection itself. We mainly focus on language selections as the indicator variables in our analysis to answer RQ1.

As the factors in our study are unbalanced (the number of projects of different language selections varies in our datasets), we employed weighted effects coding [75]. With this method, each regression coefficient indicates the relative effect of the use of a particular language selection on the response as compared to the weighted mean of the dependent variable across all samples. Like in [76], we used a Chi-Square [18] test to check the dependence between two factor variables; in a case of dependence, we used an r × c equivalent of the phi coefficient to compute the effect size [19].

We did the same analysis for RQ2, but changing the indicator variables to interfacing mechanisms and individual languages.

Table 3: Distribution of the studied (4,001) projects over the top-20 language selections by %selecting projects

Language Selection	% Selecting Projects
css html javascript	10.4%
c c++ shell	4.8%
python shell	3.6%
html python	2.7%
html ruby	2.4%
css html javascript python	2.3%
javascript python	2.2%
css html javascript shell	1.9%
javascript shell	1.9%
c shell	1.9%
java javascript	1.8%
html java	1.8%
c python	1.6%
c++ python	1.6%
objective-c ruby	1.5%
go shell	1.5%
c c++ python	1.5%
javascript php	1.4%
c c++ python shell	1.4%
java shell	1.4%

Table 4: *NBR* coefficients on the vulnerability proneness of language selections. ***AIC=42150, BIC=42383.08, Log Likelihood=-21038, Deviance=8313.3; marks: ***p<0.001, **p<0.01, *p<0.05***

Independent factors	Coeff.	Std. Error
(Intercept)	1.4672	0.051 ***
project age	0.0001	0.001 ***
language selection size	0.0483	0.004 ***
css html javascript	-0.0841	0.073
python shell	0.2818	0.069 ***
go shell	0.3234	0.077 ***
c c++ python shell	-0.1922	0.201
javascript python	-0.0925	0.113
css html javascript shell	0.1522	0.092 *
c c++ python	-0.0300	0.162
objective-c ruby	-0.2838	0.120 *
html python	-0.4557	0.109 *
css html javascript python	-0.0666	0.101
c++ python	0.4613	0.144 **
html ruby	-0.1324	0.121
c python	0.6253	0.222 ***
c c++ shell	0.7641	0.098 ***
java shell	0.2766	0.096 **
javascript shell	-0.2041	0.084
javascript php	0.1061	0.088 **
html java	-0.1493	0.064
java javascript	0.2197	0.093 *
c shell	0.3019	0.125 *

3 RESULTS

In this section, we present and discuss main results and findings.

3.1 RQ1: Language Selection's Security Relevance

We studied two aspects of the security relevance of language selection: (1) *differential vulnerability proneness of language selections*—whether some language selections are more vulnerability-prone than others, and (2) *topic traits of security relevance*—the traits of security topics in the underlying commit data that contribute to (and thus help interpret) the effects of language selections on multi-language software security. For brevity, we elaborate the results for the top (most frequent) 20 language selections in our dataset, as listed in Table 3. As we found in preliminary experiments, *examining more or even all selections did not change our conclusions.*

3.1.1 Differential Vulnerability Proneness of Language Selections. To quantify the overall relationship between language selection and the proneness, we used a *NBR* model in which language selections are encoded with weighted effects as predictors and the number of vulnerability-fixing commits as a response (i.e., dependent variable). Details of this model are summarized in Table 4.

The first two independent factors (*project size*, *language selection size*) are control variables. Our study is not focused on the impact of these factors, albeit all the impact is significant as expected. All of the other independent factors—the language selection variables (e.g., `html ruby`)—are indicator variables.

The coefficients here compare each language selection (per project) to the grand weighted mean of language selections in all projects. Each coefficient falls in one of three categories: (1) statistically insignificant according to the p values (>0.05), (2) significant and positive, and (3) significant and negative. A positive coefficient indicates the corresponding language selection (e.g., `c c++ shell`, `css html javascript shell`) is associated with more commits intended for fixing security vulnerabilities, hence is more vulnerability prone, than an *average* language selection. Likewise, a negative coefficient indicates a language selection (e.g., `html python`, `objective-c ruby`) is less vulnerability prone than the average case—in other words, these selections are less likely to result in vulnerability-fixing commits.

Table 5: *NBR* coefficients on the proneness of language selections to each of the three high-level vulnerability categories.
marks: $^{***}p<0.001$, $^{**}p<0.01$, $^{*}p<0.05$

Independent factors	Porous defense		Risky resource mgmt.		Insecure interaction	
	Coeff.	Std. Err.	Coeff.	Std. Err.	Coeff.	Std. Err.
(Intercept)	-2.167	0.157 ***	0.4746	0.070 ***	-2.7066	0.149 ***
project age	0.0001	0.001 *	0.0002	0.009 **	0.0087	0.013
language selection size	0.0811	0.010 ***	0.0624	0.005 ***	0.0767	0.010 ***
`css html javascript`	-0.3234	0.218	-0.1741	0.105 *	0.3042	0.195 *
`python shell`	0.5527	0.212 **	0.3627	0.093 ***	0.3159	0.189
`go shell`	0.4895	0.221 *	1.0764	0.099 ***	-0.5575	0.264 *
`c c++ python shell`	-0.7087	0.641	-0.6020	0.249 *	-2.1799	0.180
`javascript python`	0.7464	0.339 *	-0.8805	0.161 ***	1.8017	0.333 ***
`css html javascript shell`	0.8216	0.281 **	0.1136	0.132	1.2334	0.301 ***
`c c++ python`	-0.2442	0.509	0.0927	0.202	1.8819	0.295
`objective-c ruby`	-0.2630	0.424	0.0855	0.170	-1.4854	0.626 *
`html python`	0.0735	0.315	-1.0742	0.163 ***	0.5449	0.287
`css html javascript python`	0.9098	0.306 *	0.2371	0.144	-1.6434	0.308 *
`c++ python`	-0.1056	0.468	1.4384	0.181 ***	-0.8860	0.583
`html ruby`	-0.4525	0.357 *	-0.6938	0.188 ***	1.6767	0.289 ***
`c python`	-0.9907	0.714	1.2994	0.279 ***	-1.7480	0.253
`c c++ shell`	-0.3065	0.315	1.5537	0.123 ***	-1.8835	0.443 ***
`java shell`	-0.3145	0.316	0.7744	0.124 ***	-1.0087	0.367 **
`javascript shell`	-0.4194	0.264	-0.2721	0.120 *	-1.5733	0.294 ***
`javascript php`	0.7695	0.226 ***	-0.7815	0.125 ***	1.9509	0.200 ***
`html java`	-0.1406	0.190	-0.2671	0.091 **	0.4598	0.181 *
`java javascript`	0.6595	0.254 **	0.1913	0.125	0.6386	0.237 **
`c shell`	-0.4142	0.405	1.3067	0.156 ***	-0.2594	0.371
AIC	*25342*		*26358*		*13278*	
BIC	*25575.03*		*26591.08*		*13511.08*	
Log Likelihood	*-12634*		*-13142*		*-6602*	
Deviance	*7962.3*		*7679.5*		*4327.3*	

