2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

A Large Scale Study of Multiple Programming Languages and Code Quality

Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo
School of Information Systems
Singapore Management University
{kochharps.2012,dwijedasa,davidlo}@smu.edu.sg

***Abstract*—Nowadays, most software use multiple programming languages to implement certain functionalities based on the strengths and weaknesses of different languages. Researchers in the past have studied the impact of independent programming languages on software quality, however, there has been little or no research on the impact of multiple languages on the quality of software. Does the use of multiple languages cause more bugs? Are certain languages when used with other languages make software more bug prone? What are the relationships between multi-language usage and various bug categories?**

In this study, we perform a large scale empirical investigation to provide some answers to these questions. We gather a large dataset consisting of popular projects from GitHub (628 projects, 85 million SLOC, 134 thousand authors, 3 million commits, in 17 languages) to understand the impact of using multiple languages on software quality. We build multiple regression models to study the effects of using different languages on the number of bug fixing commits while controlling for factors such as project age, project size, team size, and the number of commits. Our results show that in general implementing a project with more languages has a significant effect on project quality, as it increases defect proneness. Moreover, we find specific languages that are statistically significantly more defect prone when they are used in a multi-language setting. These include popular languages like C++, Objective-C, and Java. Furthermore, we note that the use of more languages significantly increases bug proneness across all bug categories. The effect is strongest for memory, concurrency, and algorithm bugs.

Keywords*—*multiple programming languages, code quality, bug fix commits, multiple regression models

I. Introduction

There is a large number of programming languages available to develop different kinds of software — the Wikipedia Encyclopedia lists about 700 languages[1]. They include object-oriented languages (e.g., Java, C++), functional languages (e.g., Clojure, Haskell), procedural languages (e.g., C, Go), and many more, each with its own advantages and disadvantages. Some languages have been used for a long period of time (e.g., C), whereas others have only been around for few years (e.g., Go). The wide variety of programming languages gives developers a plethora of options to choose from.

Developers often leverage the strengths of multiple languages to cope with challenges of building complex software. By using languages that complement one another, performance, productivity, and agility may be improved. For example, to build web applications developers often use a server side language such as Perl, PHP or Python and client-side language like JavaScript. Also, many popular software available in the market are implemented in multiple languages. For example, the Linux[2] operating system is built using[3] kernel written in C with parts such as utilities and applications developed in C++, Perl and Python. Similarly, OpenCV[4], which is an open-source computer vision library, is developed using a set of languages including C++, C, Python, Java and JavaScript. Although using multiple languages has advantages, unfortunately it also raises a number of issues related to the increase in the complexity of the software and the need for proper interfaces between different languages. These issues may in turn translate to software quality problems.

Past studies have studied characteristics of programming languages by looking into a large number of projects in open repositories. For example, Bissyande et al. studied popularity, interoperability, and impact of programming languages by analyzing a large number of open-source projects hosted on GitHub [1]. They identified popular languages that are used to implement many projects, languages that are often used together, and correlations between individual language usage with project success (measured in terms of number of forks and watchers), number of issue reports, and team size. In a later work, Ray et al. studied a large number (i.e., hundreds) of popular projects from GitHub to analyze the effect of individual language usage on the number of bug fixing commits [2]. Different from Bissyande et al.'s work, they looked into bug fixing commits instead of issue reports, which are often scarce for projects hosted in GitHub [3]. They use negative binomial regression (NBR) to analyze the relationships between different languages and number of bug fixing commits. They found that some languages have significant relationships with bug proneness. Moreover, compared to procedural or scripting languages, functional languages had weaker relationships to bug proneness. Unfortunately, despite this recent interest in analyzing characteristics of programming languages from a large number of projects in open repositories, to the best of our knowledge, there has been no large-scale study on the impact of the usage of *multiple* languages on bug proneness.

To fill this gap, we analyze a large number of popular repositories in GitHub and extract information such as languages used and number of bugs. We run a parser that supports

[1]https://en.wikipedia.org/wiki/List_of_programming_languages

[2]https://www.linux.com/

[3]In this study, we define a project uses a language if the project contains parts of its code written in that language.

[4]http://opencv.org/

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.112

a large number of languages to identify the languages used and the extent to which they are used. Similar to Ray et al. [2], we detect occurrences of bug fixing commits by analyzing the commit logs. After all relevant pieces of information are extracted, we build several regression models to examine the relationship between multiple language usage and bug proneness, while controlling for factors such as project age, project size, team size, and number of commits.

We examine the following research questions:

RQ1: *Does the use of more languages correlate with higher bug proneness?*

RQ2: *Are some languages more bug prone when they are used with other languages?*

RQ3: *What are the relationships between multi-language usage and bugs of various categories?*

The contributions of our work are as follows:

1) We are the first to perform a large-scale study to analyse the impact of the usage of multiple languages in a project to its bug proneness.
2) We identify languages that are more bug prone when used in a multi-language setting.
3) We also identify the impact of multiple-language usage to bug proneness for bugs in various categories.
4) We release our dataset publicly to allow for others to replicate our findings and investigate other interesting characteristics of multiple-language usage.

The structure of the paper is as follows. In Section II, we elaborate the way we collect our dataset and its characteristics, along with the statistical methods that we use to analyze the data. In Section III, we answer the three research questions we listed above. We discuss implications in Section IV and threats to validity in Section V. Section VI describes related work. We conclude with future work in Section VII.

II. Methodology

In this section, we first present how we collected our dataset and some basic statistics of our dataset in Section II-A. We then present a semi-automated classification process that we follow to assign bug fixing commits to various categories in Section II-B. Next, we present the analysis method and answer our research questions in Section II-C.

A. Data Collection

To study whether the usage of multiple programming languages within a single project has an impact on code quality, we consider the same top 17 languages investigated by Ray et al. [2]. For each of the 17 languages, we select the top 50 projects *primarily* written in that language. We collect a new dataset because Ray et al. do not make their dataset public. Table I shows for each of the 17 languages, the top-3 most popular projects that are implemented primarily in the language. Our dataset consists of projects written in a single language as well as projects written in multiple languages. We elaborate the detail of the data collection process in the following paragraphs.

