On the Impact of Programming Languages on Code Quality

A Reproduction Study

EMERY D. BERGER, University of Massachusetts Amherst and Microsoft Research
CELESTE HOLLENBECK, Northeastern University
PETR MAJ, Czech Technical University in Prague
OLGA VITEK, Northeastern University
JAN VITEK, Northeastern University and Czech Technical University in Prague

In a 2014 paper, Ray, Posnett, Devanbu, and Filkov claimed to have uncovered a statistically significant association between eleven programming languages and software defects in 729 projects hosted on GitHub. Specifically, their work answered four research questions relating to software defects and programming languages. With data and code provided by the authors, the present paper first attempts to conduct an experimental repetition of the original study. The repetition is only partially successful, due to missing code and issues with the classification of languages. The second part of this work focuses on their main claim, the association between bugs and languages, and performs a complete, independent reanalysis of the data and of the statistical modeling steps undertaken by Ray *et al.* in 2014. This reanalysis uncovers a number of serious flaws which reduce the number of languages with an association with defects down from eleven to only four. Moreover, the practical effect size is exceedingly small. These results thus undermine the conclusions of the original study. Correcting the record is important, as many subsequent works have cited the 2014 paper and have asserted, without evidence, a causal link between the choice of programming language for a given task and the number of software defects. Causation is not supported by the data at hand; and, in our opinion, even after fixing the methodological flaws we uncovered, too many unaccounted sources of bias remain to hope for a meaningful comparison of bug rates across languages.

ACM Reference Format:
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. 1, 1 (June 2019), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

At heart, a programming language embodies a bet: the bet that a given set of abstractions will increase developers' ability to deliver software that meets its requirements. Empirically quantifying the benefits of any set of language features over others presents methodological challenges. While one could have multiple teams of experienced programmers develop the same application in different languages, such experiments are too costly to be practical. Instead, when pressed to justify their choices, language designers often resort to intuitive arguments or proxies for productivity such as numbers of lines of code.

Authors' addresses: Emery D. Berger, University of Massachusetts Amherst and Microsoft Research; Celeste Hollenbeck, Northeastern University; Petr Maj, Czech Technical University in Prague; Olga Vitek, Northeastern University; Jan Vitek, Northeastern University and, Czech Technical University in Prague.

XXXX-XXXX/2019/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, large-scale hosting services for code, such as GitHub or SourceForge, offer a glimpse into the life-cycles of software. Not only do they host the sources for millions of projects, but they also log changes to their code. It is tempting to use these data to mine for broad patterns across programming languages. The paper we reproduce here is an influential attempt to develop a statistical model that relates various aspects of programming language design to software quality.

What is the effect of programming language on software quality? is the question at the heart of the study by Ray *et al.* published at the 2014 Foundations of Software Engineering (FSE) conference [26]. The work was sufficiently well-regarded in the software engineering community to be nominated as a Communication of the ACM (CACM) *Research Highlight*. After another round of reviewing, a slightly edited version appeared in journal form in 2017 [25]. A subset of the authors also published a short version of the work as a book chapter [24]. The results reported in the FSE paper and later repeated in the followup works are based on an observational study of a corpus of 729 GitHub projects written in 17 programming languages. To measure quality of code, the authors identified, annotated, and tallied commits which were deemed to indicate bug fixes. The authors then fit a Negative Binomial regression against the labeled data, which was used to answer the following four research questions:

- RQ1 "**Some languages have a greater association with defects than others**, although the effect is small." Languages associated with fewer bugs were TypeScript, Clojure, Haskell, Ruby, and Scala; while C, C++, Objective-C, JavaScript, PHP and Python were associated with more bugs.
- RQ2 "**There is a small but significant relationship between language class and defects.** Functional languages have a smaller relationship to defects than either procedural or scripting languages."
- RQ3 "**There is no general relationship between domain and language defect proneness.**" Thus, application domains are less important to software defects than languages.
- RQ4 "**Defect types are strongly associated with languages.** Some defect types like memory errors and concurrency errors also depend on language primitives. Language matters more for specific categories than it does for defects overall."

Of these four results, it is the first two that garnered the most attention both in print and on social media. This is likely the case because those results confirmed commonly held beliefs about the benefits of static type systems and the need to limit the use of side effects in programming.