In essence, the coefficient of the *language selection* variable can be understood as the expected change in the logorithm of dependent variable (i.e., #vulnerability-fixing commits) with the other independent variables considered constant. If we define a base factor (the average of expected changes across all *language selections* here) as κ, for a given coefficient of one *language selection* γ, then we can predict the response as: $\boldsymbol{N} = \kappa \times e^{\beta}$. For example, if a project with an average language selection had five vulnerability-fixing commits, among some total number of (any kind of) commits, then the number of vulnerability-fixing commits would be expected as $5 \times e^{0.7641} = 10.74$ in the case of using `c c++ shell`, much greater than the average of five. Similarly, selecting `html python` would mean fewer ($5 \times e^{-0.4557} = 3.17$) vulnerability-fixing commits.

Vulnerability proneness of the studied projects were strongly associated with language selections overall. Some selections were clearly more prone than others; `c c++ shell` *and* `c python` *were most prone.*

3.1.2 Topic Traits of Security Relevance. To gain a deeper understand of this relevance, we examined how language selection is associated with specific categories of vulnerabilities in terms of the topic traits of the underlying commit data (logs and changed code). To that end, we built a separate *NBR* model for each of the three categories (Table 1). Each model is the same as the one for the overall relationship except that the response (dependent variable) is the number of vulnerability-fixing commits belonging to the particular category. Table 5 lists the details of the three models.

The deviance for each per-category *NBR* model is smaller than the deviance for overall vulnerability proneness in Table 4, indicating that language selection had a greater impact on the proneness to specific vulnerability categories than its overall impact on the proneness to any kind of vulnerabilities. For example, the coefficient of `css html javascript` was not significant per Table 4. In contrast, it was significant in two per-category models here.

Porous defenses. Vulnerabilities of this category are due to the misuse or lack of use of necessary security defense techniques (e.g., *missing authentication for critical function*, *use of hard-coded credentials*, *missing encryption of sensitive data*). Our results suggest a strong association between language selection and these vulnerabilities, similar to the overall association (Table 4). Certain selections were particularly strongly associated with porous defense vulnerabilities. The most noticeable is `css html javascript python` for its greatest positive coefficient 0.9098 (and highest level of significance strength too). Note that `css html javascript` has little (and insignificant) impact on these vulnerabilities, though. This contrast points to the clear security influence of `python` when interacting with those three Web languages. In particular, using just `javascript` and `python` had a similar proneness (coefficient 0.7464); thus, these two may be most responsible for the (greatest) vulnerability proneness of `css html javascript python`. On the other hand, some selections (e.g., `html ruby`) had strong but *negative* impacts (e.g., coefficient -0.4525), suggesting they are relatively secure against this vulnerability category. Similar observations can be made for other language selections.

Risky resource management (mgmt.) 48% of our vulnerability-fixing commits fall in this category of vulnerabilities, which are due to improper creation, usage, transfer, or destruction of important system resources [52] (e.g., *integer overflow/wraparound*, *uncontrolled format string*). The results show that the impact of language selection on these vulnerabilities was generally more significant than that on the *porous defenses* category. One exception is `java javascript`, which had a significant impact on the latter but almost no impact on resource management vulnerabilities. Most of the language selections having a greater-than-average association with these vulnerabilities include `c` or `c++`: both are with unmanaged memory type (hence prone to memory errors [47, 76]) and well-known for being prone to memory vulnerabilities [61].

Insecure interaction. These vulnerabilities, to which 16% of our vulnerability-fixing commits belong, are a result of insecure ways

in which data is sent and received between separate components, modules, programs, processes, threads, or systems [52] (e.g., *cross-site request forgery*, *cross-site scripting*, *SQL injection*). While this category was the weakest among the three (with respect to the residual deviance of the model), the deviance explained by language selection was still strong. The selections with significant, positive coefficients were mostly combinations of languages commonly used in Web applications (e.g., `javascript php`, `java javascript`). This can be explained by the fact that vulnerabilities induced by the interaction among these Web development languages are indeed prevalent in Web applications—in particular, interfaces between these languages are well known to be vulnerable to code injections (e.g., *cross-site scripting* commonly exploited by injecting `javascript` code [80]).

Language selection was more strongly associated with the proneness to specific categories of vulnerabilities than to vulnerabilities overall. The top selections were more prone to resource management risks and insecure interactions than to porous defense risks.

3.2 RQ2: Factors Contributing to the Relevance

Earlier studies revealed cross-language links as possible points of high risks in multi-language systems in general [54] and inter-language dependencies as contributors to the vulnerabilities of JNI programs in particular [35]. We thus examine the effects of language interfacing on the vulnerability proneness of multilingual code that uses corresponding interfacing mechanisms, as a way to justify/explain the security relevance of language selection. We start with an overview of how various mechanisms are used, followed by the same two aspects of security relevance as examined for RQ1.

Another potential contributing factor lies intuitively in the individual languages selected. Thus, we also examine their effects.

3.2.1 Overall Use of Interfacing Mechanisms. For each project, Lice mined the interfacing mechanisms as completely as possible, resulting in some language selections having hybrid mechanisms. In total, it found 8 combinations of mechanisms, over which our 4,001 language selections were non-overlappingly distributed as: pure `FFI` 8.68%, `FFI EBD` 1.19%, `FFI IMI` 24.55%, `FFI IMI EBD` 1.26%, pure `IMI` 29.23%, `IMI EBD` 23.87%, pure `EBD` 5.87%, pure `HIT` 5.36%. *Implicit* interfacing was dominant: 86% of them used `IMI`, `EBD`, or both. One reason was that only a small portion (20/66) of the language pairs whose interfacing patterns underlaid the construction of Lice (cf. §2.4.2) supported FFI—not many languages support foreign functions. Another justification lies in the benefits of implicit interfacing—it helps reduce the coupling among different (language) components and the complexity/difficulty (although increasing the flexibility) of implementing the software, especially with the availability of mature supporting frameworks (e.g., gRPC [36] for interfacing among `c, python, java, ruby`). We also found a substantial use of `EBD`, mainly due to projects using the selection {`javascript css html`}. This echoes prior findings on the common use of general-purpose programming languages interfaced with domain-specific languages [54].