Table I: Top three projects in each language

Primary Language	Projects
C	linux, redis, php-src
C++	node-webkit, phantomjs, textmate
C#	corefx, mono, SignalR
Objective-C	AFNetworking, GPUImage, SDWebImage
Go	docker, build-web-application-with-golang, kubernetes
Java	elasticsearch, storm, SlidingMenu
CoffeeScript	atom, coffee-script, hubot
JavaScript	bootstrap, d3, node
TypeScript	bitcoin, dogecoin, litecoin
Ruby	rails, gitlabhq, jekyll
Php	laravel, CodeIgniter, yii
Python	flask, django, reddit
Perl	gitolite, showdown, dotfiles
Clojure	LightTable, clojurescript, leiningen
Erlang	otp, ejabberd, mochiweb
Haskell	pandoc, yesod, Haxl
Scala	PredictionIO, scala, akka

Retrieving popular projects. To retrieve projects primarily written in each language from GitHub, we use GitHub Archive[5]. GitHub Archive is a database that records the public GitHub timeline activities to provide easy access for further analysis. The archive stores more than 20 different event types corresponding to new commits, commenting, forking, pull requests, issue tracking, and adding developers to a project. The archive is available as a public dataset on Google BigQuery[6] and is updated automatically on an hourly basis. Google BigQuery enables users to run SQL-like queries over the entire dataset.

For each selected language, we retrieve a list of GitHub repositories *primarily written* in that language by using the Google BigQuery interface of GitHub Archive[7]. Then, we count the number of *stars* acquired by each repository by counting occurrences of events of type *WatchEvent*[8]. The number of stars is an indicator of the popularity of a project, because it relates to the number of people who are interested in a project as well as the number of forks for that project [4], [5]. Next, for each language, we download the top 50 projects having highest number of stars using the *git clone* command. We discard all projects with less than 256 commits, to ensure that the selected projects have sufficient development history. The value 256 is the first quartile commit count of the 850 projects. The filtering process leaves us with 628 projects for our study.

Analyzing project evolution history. Following Ray et al.'s methodology, for each of the 628 projects, we retrieve the commit logs of non-merge commits using the *git log –no-merges –numstat* command. We obtain the author, commit date, and descriptive message of each of the non-merge commits. In addition to these pieces of information, the command also lists the files that are changed by a commit, together with the number of added and deleted lines for each file. Based on the pieces of information from the commit log, we calculate the age of a project by taking the difference between the initial commit date and the latest commit date (in days), the size of a project by calculating the number of lines of code in it, the total number of non-merge commits, and the total number of

[5]https://www.githubarchive.org/
[6]https://cloud.google.com/bigquery
[7]GitHub Archive stores the primary language used in a project.
[8]https://developer.github.com/v3/activity/events/types

Table II: Basic statistics of the dataset. Number of Associated Languages = number of other programming languages used together with the corresponding primary language.

	Project Details					Total Commits		BugFix Commits	
Primary Language	Number of Projects	Number of Associated Languages	Number of Developers	SLOC (KLOC)	Period	Number of Commits	Code Churn (KLOC)	Number of Bug Fixes	Code Churn (KLOC)
C	104	9	25,203	17,350	4/1999 to 10/2015	780,153	229,411	258,822	42,736
C++	94	12	6,855	14,179	7/2000 to 10/2015	284,331	250,238	112,591	63,881
C#	46	8	4,818	8,848	6/2001 to 10/2015	235,048	138,719	69,974	15,786
Objective-C	38	4	2,319	378	7/2008 to 10/2015	30,048	16,617	7,561	1,031
Go	41	10	6,297	4,013	3/2008 to 10/2015	117,282	33,134	32,371	4,486
Java	84	10	5,456	5,004	3/2006 to 10/2015	156,153	105,063	58,922	21,412
CoffeeScript	44	3	3,930	564	12/2009 to 10/2015	80,613	25,016	14,267	2,630
JavaScript	245	13	10,536	3,596	3/2006 to 10/2015	123,500	45,256	33,498	6,254
TypeScript	42	8	4,941	16,792	5/2002 to 10/2015	188,698	167,050	51,970	18,531
Ruby	87	8	24,184	2,991	1/1998 to 10/2015	337,457	46,014	67,619	6,577
Php	59	4	12,145	2,959	4/2003 to 10/2015	218,751	77,268	73,700	10,221
Python	152	5	11,397	1,864	7/2005 to 10/2015	199,974	37,691	59,670	6,332
Perl	38	5	1,482	297	1/2004 to 10/2015	38,231	7,944	7,649	418
Clojure	46	5	2,171	432	4/2008 to 10/2015	41,588	5,650	6,994	511
Erlang	36	7	2,429	2,670	5/2001 to 10/2015	77,411	24,425	15,579	1,977
Haskell	37	5	4,158	1,053	4/2004 to 10/2015	128,149	21,834	24,558	4,612
Scala	42	8	5,698	2,319	2/2003 to 10/2015	133,754	70,807	31,044	24,287
Summary	628		134,019	85,309	1/1998 to 10/2015	3,171,141	1,302,137	926,789	231,682

developers who contribute code to each project. For project size, we do not consider the number of files but rather the number of lines of code as the size of files can vary a lot. We use these statistics as control variables in our regression model.

We also identify all bug fixing commits made in each project using the same list of error related keywords (i.e., 'error', 'bug', 'fix', 'issue', 'mistake', 'incorrect', 'fault', 'defect', 'flaw') used by Ray et al. [2]. To identify bug fixing commits, we check if the descriptive message in the log of each commit contains one of these keywords. Next, we calculate the total number of bug fix commits per project.

Identifying programming languages used. To identify the programming languages used to implement each project, we run the CLOC[9] tool on the latest revision of the project. CLOC is capable of counting the total number of lines of source code, blank lines and comment lines of files in a given repository. The tool can detect around 159 file extensions. For each project, we first run CLOC to obtain the list of languages used in the project, along with the number of lines of code written in each of the detected languages.

We also obtain the languages associated with each non-merge commit, based on the extensions of the changed files. For each project, we calculate the total number of commits per each detected language based on this information. Next, for each project, we exclude some of the detected languages, if the language has less than 11 commits. The value 11 is the first quartile of the total number of commits associated with each detected language per project. This filtering ensures that we consider only languages corresponding to significant activity within a given project. This leaves us with a dataset containing 297 projects written in a single language and 331 projects written in multiple languages.

Table II shows some basic statistics of our dataset. The second column of the table (i.e., Project Details) shows information of projects in our dataset that are written primarily in one of the 17 languages. The third column (i.e., Total Commits) shows the total number of commits, containing code written in each of the languages, along with the total number of lines of code churned (i.e., added, deleted, or modified) by those commits. The fourth column (i.e., Bug Fix Commits) corresponds to commits that are bug fixes and describes their numbers and the total number of lines of code churned by those commits, for each of the 17 languages. From the table, we can note that our dataset consists of 628 projects, 134 thousand developers, 3.17 million commits and 926,789 bug fixing commits.