Implicit interfacing was dominantly used over explicit mechanisms like FFI (e.g., JNI), justifiable by the practical benefits of the former.

3.2.2 Differential Vulnerability Proneness of Language Interfacing. The 8 combinations of interfacing mechanisms were considered

Table 6: *NBR* coefficients on the vulnerability proneness of language interfacing mechanisms. ***AIC=42186, BIC=42267.89, Log Likelihood=-21080, Deviance=8545.8; marks: ***p<0.001, **p<0.01, *p<0.05***

Independent factors	Coeff.	Std. Error
`FFI`	0.2817	0.092 **
`FFI IMI`	0.1676	0.095 *
`FFI EBD`	-0.1454	0.415 *
`FFI IMI EBD`	0.7576	0.380
`IMI`	0.6441	0.164 ***
`IMI EBD`	-0.3059	0.092 *
`EBD`	-0.0811	0.089
`HIT`	0.4998	0.498

the independent variables (and predictors) in our NBR model here, where the number of vulnerability-fixing commits is again encoded as a response. Table 6 shows the details of this model which quantifies the overall security effects of language interfacing.

The NBR numbers revealed strong effects of language interfacing, explicit (i.e., `FFI`) or implicit (e.g., `IMI`, alone or with `FFI`), on the vulnerability proneness of multilingual code. It is known that using multiple languages make a system more vulnerable than when using a single language [35]. Our results complement by offering clear evidence of language interfacing mechanism as a major factor of the overall security relevance of using multiple languages. Indeed, contrasting the result here (e.g., Table 6) with that for RQ1 (e.g., Table 4) confirms that *the more (less) vulnerable mechanisms were those used in the more (less) vulnerable language selections.*

Table 7: *NBR* coefficients on the proneness of language interfacing mechanisms to each of the three high-level vulnerability categories. ***marks: ***p<0.001, **p<0.01, *p<0.05***

Independent factors	Porous defense		Risky resource mgmt.		Insecure interaction	
	Coeff.	Std. Err.	Coeff.	Std. Err.	Coeff.	Std. Err.
(Intercept)	-1.2328	0.152 ***	-1.5902	0.059 ***	-2.0517	0.096 ***
`FFI`	0.5434	0.315 **	0.4733	0.109 ***	-1.3708	0.180 *
`FFI IMI`	0.3562	0.321 *	0.4758	0.114 ***	-1.1683	0.239 *
`FFI EBD`	1.8357	0.266 *	0.7160	0.525	1.0950	0.634 *
`FFI IMI EBD`	0.3540	0.471	0.8610	0.487 *	-2.0167	1.186
`IMI`	1.1156	1.368 **	0.7457	0.531	0.6490	0.643 *
`IMI EBD`	-0.4470	0.320	-0.3917	0.113 **	-0.3215	0.241
`EBD`	-1.6342	0.312 *	-0.3942	0.108 ***	0.3044	0.233 **
`HIT`	-0.0088	0.270	1.2619	0.126 ***	-0.1634	0.642
AIC	*25082*		*26224*		*13330*	
BIC	*25163.89*		*26305.89*		*13411.89*	
Log Likelihood	*-12528*		*-13099*		*-6652*	
Deviance	*7521.4*		*7374.5*		*3897.1*	

Three mechanisms (`FFI`, `FFI IMI`, `IMI`) were significantly more vulnerability-prone, which accounted for the majority (62.5%) of the language selections in our dataset. Meanwhile, other mechanisms were less prone (e.g., `FFI EBD`) or not significantly associated with the proneness (e.g., `EBD`, `HIT`). Based on our extensive case studies by code inspection, a key reason for these differential effects is that `FFI` and `IMI` allow for immediate data interoperations at language boundaries, via code-level (explicit/implicit) data/control flows induced by direct function calls. Such high data interoperability promises for programming ease but welcomes additional surfaces for stealthy attacks. In contrast, these are not readily possible with `EBD` or `HIT`. Interestingly, `EBD` seemed to be not only more secure than average by itself, it also appeared to help mitigate the insecurity of more vulnerable mechanisms (e.g., `FFI` versus `FFI EBD`, `IMI` versus `IMI EBD`). This is likely because its use helped reduce the amount of vulnerable cross-language data exchanges that would otherwise be fully handled by the vulnerable mechanisms.

Both explicit and implicit language interfacing mechanisms were strongly associated with code vulnerability proneness overall; FFI and IMI were the most vulnerability-prone mechanisms.

3.2.3 Topic Traits of Interfacing Effects. To understand the effects on specific vulnerability categories, we built a separate NBR model for each category, using the same predictors as in §3.2.2 but the number of vulnerability-fixing commits only belonging to that category as a response. Table 7 lists the details of the three models.

The results show that the three overall most vulnerable mechanisms were strongly associated with each of the two dominating vulnerability categories (cf. Table 1), which largely explained the overall association strength. Meanwhile, among these three, both `FFI` and `FFI IMI` actually had less-than-average proneness to *Insecure interaction* vulnerabilities. The reason is because these vulnerabilities were typically associated with Web languages (cf. §3.1.2), which do not interact through `FFI`. This explanation was consolidated by the positive association between this vulnerability category and `EBD`, the dominant mechanism of interfacing across Web languages (e.g., `javascript php`). The same can also explain the strong effect of `FFI EBD` on this category. To verify, we manually sampled and examined 10 projects that use {`FFI EBD`} and found that 7 of them used the language selection {`c c++ css html javascript`}—the domination of Web languages caused *insecure interactions*.

Proneness to insecure interaction vulnerabilities was most strongly associated with using EBD but the strength was counteracted by using IMI; The most overall-vulnerability-prone mechanisms (FFI, IMI) were most prone to porous defense vulnerabilities.

3.2.4 Effects of Individual Languages. We performed the same NBR analysis as in §3.2.2, with (1) the model predictors changed to individual languages among the studied selections (1st column of Table 4), and (2) every data point changed to k points each for one of the k languages selected in the project but with the same project vulnerability proneness value. Table 8 shows the model details. Overall, the associations of individual languages were strong, even stronger than those of language selections, with the proneness to both vulnerabilities in general and the three specific vulnerability categories. The most prone languages were `java`, `c`, `go`, and `c++`.

However, put together with Tables 4 and 5, the results show that the proneness of these individual languages had generally *no* consistent correlation with that of language selections. In some cases, the language selections (e.g., `c python`) and some of the selected individual languages (e.g., `c`) were consistently prone or not. Yet in more cases, language selections (e.g., `c c++ python`) that include one or more strongly prone languages (e.g., `c` and `c++`) were not significantly prone at all (or even negatively prone). Also, there are language selections (`c python`, `c++ python`, `c c++ shell`) that have much stronger proneness than any of the individual languages selected.

On the other hand, contrasting Tables 6, 7, and 8 revealed mostly consistent correlation in the proneness between language selections and the corresponding interfacing mechanisms. For instance, the proneness of `c c++ shell` and that of the interfacing mechanism most commonly used in this selection `IMI` were consistently strong. For another example, the proneness of `html ruby` and `html java` was negative, which is consistent with the negative proneness of `EMD`, the most commonly used interfacing mechanism in those selections.

Table 8: *NBR* coefficients on the proneness of individual languages. ***marks: ***p<0.001, **p<0.01, *p<0.05***

Independent factors	Overall		Porous defense		Risky resource mgmt.		Insecure interaction	
	Coeff.	Std. Err.	Coeff.	Std. Err.	Coeff.	Std. Err.	Coeff.	Std. Err.
(Intercept)	1.2455	0.074 ***	-2.2314	0.121 ***	-0.4693	0.102 ***	-3.5789	0.224 **
`html`	-0.0747	0.044 *	0.2049	0.067	-0.1473	0.063 *	0.7461	0.133 *
`javascript`	0.0451	0.057 ***	0.1206	0.077	-0.1983	0.077 *	0.5609	0.158 ***
`shell`	0.1329	0.037	0.0596	0.061	0.1935	0.052 *	0.1132	0.114
`ruby`	0.2004	0.069 **	0.7881	0.104 *	-0.4241	0.104	1.3293	0.186 ***
`go`	0.4894	0.086 **	0.9371	0.127 *	0.9696	0.113 ***	0.2514	0.293
`java`	0.6584	0.065 ***	1.0520	0.090 ***	0.6849	0.087 ***	0.6879	0.190 ***
`css`	0.0074	0.048	-0.0615	0.075	-0.1004	0.071	0.1032	0.141
`python`	0.0531	0.045 *	0.3908	0.073	-0.2044	0.061 *	0.7386	0.134 **
`c`	0.5444	0.061 ***	-0.1515	0.093	1.1080	0.078 ***	0.1369	0.193
`c++`	0.4271	0.059 ***	-0.0018	0.090	0.7112	0.075 ***	-0.7995	0.195 *
`php`	0.2480	0.074 ***	0.9188	0.100 ***	-0.4876	0.109 *	2.1335	0.191 ***
`objective-c`	0.3827	0.131 *	-0.6488	0.295 *	0.5943	0.189 **	-1.8621	0.644 *
AIC	*31056*		*5202.8*		*17879.4*		*5200.4*	
BIC	*31040.29*		*5187.09*		*17863.69*		*5184.69*	
Log Likelihood	*-15516*		*-2589.4*		*-8927.7*		*-2588.2*	
Deviance	*5615.3*		*1808.3*		*5672.7*		*2141.0*	

Table 9: Sampled projects and numbers of confirmed vulnerabilities (#Vul) in each, by language selection (LangSel, left table) and interfacing mechanism (InterMech, right table)

LangSel	Projects	#Vul	
`c c++ shell`	Pencil [68]	6	15
	Ncmpcpp [59]	5	
	Proxysql [84]	3	
	Fontforge [32]	1	
	Lnav [82]	0	
`c python`	Lily [29]	6	9
	RediSearch [77]	2	
	Pillow [71]	1	
	Osmc [66]	0	
	Phoenix [23]	0	
`java shell`	Ehcache3 [24]	5	6
	Elassandra [25]	1	
	Quasar [87]	0	
	Smali [37]	0	
	Siddhi [81]	0	

InterMech	Projects	#Vul	
`FFI`	Angr [8]	6	13
	RestKit [78]	5	
	Moderngl [57]	1	
	Printrun2 [44]	1	
	SDMongoDB [20]	0	
`FFI_IMI`	Mysql5.6 [26]	5	8
	Brackets-shell [6]	2	
	Openbr [64]	1	
	UPX [88]	0	
	VisPy [90]	0	
`IMI_EBD`	UI-Grid [9]	2	4
	Flask-Admin [31]	1	
	Wormhole [51]	1	
	Securedrop [33]	0	
	Statamic3 [83]	0	

The proneness of language selections was generally consistently correlated with that of the underlying interfacing mechanisms, but not consistently with that of the individual languages selected.

3.3 RQ3: Real-World Multilingual Vulnerability

Our results for RQ1 and RQ2 revealed (1) strong *statistical* relevance of language selection to vulnerability proneness of multilingual code and (2) strong *statistical* contribution of language interfacing mechanism to that relevance. Now questions remain: (a) do those general, statistical findings hold in *concrete* multilingual code? and (b) does the statistically strong proneness indicate *actual* vulnerabilities? and if so (c) what do they look like and how are they *induced by* the language selections and interfacing mechanisms?. To answer these, we conducted two large-scale manual studies: one to (1) *concretize/validate statistical findings* from RQ1 and RQ2, while the other to (2) *demonstrate/explain actual vulnerabilities* in multilingual code indicated by the statistical proneness.

3.3.1 Concretize/Validate Statistical Findings. We randomly chose 3 top language selections and randomly sampled 5 projects for each, as listed in Table 9 (left). Then, we randomly sampled 100 commits for each project and looked into each commit to confirm if it was indeed for fixing an *actual* vulnerability. The last two columns of the table show the number of confirmed vulnerabilities (#Vul) per project and the total for the language selection. The per-project #Vuls are sorted to ease comparisons across language selections.

As per our statistical numbers (Table 4), the selection c c++ shell was noticeably more vulnerability-prone than the selection c python (coefficient 0.7641 vs. 0.6253, with the same level of significance strength—p<0.001). Now per the numbers of actual vulnerabilities found, the c c++ shell projects were indeed noticeably more vulnerable than the c python projects in our random samples. Similarly, the statistical finding that java shell was the least prone to vulnerabilities (coefficient 0.2766 with p<0.01) among the three chosen selections is also consistent with our case study results here (the java shell projects sampled had the least #Vuls).