Table III shows what are the other languages that are used together with each primary language considering projects written in multiple languages.

Table III: Languages used with each primary language

Primary Language	Languages Used with the Primary Language
C	Java, C++, Ruby, PHP, Python, Javascript, Perl, Objective-C, TypeScript
C++	Javascript, C, Java, Objective-C, Python, Ruby, CoffeeScript, PHP, Perl, Haskell, C#, TypeScript
C#	C++, C, Javascript, Ruby, Perl, Python, Objective-C, Java
Objective-C	Python, Javascript, C, Ruby
Go	C++, TypeScript, Javascript, Ruby, C, Python, Perl, Java, PHP, Objective-C
Java	C++, C, Python, Javascript, Clojure, Perl, Ruby, Objective-C, C#, Scala
CoffeeScript	Javascript, Ruby, Python
JavaScript	C#, PHP, C++, Clojure, CoffeeScript, Ruby, Erlang, Python, Haskell, Java, Go, TypeScript, Scala
TypeScript	C++, Java, Python, Javascript, CoffeeScript, C, Perl, PHP
Ruby	CoffeeScript, Javascript, Python, C++, C, Java, Perl, PHP
PHP	Javascript, Python, Perl, C
Python	Javascript, C, C++, CoffeeScript, PHP
Perl	Javascript, Java, Python, C++, C
Clojure	Javascript, Java, C, Objective-C, Ruby
Erlang	C, Python, Javascript, PHP, C++, Java, Clojure
Haskell	C, Javascript, Python, Perl, Java
Scala	Java, Ruby, Javascript, Perl, C#, Python, CoffeeScript, Clojure

B. Categorizing Bugs

Developers often leave important pieces of information in commit logs while fixing software bugs. Such important pieces

[9]http://cloc.sourceforge.net/

Table IV: Categories and distribution of bugs in the training and overall dataset

	Bug Type	Bug Description	Search Keywords/Phrases	Training Data (Count)	Training Training (%)	Overall Data **(Count)**	Overall Data **(%)**
Cause	Algorithm (Algo)	algorithmic and logical errors	algorithm	47	0.05	688	0.07
	Concurrency (Conc)	multi-threading or multi-processing related issues	deadlock, race condition, synchronization error, mutex, semaphore, starvation, locking, multiple threads	917	0.99	9,290	1.01
	Memory (Mem)	incorrect memory handling	memory leak, null pointer, buffer overflow, heap overflow, dangling pointer, double free, segmentation fault, segfault, space leak, dereference, memory corruption, memory overrun, heap overrun	3,434	3.74	34,320	3.74
	Programming (Prog)	generic programming errors	exception handling, error handling, type error, typo, compilation error, copy-paste error, refactoring, missing switch case, faulty initialization, default value, declaration, syntax, counter, signature, variable, regexp, cut-paste error, operator, inconsistent name, parameter, **argument, procedure**	86,620	94.25	864,645	94.10
Impact	Security (Sec)	correctly runs but can be **exploited by attacks**	buffer overflow, security, password, oauth, **ssl,vulnerability, attack**	2,916	3.17	25,692	2.80
	Performance (Perf)	correctly runs with delayed **response**	optimization problem, performance, latency, **speed, delayed, throughput**	1,537	1.67	13,750	1.50
	Failure (Fail)	crash or hang	reboot, crash, hang, restart, freeze	6,330	6.89	61,275	6.67
	Unknown (Unkn)	not part of the above seven categories		869	0.95	9,663	1.03

of information may include the root cause of the bug, how the bug affects the functionality of the software, and the fixes made to correct the bug. We use these pieces of information to categorize the identified bug fix commits of each project, taking a similar approach followed by Ray et al. [2].

Each bug fix commit is categorized based on its *cause* and *impact*. The cause and impact dimensions include several categories. The *cause* categories are: algorithmic, concurrency, memory, programming, and unknown. The *impact* categories are: security, performance, failure, and unknown. Each bug fix commit is assigned a cause and an impact category following a semi-automated classification process. For example, a commit with log message "*Fix a crash caused by uninitialized variable m_transport*" of the *TrinityCore*[10] C++ project corresponds to a bug caused by an uninitialized variable (i.e., a programming error) and the impact was a crash (i.e., a failure). Thus, this bug fix commit is assigned to the programming category (for cause) and failure category (for impact). The classification process is performed by following the two steps described below. In the first step, we iteratively match a small number of bug fixing commits against multiple lists of keywords (which gets manually refined at the end of each iteration) to get a training set of labeled commits. In the second step, we use a classifier built from the training data to label the rest of the bug fixing commits. We elaborate these two steps below.

Step 1: Keyword matching. We randomly select 10% of the total bug fixing commits (926,789) obtained from the 628 projects. We employ semi-automated way to classify each bug fixing commit using a set of keywords and phrases. For example, if a bug fixing commit log contains "deadlock", "race condition" or "synchronization error", we classify the commit as a concurrency bug. We classify each commit separately for its cause and impact category. Some commit logs may contain keywords from multiple cause and impact categories. Such cases are resolved by assigning the category that matches the maximum number of keywords in the logs. At the end, one category is assigned for cause and another for impact.

To reduce the number of commits with both cause and impact assigned to the unknown category, we iteratively increase the number of keywords and phrases used for the classification process. Our first iteration uses the keywords proposed by Ray et al. [2]. Unfortunately, at the end of this iteration, there were still many commits that belonged to the unknown category. We contacted Ray et al. who advised us to increase the number of keywords. Thus, we performed additional iterations. At the end of each iteration, we retrieved the most frequent words from the commits assigned to the unknown category. Then, we manually identified additional keywords for the cause and impact categories. Next, we redid the classification of the bug fixing commits using the expanded list of keywords and phrases. We continued this process until the number of commits with both their cause and impact assigned to the unknown category is reduced to less than 1% of the training data.

Table IV shows the descriptions of the bug categories along with some of the keywords used to identify them. The percentages do not add up to 100% because a bug fixing commit can be assigned to both cause and impact categories. The row corresponding to the category unknown at the bottom of the table shows the number and percentages of bug fixing commits that have *both* their cause and impact categories assigned to the unknown category.

Step 2: Supervised classification. We use the labeled training data created at the end of Step 1 to classify the remaining 90% of the bug fixing commits using a supervised learning technique. We first convert the descriptive message in each bug fixing commit log into a bag of words. Then, we identified and removed commit-specific keywords that only had a single occurrence in all bug fixing commit messages. Next, we remove stop words and performed stemming using the popular Porter stemming algorithm[11]. As the final step, we used the well-known Support Vector Machine (SVM) classification

[10]https://github.com/TrinityCore/TrinityCore

[11]http://tartarus.org/martin/PorterStemmer

algorithm[12] to create a classifier from the training data and to classify the remaining bug fixing commits.

To evaluate the built SVM classifier, we randomly choose 270 bug fixing commits for manual inspection. Next, we compared the results obtained using the built classifier with the manual inspection result. Table V summarizes the precision and recall obtained for each category. Precision for a category refers to the proportion of bug fixing commits that are correctly assigned to the category, among those that are assigned to the category. Recall for a category refers to the proportion of bug fixing commits that are correctly assigned to the category, among those that truly belong to the category. We find that the precision and recall are high (more than 80%) — similar to the precision and recall of the classifier used by Ray et al. to label their bug fixing commits.

Table V: Classifier precision and recall results

	Precision	Recall
Performance	100%	83.33%
Security	78.57%	73.33%
Failure	81.81%	90.00%
Memory	96.00%	80.00%
Programming	65.91%	96.67%
Concurrency	87.50%	70.00%
Algorithm	96.30%	86.67%
Average	86.58%	82.86%

The last two columns of Table IV summarizes the total numbers and percentages of bug fixing commits classified to each category by our two-step semi-automated classification process. The results show that most of the bug fixing commits are related to programming errors (94.10%). This is reasonable because this category includes many common errors such as issues with exception handling, typos, type errors, incorrect initializations, incorrect control flows, compilation and build errors, etc. A similar finding was reported by Ray et al. [2]. Around 3.74% of the bug fixing commits are related to incorrect memory handling; 1.01% are related to concurrency bugs, and 0.07% are related to algorithmic and logical errors. When considering impact, we find that 6.67%, 2.80%, and 1.50% of the bug fixing commits belong to the failure, security, and performance category respectively. Around 1.03% of the bug fixing commits have both their cause and impact categories assigned to the unknown category.

C. Statistical Method

We use regression analysis to study the relationship between a dependent variable and a set of independent variables given a set of control variables. We consider the same dependent and control variables that were used by Ray et al. [2] and vary the independent variables based on our research questions. Our dependent variable is the number of bug fixing commits and control variables are project age (age), project size (size), number of developers (developers), and total number of commits (commits). The control variables are intuitively positively correlated with the number of bug fixing commits. Projects of longer age, of larger size, with more developers, and with more commits, generally have more bug fixing commits. Similar to Ray et al., we use negative binomial regression (NBR) [6] as the regression analysis method. NBR is a generalized linear model that can handle over-dispersion.

To check for excessive multi-collinearity, we compute variance inflation factors (VIFs), which measure how much the variances of the estimated coefficients in a regression model are inflated because of linear dependencies among independent and control variables. Although there is no threshold value for VIF, we use the commonly accepted value of 5 [6]. We compute VIFs of all variables for all NBR models built to answer our research questions and find that no multi-collinearity problem exists.

Moreover, because the number of projects of each language differs, our dataset is unbalanced. Thus, we employ weighted effects coding [7] to deal with this data imbalance problem. Weighted effects coding allows for a straightforward interpretation of the coefficients learned by NBR for the independent/control variables. Weighted effects coding represents the relative effect of an independent/control variable on the dependent variable, compared to the weighted mean of the values of the dependent variable across all projects.

III. Findings

In this section, we describe findings which answer each of our three research questions.

A. RQ1: Does the use of more languages correlate with higher bug proneness?

Motivation: A significant number of projects use multiple programming languages in their implementations — 331 out of the 628 projects in our dataset are implemented using multiple programming languages. The usage of multiple programming languages in a project potentially increases project complexity and necessitate the creation of proper interfaces between different languages. This raises a question as to whether the usage of more programming languages is harmful to project quality. This motivates us to analyse if adding more languages significantly increases bug proneness measured by the number of bug fixing commits.

Methodology: We use the NBR method described in Section II-C with the number of languages used to implement a project (num-langs) as the independent variable. The dependent variable is the number of bug fixing commits. The control variables are: project's age, size, number of developers (developers), and total number of commits (commits).

Findings: Table VI shows the coefficients of the regression model built using NBR and the impact of various control and independent variables on the dependent variable. We can observe that the impact of all independent and control variables are statistically significant — p-values are all < 0.05. Focusing on the independent variable, the result shows that the coefficient (Coeff.) is 0.273, indicating that the number of languages has most impact on the number of bug fixing commits.

The coefficient of an independent/control variable corresponds to the expected change in the log of the dependent

[12]https://www.csie.ntu.edu.tw/~cjlin/libsvm/

variable, when there is a one unit change in the independent/control variable and all other variable values are held constant. From the coefficient of *num-langs* in Table VI, we can infer that with a unit increase in the number of languages of a project, we can expect an increase in the number of bug fix commits by a factor of $e^{0.273} = 1.31$. For example, consider a project with 100 bug fixing commits. If the project is to be implemented in one more programming language, the number of bug fixing commits is expected to increase to 100×1.31 = 131 bug fixing commits. From the table, we can also note that implementing a project in one more language has more impact to bug proneness than adding a unit to the project's age (i.e., one more day), size (i.e., one more SLOC), number of developers (i.e., one more developer), and commits (i.e., one more commit).

Table VI: Coefficients of the negative binomial regression model for RQ1. Coeff. = coefficient, age = time between the first commit and the last commit (in days), size = project size (in SLOC), developers = number of developers in the project, commits = total number of commits, num-langs = number of unique languages used in the project. AIC = 9164.93, BIC = 9196.03, Log Likelihood = -4575.46, Deviance = 695.19.

	Coeff.	Std. Error	z-val	Pr($>\|z\|$)	
(Intercept)	4.718	0.089	53.19	0.000	***
age	$2.939e^{-04}$	$3.794e^{-05}$	7.75	0.000	***
size	$3.390e^{-07}$	$1.310e^{-07}$	2.59	0.010	**
developers	$9.807e^{-04}$	$1.107e^{-04}$	8.86	0.000	***
commits	$6.048e^{-05}$	$5.177e^{-06}$	11.68	0.000	***
num-langs	0.273	0.029	9.50	0.000	***

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Number of languages has a significant impact on increasing bug proneness.