With a similar procedure but starting with 3 randomly chosen interfacing mechanisms, we manually studied another random set of 5 projects per mechanism, as listed in Table 9 (right). Per Table 6, FFI (coefficient 0.2817 with p<0.01) was more vulnerability-prone than FFI_IMI (coefficient 0.1676 with p<0.5). This is consistent with the actual vulnerabilities found in our samples—the projects using FFI had generally more vulnerabilities than those using FFI_IMI. The least statistical proneness of IMI_EBD (coefficient -0.3059 with p<0.5) was also confirmed by the fewest vulnerabilities overall in our sampled projects using this interfacing mechanism.

Our statistical findings on the differential vulnerability proneness of both language selection (RQ1) and interfacing mechanism (RQ2) were validated by (consistent with) the magnitude of actual vulnerabilities in concrete multilingual code in our extensive case studies.

3.3.2 Demonstrate/Explain Actual Vulnerabilities. We observed wide presences of cross-language vulnerabilities in our dataset. For space limits, we demonstrate that in 4 cases that cover different language selections and interfacing mechanisms. In the per-case figures, arrow-dashed lines show the vulnerable information flows, of which cross-language ones are in red; the associated (vulnerability-fixing) commits are indicated by the code diffs with deleted lines grayed.

Case 1 (Figure 3). Revealed via commit [72], the TIFF image is the user input, taken (line 1) and loaded (line 5) in python and then decoded (line 11) eventually, via FFI (offered with python's c extension [74]), by _decodeStrip in c (line 20). Within this c function, the incorrect boundary check (line 27) led to a *buffer over-read* vulnerability at line 32. The check should ensure the real row size of the TIFF image (row_byte_size) is less than the expected (calculated) size (unpacker_row_byte_size), as fixed by the commit (line 28). This vulnerability is *explained by* the selection of c, which is memory-unsafe, and the use of interfacing mechanism FFI, which propagates the python data to the unsafe memory operation in c.

Case 2 (Figure 4). Found via commit [60], an *insecure string copy* vulnerability was induced by FFI (offered with native abstractions for Nodejs [5]) also but in a different language selection c++ javascript. In javascript, testPort was taken as the user input (line 1). At line 3, a SerialPort object was initialized; then the c++ function open (defined at line 20) was called (line 4). Eventually, the vulnerability occurred at line 15, where the string format specifier (%s) was missing in the c++ function sprintf, causing the insecure string copy vulnerability across the two languages here.

Case 3 (Figure 5). The commit [14] led us to a *cross site request forgery* (CSRF) across 3 languages: twig interfaced with javascript via the EBD mechanism, while javascript and php exchanged data (a token) over the network (i.e., via the IMI mechanism). The vulnerability was fixed by retrieving and validating the token (line 3)

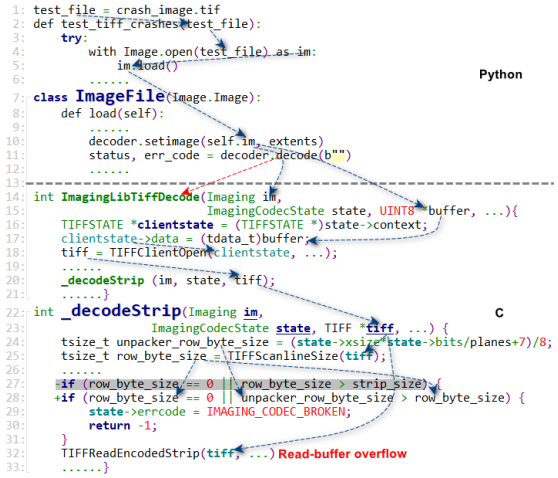

Figure 3: Case 1–buffer over-read across c python.

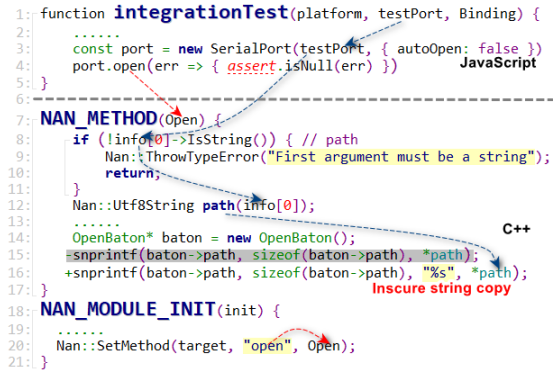

Figure 4: Case 2–insecure string copy across c++ javascript.

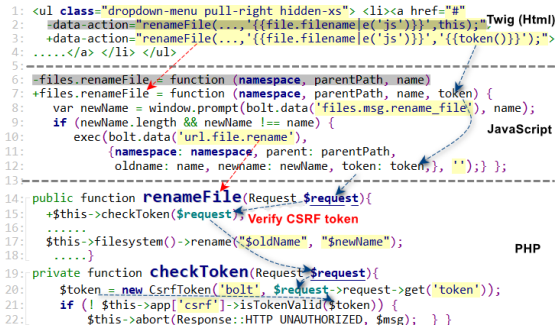

Figure 5: Case 3–CSRF across twig javascript php.

for a file operation (renaming). The token was integrated into the javascript function renamefile, which was embedded in twig (html) at line 3. Then, the definition of renamefile was modified accordingly in javascript (line 7), including adding parameters and obtaining the token. In php, the token was extracted from the request (line 20) and verified (line 21). If the token is invalid, the server would refuse the request and abort (line 22).

Case 4 (Figure 6). The commit [28] fixed a *NULL pointer dereference* vulnerability explained by using the lily language and c that interfaced via the HIT mechanism. A code snippet in lily was passed to an interpreter t at lines 5-9, where the variable

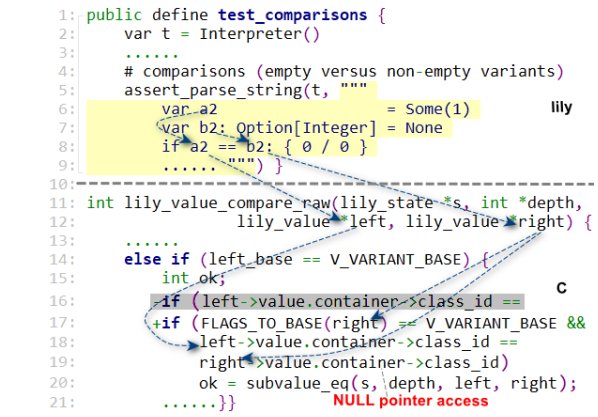

Figure 6: Case 4–NULL pointer dereference across `lily` `c`.