B. RQ2: Are some languages more bug prone when they are used with other languages?

Motivation: From RQ1, we find that in general, increasing the number of languages increases bug proneness. In this research question, we want to identify languages that are more bug prone when used in a multi-language setting.

Methodology: To answer RQ2, we introduce 34 independent variables. Seventeen of these variables correspond to the usage of one of the 17 languages as the primary language of a project in a single-language setting. The other seventeen correspond to the usage of the 17 languages as the primary language of a project in a multi-language setting. We use the same control and dependent variables as RQ1.

Findings: Table VII shows the impact of using each of the 17 languages on the number of bug fixing commits in a single-language (denoted as $\langle \text{language} \rangle_S$) and multi-language (denoted as $\langle \text{language} \rangle_M$) setting. From the table, we can note that the coefficients of the languages are not always statistically significant. The statistically significant ones are marked with one or multiple asterisks. There are 20 of them. For those that are not statistically significant (i.e., 14 of them), unfortunately not much conclusion can be drawn.

For some languages, the coefficient for the single-language setting is significant, while the one for the multi-language setting is not (four languages: CoffeeScript, Ruby, Erlang, Haskell). For some other languages, it is the other way around — the coefficient for the multi-language setting is significant, while the one for the single-language setting is not (four languages: C, Go, PHP, Python). For yet other languages, their coefficients for both settings are not significant (three languages: C#, JavaScript, Perl). Unfortunately, for such languages (11 languages), we cannot compare the two settings (i.e., single-language and multi-language), because the coefficient of at least one of the settings is inconclusive.

Thus, we focus on languages with statistically significant coefficients for both single and multi-language settings. We find six languages with statistically significant coefficients: C++, Objective-C, Java, TypeScript, Clojure, and Scala. For all of them, we consistently find that their coefficients are larger when they are used in a multi-language setting. This means that there is a statistically significant support that using these languages in a multi-language setting (rather than a single-language setting) increases bug proneness. The findings for the other eleven languages do not refute the six languages, because we can not conclude when coefficients are not statistically significant.

Six languages including C++, Objective-C, Java, TypeScript, Clojure, and Scala are more defect prone when they are used with other languages. The results are inconclusive for the other eleven languages.

C. RQ3: Does the correlation between number of languages and bug proneness exist for various bug categories?

Motivation: RQ1 finding highlights that number of languages is positively and significantly correlated to bug proneness. Because there are different categories of bugs, in this research question we investigate whether the positive correlation remains considering various bug categories. Additionally, we would like to investigate the categories of bugs that are the most impacted by the number of languages. Results of these investigations will shed light on the relationship between the use of multiple programming languages and bugs.

Methodology: To answer RQ3, we build multiple NBR models, one for each bug category i.e., four models for different categories of bug causes (memory, concurrency, programming and algorithm) and three for different categories of bug impacts (security, failure and performance). For each NBR model, we use the same variables as for RQ1.

Findings: Table VIII shows the coefficients of the NBR models built for the four bug cause categories. For each of the categories, we find that number of languages still has a significant impact on the number of bug fixing commits (p-value < 0.001). The coefficients are all larger than zero indicating that the more the languages, the larger the number of bug fixing commits of each category is likely to be.

Comparing the coefficients, among the four bug categories, the effect of the number of languages is higher for memory (coefficient = 0.421), concurrency (coefficient = 0.396) and algorithm (coefficient = 0.380) bugs than programming bugs

Table VII: Coefficients of the negative binomial regression model for RQ2. AIC = 9125.36, BIC = 9298.62, Log Likelihood = -4523.68, Deviance = 686.59

	Coeff.	Std. Error	z-val	Pr($>\|z\|$)	
(Intercept)	5.208	0.074	70.864	$< 2e^{-16}$	***
age	$3.091e^{-04}$	$3.828e^{-05}$	8.075	$6.77e^{-16}$	***
size	$4.143e^{-07}$	$1.462e^{-07}$	2.833	0.005	**
developers	$7.672e^{-04}$	$1.068e^{-04}$	7.181	$6.92e^{-13}$	***
commits	$6.077e^{-05}$	$5.286e^{-06}$	11.498	$< 2e^{-16}$	***
C_S	0.024	0.291	0.083	0.934	
C_M	0.331	0.136	2.250	0.024	*
$C++_S$	-0.550	0.275	-2.005	0.045	*
$C++_M$	0.609	0.138	4.403	0.000	***
$C\#_S$	0.044	0.159	0.281	0.779	
$C\#_M$	0.276	0.194	1.421	0.155	
Objective-C_S	-0.416	0.167	-2.497	0.013	*
Objective-C_M	0.909	0.345	2.634	0.008	**
Go_S	0.193	0.216	-0.893	0.372	
Go_M	0.475	0.152	3.129	0.002	**
$Java_S$	-0.568	0.164	-3.473	0.001	***
$Java_M$	0.530	0.192	2.767	0.006	**
$CoffeeScript_S$	-0.798	0.347	-2.302	0.021	*
$CoffeeScript_M$	-0.245	0.244	-0.101	0.920	
$Javascript_S$	0.164	0.130	1.264	0.206	
$Javascript_M$	0.079	0.109	0.725	0.469	
$TypeScript_S$	-3.664	0.915	-4.005	0.000	***
$TypeScript_M$	0.542	0.131	4.139	0.000	***
$Ruby_S$	-0.541	0.164	-3.297	0.001	***
$Ruby_M$	0.262	0.182	1.444	0.149	
PHP_S	-0.127	0.176	-0.721	0.471	
PHP_M	0.695	0.181	3.844	0.000	***
$Python_S$	-0.092	0.156	-0.589	0.556	
$Python_M$	0.671	0.186	3.614	0.000	***
$Perl_S$	-0.482	0.273	-1.768	0.077	
$Perl_M$	0.362	0.387	0.935	0.350	
$Clojure_S$	-0.730	0.139	-5.257	0.000	***
$Clojure_M$	-0.686	0.274	-2.508	0.012	*
$Erlang_S$	-0.883	0.150	-5.876	0.000	***
$Erlang_M$	0.100	0.258	0.389	0.697	
$Haskell_S$	-0.602	0.171	-3.515	0.000	***
$Haskell_M$	-0.037	0.198	-0.188	0.851	
$Scala_S$	-0.629	0.213	-2.948	0.003	**
$Scala_M$	0.371	0.159	2.338	0.019	*

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$; S=Single-language, M=Multi-language

(coefficient = 0.275). Programming bugs is the largest category of bugs and thus its coefficient is close to the coefficient estimated for RQ1 (i.e., considering all bugs). The three other bug categories often correspond to more complicated bugs involving incorrect memory handling, concurrency, and algorithmic errors. To avoid these bugs, developers must carefully consider many factors and deal with the complexity of a software system (c.f., [8], [9]) which may be made worse with the use of multiple languages.

Table IX shows the coefficients of the NBR models built for the three bug cause categories. Again, we find that number of languages still has a significant impact on the number of bug fixing commits (p-value < 0.001). The coefficients are all also larger than zero indicating that the more the languages, the larger the number of bug fixing commits of the three categories is likely to be. Comparing the coefficients, we find that the effect of the number of languages is higher for bugs causing performance issues (coefficient = 0.339) and failures (coefficient = 0.336) than those causing security issues (coefficient = 0.315). This result is reasonable because interaction of multiple languages may often result in performance problems caused by the additional runtime overhead due to infrastructure libraries [10], [11], [12].

Number of languages has a positive and significant impact on the number of bug fixing commits for all categories of bugs. The impact is highest for memory, concurrency, and algorithm bugs.

Table VIII: Coefficients of the negative binomial regression models for different bug cause categories.

	Memory			Concurrency			Programming			Algorithm		
	Coeff.	Std. Err.	Sig.	Coeff.	Std. Err.	Sig.	Coeff.	Std. Err.	Sig.	Coeff.	Std. Err.	Sig.
(Intercept)	0.297	0.148	*	0.265	0.160		4.651	0.089	***	-2.855	0.298	***
age	$4.078e^{-04}$	$6.230e^{-05}$	***	$6.296e^{-05}$	$6.750e^{-05}$		$2.951e^{-04}$	$3.804e^{-05}$	***	$4.031e^{-04}$	$1.160e^{-04}$	***
size	$8.519e^{-07}$	$2.107e^{-07}$	***	$1.978e^{-07}$	$2.283e^{-07}$		$3.162e^{-07}$	$1.313e^{-07}$	*	$1.062e^{-06}$	$3.465e^{-07}$	**
developers	$4.875e^{-04}$	$1.782e^{-04}$	**	$4.272e^{-04}$	$1.899e^{-04}$	*	$9.973e^{-04}$	$1.110e^{-04}$	***	$4.240e^{-04}$	$2.955e^{-04}$	
commits	$4.017e^{-05}$	$8.335e^{-06}$	***	$5.308e^{-05}$	$8.942e^{-06}$	***	$6.074e^{-05}$	$5.191e^{-06}$	***	$3.036e^{-05}$	$1.378e^{-05}$	*
num-langs	0.421	0.047	***	0.396	0.050	***	0.275	0.029	***	0.380	0.085	***
AIC	4102.07			3270.45			9091.50			1057.20		
BIC	4133.17			3301.55			9122.60			1088.30		
Log Likelihood	-2044.03			-1628.23			-4538.75			-521.60		
Deviance	705.23			666.22			695.66			314.83		
Residual Deviance (NULL)	1935.00			1373.57			2449.50			572.37		

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table IX: Coefficients of the negative binomial regression models for different bug impact categories.

	Security			Failure			Performance		
	Coeff.	Std. Err.	Sig.	Coeff.	Std. Err.	Sig.	Coeff.	Std. Err.	Sig.
(Intercept)	0.239	0.145		1.202	0.127	***	-0.281	0.159	
age	$4.721e^{-04}$	$6.092e^{-05}$	***	$3.880e^{-04}$	$5.375e^{-05}$	***	$3.526e^{-04}$	$6.620e^{-05}$	***
size	$6.324e^{-07}$	$2.061e^{-07}$	**	$8.896e^{-07}$	$1.832e^{-07}$	***	$1.400e^{-06}$	$2.188e^{-07}$	***
developers	$1.106e^{-03}$	$1.735e^{-04}$	***	$7.990e^{-04}$	$1.549e^{-04}$	***	$5.555e^{-04}$	$1.858e^{-04}$	**
commits	$4.925e^{-05}$	$8.131e^{-06}$	***	$5.372e^{-05}$	$7.246e^{-06}$	***	$4.426e^{-05}$	$8.680e^{-06}$	***
num-langs	0.315	0.046	***	0.336	0.041	***	0.339	0.050	***
AIC	4116.13			5171.43			3286.50		
BIC	4147.23			5202.52			3317.60		
Log Likelihood	-2051.07			-2578.71			-1636.25		
Deviance	703.94			725.60			661.89		
Residual Deviance (NULL)	1785.47			2107.50			1660.21		

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

IV. Discussion

In this section, we first highlight the implications of our findings in Section IV-A. We then acknowledge the limitations of our study in Section IV-B.

A. Implications

We briefly describe the implications of our findings on developer activities and future research:

Developers must weigh the cost of using multiple languages on software quality. Many software projects are developed in multiple languages. A project team may have multiple members who are proficient in different programming languages and they may choose to implement features and contribute code in their favorite languages. Although using multiple languages may bring several benefits (e.g., productivity), developers must to be careful while choosing languages. Our results show that adding new languages to implement a project can significantly increase the number of bug fix commits. Additionally, several popular languages like C++, Objective-C, Java, TypeScript, Clojure and Scala become more defect prone when used in multi-language setting. Our findings suggest that developers should not unnecessarily use more languages because it may impact software quality. The usage of an additional programming language in the implementation of a project should be supported with sufficient justification.

More research to quantify the benefit of multiple language use is needed. Although anecdotal evidence exists on the benefit of using multiple languages, unfortunately, no research study has empirically demonstrated these benefits especially via large-scale studies. We encourage future researchers to perform such studies so that the benefits of proper usages of multiple programming languages can be measured and quantified. Developers can then pick languages in a more informed way by considering the trade-offs of multiple benefit and cost factors.

More research to identify design patterns and anti-patterns in multiple language use is needed. The fact that successful projects, like Linux and OpenCV, are implemented in multiple languages suggests that good practices (i.e., design patterns) exist in the usage of multiple languages in implementing a project. Our findings which highlight that adding more languages significantly increases bug proneness suggests that poor practices (i.e., anti-patterns) exist too. We encourage researchers to identify these design patterns and anti-patterns that can help developers to better use multiple languages in their projects and thus benefit from the strengths of each of the languages without sacrificing software quality. It would also be interesting to identify common bug patterns that may affect projects written in a particular set of programming languages (c.f. [13]).