`b2` was initialized as `None`. When `b2` reached line 8, the interpreter implemented in `c` tried to decode its class id at line 19 (`right->value.container->class_id`). However, the container pointer was NULL when `b2` was None, causing the NULL pointer dereference vulnerability. To fix this issue, the interpreter added a validation check at line 17 before dereferencing the pointer.

Cross-language vulnerabilities were widely present in the studied multilingual code and were all explainable through the language selection and interfacing mechanisms among the selected languages.

4 RELATED WORK

Study of quality effects of languages. Recent studies have looked at the fault proneness of design smells in multi-language software [4] and how inter-language dependencies affect code quality [35]. Yet they were both limited to one particular language selection (Java-C) and a small set (only 10 or less) sample projects. The study by Abidi *et al.* [3] revealed that understandability is the most impacted quality attribute in a multi-language system based on the perceptions of 93 developers. Another study provided empirical evidence that most developers had encountered at least one bug related to cross-language linking and that the use of multiple languages increased the difficulty of bug fixing [54]. A few other relevant studies [76, 93] are on individual languages rather than language combinations. Our work differs in addressing the security impact of language selection in real-world multi-language projects. On the other hand, it would be interesting future work to study the vulnerability proneness of *single-language* projects—prior works [10, 45] examined single-language code's proneness to defects in general (including security defects) but did not particularly investigate the proneness to security vulnerabilities. These kinds of results can be compared with and complement ours.

Analysis of language interactions. Bissyandé *et al.* measured the closeness between different languages that informs about language interoperability [13]. Similar results were also obtained through an assessment of polyglotism, which indicated strong relationships between languages and revealed the sets of languages that tend to be used together in practice [53, 86]. Our study also concerns the interactions between languages but indirectly yet more deeply, as we looked at languages in selections. Our work also differs by looking beyond the companionship of languages, into the security rationales underlying the language interactions (via interfacing).

Commit-based vulnerability identification. Like our VC technique, prior works also explored identifying vulnerabilities in code repositories through analysis of commits. D2A [94] uses static analyzers and learning-based classifiers to detect vulnerability-fixing commits. Another relevant tool is VCCFinder [70], which uses code metrics and GitHub metadata as features to train a SVM classifier to label a commit as vulnerability-contributing or not. The tool in [96] uses features based on commit messages and bug reports to classify a commit or report as vulnerability-related or not. Our approach is different and tended to be more accurate (83% accuracy versus: 53% by D2A, 36% by VCCFinder, 50% by [96] for commit classification).

Real-world vulnerability datasets. We contribute a dataset on vulnerability-fixing commits, which is similar to D2A [94] in nature but much larger in size (ours 141,380 versus their 18,653 such commits) with much (30%) greater accuracy. Meanwhile, our dataset complements to CVE-based datasets such as BigVul [27] and CVEfixes [11], which are much smaller and for C/C++ code only.

5 DISCUSSION

5.1 Tools and Dataset

Despite the overwhelming and growing dominance of multilingual systems [49, 54, 86] in real-world software domains, tool support for understanding and analyzing these systems is lacking. Beyond our empirical findings, we contribute two novel tools: VC (§2.3) for automatically identifying/classifying vulnerability-fixing commits, and Lice (§2.4) for automatically identifying/classifying language interfacing mechanisms [48]. Both tools offer promising accuracy; they not only enabled our study but also will empower future characterization studies and tool development for multi-language software. In particular, VC *outperformed* the state-of-the-art peer tools (as discussed in §4), while Lice is the *first* of its kind based on our *novel/first taxonomy* of language interfacing mechanisms.

Importantly, using VC, we contribute an important, much-needed, sizable dataset: a set of 141.38K vulnerability-fixing commits, along with the repository information, associated issue ids, and bug tagging whenever available. Beyond supporting relevant empirical studies, this dataset can boost the performance of machine learning (ML), especially deep learning (DL) [63], based vulnerability analysis (e.g., detection [95], repair [39]) tools. Extant such tools suffer poor performance on real-world datasets [95] mainly because the ML/DL models were only trained on largely artificial datasets which do not well represent realistic vulnerabilities and codebases—existing realistic datasets (e.g., [11, 27]) are small while existing sizeable datasets are only artificial. Our commit dataset corresponds to 141.38K pairs of real-world vulnerable samples and their fixed versions, which can serve as a large-scale, realistic dataset to train DL/ML models hence improve their performance against real-world software. As discussed in §4, ours is much larger and of higher quality (due to VC's higher accuracy) than the most recent peer dataset.

Lice can also be used beyond our studies. For instance, with the interfacing mechanism it reports, it can guide the development of cross-language vulnerability detection [50] techniques. Interfacing-mechanism-specific detection rules/algorithms can be then devised, which would be more accurate than a universal/generic technique handling multilingual code of arbitrary interfacing mechanisms.

5.2 Implications

5.2.1 Attention to Cross-Language Vulnerabilities. We found strong vulnerability-proneness of multilingual code (§3.1,§3.2). Our extensive case studies further confirmed that the proneness did indicate actual vulnerabilities (§3.3). Yet, existing works on these vulnerabilities are rare. For example, after checking 1,000+ papers on program analysis randomly chosen from those published in the past 5 years in the ACM digital library, we found only 0.8% of these are *generally* related to cross-language analysis, and these 0.8% almost [58] exclusively targeted a particular language selection `c java` [4, 40, 46, 91]. However, `c java` was not even in our top-20 language selections. We suggest that our community starts to attend to this severe gap and urgent issue by investing in tools and studies on cross-language defects while going beyond Java-C (JNI) software.

5.2.2 Practical Recommendations for Researchers. While both based on GitHub, earlier studies [10, 76] found little effect of *individual languages* on software proneness to defects, whereas ours revealed strong effects of language selection on the vulnerability proneness of multilingual code (§3.1). We further found (§3.2) that these effects were generally correlated with those of language interfacing mechanisms, rather than with those of individual languages. Thus, we suggest researchers examine and *utilize* language interfaces when addressing multilingual code security.

We found FFI and IMI as the most vulnerability-prone mechanisms for language interfacing. Both of them allow for direct data interoperations via code-level invocations (explicit or implicit). We suggest to prioritize on multilingual code using these mechanisms, addressing not only explicit cross-language interactions (e.g., via foreign/native functions) but also implicit ones (e.g., via RPC).