More research is needed to identify languages that are less (or more) compatible when used together. Our study highlights languages that are more bug prone when used in the multi-language setting. However, we have not looked into identifying pairs or even sets of languages that are more compatible with one another (i.e., their use results in no increase in bug proneness, or even a reduction in bug proneness), and those that are less compatible (i.e., their use results in a major increase in bug proneness). It would be interesting to identify these pairs and perform a qualitative study to identify reasons why they are less (or more) compatible to be used together.

B. Limitations

First we only consider the latest revision of the project. It is possible that over time developers added or deleted code in a new language not present in the current revision. Second, our projects belong to different domains and thus results across all domains (reported in this work) may differ from results for a particular domain. Third, we check bug proneness of each language in single-language and multi-language settings. We do not investigate if certain pairs of languages are likely to be more buggy when they are used together. Fourth, we do not differentiate cases when code written in different languages are dependent or independent of one another.

V. Threats to Validity

Our study uses a similar methodology that was used by Ray et al. [2]. Thus, our study shares many threats to validity as this prior work.

Construct Validity: First, we use bug fixing commits as a proxy for bug proneness. We choose not to use number of issue reports as a proxy because not all the projects use an issue tracking system and not all bugs are logged in the issue tracking system [3]. Still, we acknowledge that a bug may be represented by more than one bug fixing commits. Furthermore, some bug fixing commits may not be described as such in commit logs and thus may be missed in our study. Unfortunately, manually identifying the exact number of bugs for a large dataset like ours is very difficult. Second, we categorize bug fixing commits into different cause and impact categories using a semi-automated classification process. It is possible that some bugs are wrongly categorized. To minimize the threat to validity, two of the authors performed some manual steps of the classification process (e.g., the identification of ground truth labels for the 270 bug fixing commits). The two authors independently perform such manual steps, compare results, and reconcile any differences. Third, when we count the number of developers it is possible that the same developer might use different email addresses and we count the developer multiple times.

Internal Validity: Our study relies on CLOC to identify the languages used by a project. The tool may have wrongly identify languages used by a project (or commit) and the number of lines of code written in a language for a project (or a commit). To reduce the threat to validity, we have manually checked some randomly selected commits and verified that the CLOC tool behaves correctly. We do not consider complexity of the applications which might impact the results. However, we do consider size of the applications as one of the control variables. Furthermore, we do not check for and remove dead code.

External Validity: We consider projects hosted on GitHub. As such, our results may not be valid for projects on other platforms. However, we reduce this threat by analysing a large number of projects.

VI. Related Work

We first highlight related work on programming language adoption and use in Section VI-A. Next, we present related work that investigate the impact of programming languages on software quality in Section VI-B. Moreover, we present some large scale studies that analyze GitHub in Section VI-C.

A. Programming Language Adoption and Use

Meyerovich and Rabkin study factors that impact language adoption by analyzing over 200,000 projects from SourceForge, 590,000 projects tracked by Ohloh, and multiple surveys of thousands of developers [14]. They study several aspects such as language popularity, niche languages, developer migration from one language to another, demographic influence on language selection, process of learning languages and belief about languages. They find that language adoption follows a power law and it is affected by open-source libraries, existing code and prior experience of developer, whereas language features such as performance and reliability do not have an impact. Furthermore, developers learn and forget different languages throughout their career and exposure through education results in selecting varied languages whereas age has a little effect on language adoption.

Bissyande et al. conduct a large scale study on 100,000 open-source projects from GitHub to analyze popularity and interoperability of programming languages and their impact on project success (measured in terms of its popularity) [1]. They collect data such as developer contributions and issue reports from projects written in 30 programming languages of different types, for example, object-oriented, procedural, scripting, and functional. They find that Ansi C, Ruby and Python are popular languages among developers, and JavaScript, Shell and Ruby are often used with other programming languages.

Okur et al. study open-source projects that adopted Microsoft's parallel libraries – Task Parallel Library (TPL) and Parallel Language Integrated Query (PLINQ) – to understand the usage of parallel libraries by developers [15]. They analyze a total of 655 open-source projects written in C#, consisting of a total of more than 17 million lines of code (mLOC). Their results show that developers often unnecessarily make parallel code complex, applications of different sizes adopt libraries differently, and misuse of parallel constructs leads to code with parallel syntax but sequential execution.

Karus et al. study 22 open-source projects to analyze the evolution of different languages and artifacts like documentation, binaries and graphic files [16]. The 22 projects were developed in several language such as C, C++, Java, Python, Ruby, PHP and JavaScript. They find that most of the Java developers work with XML, while only half of C developers did so and new developers use fewer file types for their initial commits. Furthermore, authors find that knowing multiple languages is not enough but developers must also understand various programming paradigms.

Chen et al. study the evolution data of 17 different languages including C, C++, Java, Basic, Cobol, Fortran and 11 other languages [17]. They analyse several intrinsic factors such as generality, reliability, machine independence, extensibility, maintainability, efficiency, simplicity, and implementability. They also consider several extrinsic factors such as organizational, government and technology supports. They find that factors such as generality, reliability, machine indepen-

dence, and extensibility have correlations with the number of developers who choose the language as their primary language.

Similar to the above mentioned studies, we also investigate the adoption of languages. In particular, we investigate the adoption of multiple languages in single projects. We find that a close to half of the top projects that we download from GitHub (331 out of 628 projects) are implemented in more than one language. Different from Bissyande et al.'s study, which also analyze the interoperability of languages, we perform several data cleaning steps, e.g., focusing on popular projects, ignoring projects with little activity, etc. Due to the large number of projects in GitHub that are of poor quality [18], this data cleaning step is necessary.

B. Programming Language and Software Quality

Nanz et al. study a dataset of Rosetta code containing 8,087 solutions of 745 common programming tasks in 8 different languages to investigate several characteristics such as size of the program, running time, memory usage and defect proneness [19]. They find that functional and scripting languages help in writing concise code and languages that compile into bytecode produce smaller executables compared to the ones that produce native machine code.

Bhattacharya et al. study four open-source projects which use C and C++, i.e., Firefox, Blender, VLC Media Player and MySQL to understand the impact of languages on software quality [20]. They compute several statistical measures while controlling for factors, such as developer competence and software process. They find that applications previously written in C are migrating to C++ and C++ code is often of higher quality, less prone to bugs, and easier to maintain than C code.

Pankratius et al. perform a controlled comparative study on thirteen programmers who have worked on Java and Scala projects, to analyze aspects such as effort, language usage, performance, and programmer satisfaction [21]. They find that Scala code is more compact than Java code, however, Java scales better on parallel hardware.