5.2.3 Practical Recommendations for Tool Builders. In particular for those to build tools for multilingual security, we suggest they leverage ML/DL models in addition to program analysis approaches. ML/DL models can be built using attributes of language selections (e.g., size, properties like types of each selected language) and those of interfacing (e.g., mechanism category, interface parameters) as learning features, while leveraging our large, realistic vulnerability dataset for training and validation. Meanwhile, given the diversity of language selection and interfacing, we suggest code-analysis-based approaches focus on *language-independent* analyses (e.g., [12])—developing a language-selection-specific analysis like the ones for Java-C is not sufficient [92]. One specific approach is to compute *cross-language information flow* to identify multilingual vulnerabilities, as we demonstrated in our case studies (§3.3).

5.2.4 Practical Recommendations for Developers. Our results clearly indicate that some language selections (e.g, `c python`) were significantly more prone to vulnerabilities than others (§3.1) and the proneness did indicate real vulnerabilities (§3.3), so were some interfacing mechanisms (e.g., FFI, IMI) than others (§3.2). We suggest developers choose less prone selections and mechanisms, if possible, to make the multilingual code potentially more secure. Moreover, we found that it was not uncommon that different kinds of interfacing mechanisms were used for the same language selection. Thus, if a language selection must be chosen, we suggest to still choose less vulnerability-prone mechanisms for interfacing among the chosen languages if possible. Finally, based on Tables 5 and 7, we suggest to make choices of the selections and mechanisms that are less prone to the *categories* of vulnerabilities that are most concerning.

5.3 Threats to Validity

The validity of our results was affected by the inaccuracy of our Lice tool used for language interfacing classification. Moreover, the tool was built on our manual summaries of interfacing patterns among the top 12 mainstream languages, and it was applied in our study around the manually derived taxonomy. And the technique was evaluated only against a small sample. Biases and errors during such manual processes may have caused greater inaccuracy in our study results than what we reported. A similar threat applies to our tool for commit-based vulnerability categorization and the manual process of evaluating its accuracy. In addition, the correctness of both tools is limited by the accuracy of the underlying tools used (NLTK, Regex, and *FuzzyWuzzy*). To mitigate these threats, we followed a cross-validation procedure by which the results were approved by the three authors involved. We also chose to sample *randomly* in the manual evaluation to reduce the associated threat.

Relative to the population of multi-language software, our datasets only represent small samples. Thus, our findings and conclusions are best interpreted for the sampled projects, and we cannot claim that our results generalize to any other projects. To reduce this threat, we used a reasonably large dataset and applied multiple project filters along with random sampling to reduce data noises. Meanwhile, we cannot rely on CVEs due to the small dataset size.

In addition, we did not differentiate languages of different classes (data modeling versus programming languages) and treated all languages equally in calculating code sizes in bytes. We actually intended not to limit our study to a particular language class—note that non-programming languages can play critical roles in code vulnerabilities too (e.g., the `twig` templating language contributed to the vulnerability in Figure 5); We also did not rule out any particular kind of language, because we do not have a-priori knowledge regarding which kinds must not be relevant to vulnerabilities. Yet this treatment may have caused biases in our results interpretations. The validity of most of our results is also affected by the inaccuracy of GitHub linguist [2] we used for retrieving language selections.

6 CONCLUSION

We presented a large-scale study on the vulnerability proneness of multilingual code, enabled by a novel taxonomy of language interfacing mechanisms, two novel/dedicated tools, and new/sizable datasets. From 4,001 GitHub projects and 20.37 million commits in their repositories, we revealed strong statistical relevance of language selection and interfacing mechanism to the proneness, and even stronger ones with respect to specific categories of vulnerabilities. We further conducted *extensive* manual inspection studies of 50 sample projects with 500 commits for each, both randomly sampled, hence validated our statistical findings and demonstrated actual multilingual vulnerabilities indicated by the proneness. Based on our results, we provided insights and actionable suggestions on addressing multilingual vulnerabilities for various stakeholders.

ACKNOWLEDGMENT

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

REFERENCES

[1] 2020. GitHub: a US-based global company, provides hosting for software development version control using Git. https://github.com/.

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

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

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

[5] Adilapapaya. 2020. Native Abstractions for Node.js. http://adilapapaya.com/docs/nan/.

[6] Adobe. 2021. The CEF3-based application shell for Brackets. https://github.com/adobe/brackets-shell.

[7] Paul D Allison and Richard P Waterman. 2002. Fixed–effects negative binomial regression models. *Sociological methodology* 32, 1 (2002), 247–265.

[8] angr. 2021. A platform-agnostic binary analysis framework. https://github.com/angr/angr.

[9] AngularUI. 2021. An AngularJS data grid. https://github.com/angular-ui/ui-grid.

[10] 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.

[11] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.

[12] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120.

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

[14] bolt. 2021. A CSRF issue fixing commit in bolt. https://github.com/bolt/bolt/commit/ca49f43934b5381ca15e4e8f13e74b9aa6cec7ea.

[15] Francesca Del Bonifro, Maurizio Gabbrielli, and Stefano Zacchiroli. 2021. Content-Based Textual File Type Detection at Scale. In *2021 13th International Conference on Machine Learning and Computing*. 485–492.

[16] Yaofei Chen, Rose Dios, Ali Mili, Lan Wu, and Kefei Wang. 2005. An empirical study of programming language trends. *IEEE software* 22, 3 (2005), 72–79.

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

[18] PoC Consul and Felix Famoye. 1992. Generalized Poisson regression model. *Communications in Statistics-Theory and Methods* 21, 1 (1992), 89–109.

[19] Harald Cramir. 1946. Mathematical methods of statistics. *Princeton U. Press, Princeton* 500 (1946).

[20] Spring Data. 2021. Spring Data MongoDB. https://github.com/spring-projects/spring-data-mongodb.

[21] Daniel P Delorey, Charles D Knutson, and Christophe Giraud-Carrier. 2007. Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects. In *Second International Workshop on Public Data about Software Development (WoPDaSD'07)*.

[22] Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Phuong T Nguyen. 2020. A Multinomial Naïve Bayesian (MNB) Network to Automatically Recommend Topics for GitHub Repositories. In *Proceedings of the Evaluation and Assessment in Software Engineering*. 71–80.

[23] Robin Dunn. 2021. wxPython Project Phoenix. https://github.com/wxWidgets/Phoenix.

[24] ehcache. 2021. Ehcache3:Java's most widely-used cache. https://github.com/ehcache/ehcache3.

[25] elassandra. 2021. Elassandra:an Apache Cassandra distribution. https://github.com/strapdata/elassandra.

[26] Facebook. 2021. MySQL Server 5.6. https://github.com/facebook/mysql-5.6.