Our study extends the above-mentioned studies by shedding more insight into the relationship between programming languages and software quality. We look into the effect of using multiple programming languages on defect proneness. This has not been investigated before in a prior work and thus complements them well. Our study highlights that the use of more programming languages significantly impacts bug proneness. Additionally, 6 out of the 17 languages that we analyze, i.e., C++, Objective-C, Java, TypeScript, Clojure, and Scala, are more bug-prone when they are used with other programming languages in implementing a software project. Furthermore, the use of more programming languages adversely affects all bug types with memory, concurrency, and algorithm bugs being the prominent ones. These findings were not investigated and reported by prior studies.

C. Other Large Scale Studies on GitHub

Casalnuovo et al. study 100 most popular C and C++ projects in GitHub to understand the correlation between asserts and defect occurrence and their results show that asserts have an effect on software quality [22]. Vasilescu et al. use mixed-methods approach by surveying thousands of developers and analysing thousands of projects in GitHub to investigate the relationship between gender and tenure diversity on team productivity and turnover [23]. Their findings suggest that a gender and tenure diverse team has higher productivity. Vasilescu et al. study 246 projects in GitHub to investigate the impact of usage of Continuous Integration (CI) on quality and productivity [24]. They find that teams using CI have more pull requests accepted from core contributors as well as fewer rejections from the external ones and using CI leads to finding more bugs.

Gousios et al. investigate 291 projects written in various languages to understand the pull-based software development model on GitHub [25]. They find that only 14% of the active projects use pull-requests and 60% of the pull-requests are processed in a day. Kochhar et al. investigate 50,000 projects to study the correlation between the presence of test cases and various project development characteristics, including the lines of code and the size of development teams [26], [27].

VII. Conclusion and Future Work

Developers often leverage the strengths of different programming languages to implement a software project. However, adding more languages in the implementation of a project can potentially increase the complexity of the project, which may translate into software quality issues. In this study, we empirically investigate the impact of increasing languages used to implement a project on bug proneness measured in terms of the number of bug fixing commits. We analyze more than 600 projects written in 17 different languages from GitHub and build regression models to study the effect of increasing languages on bug proneness while controlling for various factors such as project age, project size, number of developers, and number of commits.

Our empirical study leads to the following findings:

1) Increasing the number of languages to implement a project significantly increases bug proneness.
2) Six languages including C++, Objective-C, Java, TypeScript, Clojure, and Scala are significantly more defect prone when used in multi-language projects.
3) The impact of the number of languages to cause higher bug proneness is significantly observed for all kinds of bug categories. The impact is the highest for memory, concurrency, and algorithm bugs.

As a future work, we would like to identify pairs of languages that are less compatible and more bug prone. We are also interested to investigate the types of bugs that affect multi-language programs, and recommend mitigation strategies to deal with them. We are also interested to expand our study beyond the 17 languages considered in this paper.

Dataset

Our dataset is made publicly available and it can be downloaded from: *https://github.com/smusis/multiple-languages*.

Acknowledgement

We would like to thank Ray et al. for providing clarifications related to the empirical study settings reported in their paper [2].

REFERENCES

[1] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference*, pp. 303–312, 2013.

[2] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "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*, pp. 155–165, 2014.

[3] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillère, J. Klein, and Y. L. Traon, "Got issues? who cares about it? A large scale investigation of issue trackers from github," in *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pp. 188–197, 2013.

[4] H. Borges, M. T. Valente, A. Hora, and J. Coelho, "On the popularity of github applications: A preliminary note," *CoRR*, vol. abs/1507.00604, 2015.

[5] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: Transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work (CSCW)*, pp. 1277–1286, 2012.

[6] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, "Applied multiple regression/correlation analysis for the behavioral sciences," *Lawrence Erlbaum*, 2003.

[7] D. Posnett, C. Bird, and P. Devanbu, "An Empirical Study on the Influence of Pattern Roles on Change-Proneness," *Empirical Software Engineering, An International Journal*, vol. 16, pp. 396–423, 2010.

[8] Heap Analysis: Making Memory Errors a Thing of the Past, http://www.qnx.com/developers/docs/6.4.0/neutrino/prog/hat.html, Last accessed on November 16, 2015.

[9] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 329–339, 2008.

[10] Stack Overflow, Why do you or do you not implement using polyglot solutions?, http://stackoverflow.com/questions/86151/why-do-you-or-do-you-not-implement-using-polyglot-solutions, Last accessed on November 16, 2015.

[11] Shutterstock's polyglot approach, https://jaxenter.com/shutterstocks-polyglot-approach-117077.html, Last accessed on November 16, 2015.

[12] Polyglot Programming: Is building applications with multiple languages a good practice?, http://stackoverflow.com/questions/141669/polyglot-programming-is-building-applications-with-multiple-languages-a-good-pr, Last accessed on November 16, 2015.

[13] G. Tan and J. Croft, "An empirical security study of the native code in the jdk," in *Proceedings of the 17th Conference on Security Symposium*, pp. 365–377, 2008.

[14] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pp. 1–18, 2013.

[15] S. Okur and D. Dig, "How do developers use parallel libraries?," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, pp. 54:1–54:11, 2012.

[16] S. Karus and H. Gall, "A study of language usage evolution in open source software," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, pp. 13–22, 2011.

[17] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang, "An empirical study of programming language trends," *IEEE Software*, vol. 22, no. 3, pp. 72–78, 2005.

[18] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. Damian, "The promises and perils of mining github," in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pp. 92–101, 2014.

[19] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pp. 778–788, 2015.

[20] P. Bhattacharya and I. Neamtiu, "Assessing programming language impact on development and maintenance: A study on c and c++," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pp. 171–180, 2011.

[21] V. Pankratius, F. Schmidt, and G. Garreton, "Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java," in *34th International Conference on Software Engineering (ICSE)*, pp. 123–133, 2012.

[22] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray, "Assert use in github projects," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pp. 755–766, 2015.

[23] B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov, "Gender and tenure diversity in github teams," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI)*, pp. 3789–3798, 2015.

[24] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 805–816, 2015.

[25] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pp. 345–355, 2014.

[26] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "Adoption of software testing in open source projects-a preliminary study on 50, 000 projects," in *17th European Conference on Software Maintenance and Reengineering, (CSMR)*, pp. 353–356, 2013.

[27] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *2013 13th International Conference on Quality Software (QSIC)*, pp. 103–112, 2013.