[27] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[28] FascinatedBox. 2021. A NULL pointer access fixing commit in lily. https://api.github.com/repos/FascinatedBox/lily/commits/7c848602989f59be50f30e096135c520d086650a.

[29] FascinatedBox. 2021. A programming language focused on expressiveness and type safety. https://github.com/FascinatedBox/lily.

[30] Jason Firch and Josh Allen. 2021. Cyber Security Trends You Can't Ignore In 2021. https://purplesec.us/cyber-security-trends-2021/.

[31] flask admin. 2021. A batteries-included, simple-to-use Flask extension. https://github.com/flask-admin/flask-admin.

[32] fontforge. 2021. A free (libre) font editor. https://github.com/fontforge/fontforge.

[33] freedompress. 2021. An open-source whistleblower submission system. https://github.com/freedomofpress/securedrop.

[34] 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 21)*. 2093–2110.

[35] 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* (2020).

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

[37] Ben Gruver. 2021. An assembler/disassembler for the dex. https://github.com/JesusFreke/smali.

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

[39] Jacob A Harer, Onur Ozdemir, Tomo Lazovich, Christopher P Reale, Rebecca L Russell, Louis Y Kim, and Peter Chin. 2018. Learning to repair software vulnerabilities with generative adversarial networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 7944–7954.

[40] 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.

[41] Capers Jones. 2009. *Software engineering best practices*. McGraw-Hill, Inc.

[42] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.

[43] 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.

[44] kliment. 2021. The master branch holds the development of Printrun 2.x. https://github.com/kliment/Printrun.

[45] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. 2016. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 563–573.

[46] 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.

[47] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: memory leak detection using partial call-path analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1621–1625.

[48] 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)*.

[49] 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.

[50] 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.

[51] magic wormhole. 2021. A library and a command-line tool. https://github.com/magic-wormhole/magic-wormhole.

[52] Bob Martin, Mason Brown, Alan Paller, Dennis Kirby, and Steve Christey. 2011. 2011 CWE/SANS top 25 most dangerous software errors. https://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf. (2011).

[53] 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.

[54] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development* 5, 1 (2017), 1.

[55] Leo A Meyerovich and Ariel S Rabkin. 2013. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 1–18.

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

[57] ModernGL. 2021. A python wrapper over OpenGL 3.3+ core. https://github.com/moderngl/moderngl.

[58] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *Static Analysis Symposium (SAS)*.

[59] ncmpcpp. 2021. A NCurses Music Player Client. https://github.com/ncmpcpp/ncmpcpp.

[60] node serialport. 2021. A insecure string copy fixing commit in node-serialport. https://github.com/serialport/node-serialport/commit/a250f0983c2ca9b2b0fd5d9a00e1d88b0e8d2480.

[61] Yu Nong and Haipeng Cai. 2020. A preliminary study on open-source memory vulnerability detectors. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 557–561.

[62] Yu Nong, Haipeng Cai, Pengfei Ye, Li Li, and Feng Chen. 2021. Evaluating and comparing memory error vulnerability detectors. *Information and Software Technology* 137 (2021), 106614.

[63] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating Realistic Vulnerabilities via Neural Code Editing: An Empirical Study. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[64] OpenbiometricsOrg. 2021. Open Source Biometric Recognition. https://github.com/biometrics/openbr.

[65] ORACLE. 2020. Java Native Interface Specification Contents. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html. (2020).

[66] OSMC. 2021. Open Source Media Center. https://github.com/osmc/osmc.

[67] Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. 2016. User-perceived source code quality estimation based on static analysis metrics. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 100–107.

[68] Pencil2D. 2021. An animation/drawing software. https://github.com/pencil2d/pencil.

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

[70] 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.

[71] Pillow. 2021. The Python Imaging Library. https://github.com/python-pillow/Pillow.

[72] Pillow. 2021. A read buffer overflow fixing commit in Pillow. https://github.com/python-pillow/Pillow/commit/87934e22d056cb72ad6c7e9dc48e06d2a02e2dec.

[73] Python. 2020. Extending Python with C or C++. https://docs.python.org/3/extending/extending.html. (2020).

[74] Python. 2020. Python 3.9.2 documentation. https://docs.python.org/3.9.

[75] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 432–441.

[76] 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.

[77] RediSearch. 2021. The querying, secondary indexing, and full-text search for Redis. https://github.com/RediSearch/RediSearch.

[78] RestKit. 2021. A Objective-C framework for implementing RESTful web services clients. https://github.com/RestKit/RestKit.

[79] Simone Romano, Maria Caulo, Matteo Buompastore, Leonardo Guerra, Anas Mounsif, Michele Telesca, Maria Teresa Baldassarre, and Giuseppe Scanniello. 2021. G-Repo: a Tool to Support MSR Studies on GitHub. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 551–555.

[80] Hossain Shahriar and Mohammad Zulkernine. 2011. Injecting comments to detect JavaScript code injection attacks. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*. IEEE, 104–109.

[81] siddhi io. 2021. A cloud native Streaming and Complex Event Processing engine. https://github.com/siddhi-io/siddhi.

[82] Tim Stack. 2021. LNAV – The Logfile Navigator. https://github.com/tstack/lnav.

[83] Statamic. 2021. The flat-first, Laravel + Git powered CMS. https://github.com/statamic/cms.

[84] Sysown. 2021. The proxy for MySQL and forks. https://github.com/sysown/proxysql.

[85] Gang Tan and Greg Morrisett. 2007. ILEA: Inter-language analysis across Java and C. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*. 39–56.

[86] Federico Tomassetti and Marco Torchiano. 2014. An empirical assessment of polyglot-ism in github. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. 1–4.

[87] Parallel Universe. 2021. Fibers, Channels and Actors for the JVM. https://github.com/puniverse/quasar.

[88] UPX. 2021. An advanced executable file compressor. https://github.com/upx/upx.

[89] Bogdan Vasilescu, Alexander Serebrenik, and Mark GJ van den Brand. 2013. The Babel of software development: Linguistic diversity in Open Source. In *International Conference on Social Informatics*. Springer, 391–404.

[90] vispy. 2021. A high-performance interactive 2D/3D data visualization library. https://github.com/vispy/vispy.

[91] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1137–1150.

[92] 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)*.

[93] Jie Zhang, Feng Li, Dan Hao, Meng Wang, Hao Tang, Lu Zhang, and Mark Harman. 2019. A Study of Programming Languages and Their Bug Resolution Characteristics. *IEEE Transactions on Software Engineering* (2019).

[94] 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.

[95] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *arXiv preprint arXiv:1909.03496* (2019).

[96] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 914–919.