Correlation is not causality, but it is tempting to confuse them. The original study couched its results in terms of *associations* (*i.e.*, correlations) rather than *effects* (*i.e.*, *causality*) and carefully qualified effect size. Unfortunately, many of the paper's readers were not as careful. The work was taken by many as a statement on the impact of programming languages on defects. Thus, one can find citations such as:

- "*...They found language design did have a significant, but modest effect on software quality.*" [23]
- "*...The results indicate that strong languages have better code quality than weak languages.*" [31]
- "*...functional languages have an advantage over procedural languages.*" [21]

Table 1 summarizes our citation analysis. Of the 119 papers that were retrieved,[1] 90 citations were either passing references (Cursory) or discussed the methodology of the original study (Methods). Out of the citations that discussed the results, 4 were careful to talk about associations (*i.e.*, correlation), while 26 used language that indicated effects (*i.e.*, causation). It is particularly interesting to observe that

	Cites	Self
Cursory	77	1
Methods	12	0
Correlation	2	2
Causation	24	3

Table 1. Citation analysis

[1]Retrieval performed on 12/01/18 based on the Google Scholar citations of the FSE paper; duplicates were removed.

even the original authors, when they cite their own work, sometimes resort to causal language. For example, Ray and Posnett write "*Based on our previous study [26] we found that the overall effect of language on code quality is rather modest.*" [24], Devanbu writes "*We found that static typing is somewhat better than dynamic typing, strong typing is better than weak typing, and built-in memory management is better*" [5], and "*Ray [...] said in an interview that functional languages were boosted by their reliance on being mathematical and the likelihood that more experienced programmers use them.*" [15]. Section 2 of the present paper gives a detailed account of the original study and its conclusions.

Given the controversy generated by the CACM paper on social media, and some surprising observations in the text of the original study (*e.g.*, that Chrome V8 is their largest JavaScript project—when the virtual machine is written in C++), we wanted to gain a better understanding of the exact nature of the scientific claims made in the study and how broadly they are actually applicable. To this end, we chose to conduct an independent reproduction study.

A reproduction study aims to answer the question *can we trust the papers we cite?* Over a decade ago, following a spate of refutations, Ioannidis argued that most research findings are false [13]. His reasoning factored in small effect sizes, limited number of experiments, misunderstanding of statistics, and pressure to publish. While refutations in computer science are rare, there are worrisome signs. Kalibera *et al.* reported that 39 of 42 PLDI 2011 papers failed to report any uncertainty in measurements [29]. Reyes *et al.* catalogued statistical errors in 30% of the empirical papers published at ICSE [27] from 2006 to 2015. Other examples include the critical review of patch generation research by Monperrus [20] and the assessment of experimental fuzzing evaluations by Klees *et al.* [14]. To improve the situation, our best bet is to encourage a culture of *reproducible research* [8]. Reproduction increases our confidence: an experimental result reproduced independently by multiple authors is more likely to be valid than the outcome of a single study. Initiatives such as SIGPLAN and SIGSOFT's artifact evaluation process, which started at FSE and spread widely [16], are part of a move towards increased reproducibility.

Methodology. Reproducibility of results is not a binary proposition. Instead, it spans a spectrum of objectives that provide assurances of different kinds (see Figure 1 using terms from [9, 29]). *Experimental repetition* aims to replicate the results of some previous work with the same data and methods and should yield the same numeric results. Repetition is the basic guarantee provided by artifact evaluation [16]. *Reanalysis* examines the robustness of the conclusions to the methodological choices. Multiple analysis methods may be appropriate for a given dataset, and the conclusions should be robust to the choice of method. Occasionally, small errors may need to be fixed, but the broad conclusions should hold. Finally, *Reproduction* is the gold standard; it implies a full-fledged independent experiment conducted with different data and the same or different methods. To avoid bias, repetition, reanalysis, and reproduction are conducted independently. The only contact expected with the original authors is to request their data and code.

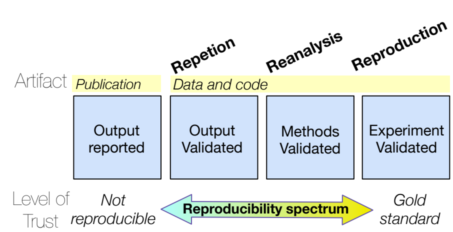

Fig. 1. Reproducibility spectrum (from [22])

Results. We began with an experimental repetition, conducting it in a similar fashion to a conference artifact evaluation [16] (Section 3 of the paper). Intuitively, a repetition should simply be a matter of running the code provided by the authors on the original data. Unfortunately, things often don't work out so smoothly. The repetition was only partially successful. We were able to mostly replicate RQ1 based on the artifact provided by the authors. We found ten languages with a statistically significant association with errors, instead of the eleven reported. For RQ2, we uncovered classification errors that made our results depart from the published ones. In other words, while we could repeat the original, its results were meaningless. Lastly, RQ3 and RQ4 could not be repeated due to missing code and discrepancies in the data.

For *reanalysis*, we focused on RQ1 and discovered significant methodological flaws (Section 4 of this paper). While the original study found that 11 out of 17 languages were correlated with a higher or lower number of defective commits, upon cleaning and reanalyzing the data, the number of languages dropped to 7. Investigations of the original statistical modeling revealed technical oversights such as inappropriate handling of multiple hypothesis testing. Finally, we enlisted the help of independent developers to cross-check the original method of labeling defective commits, which led us to estimate a false positive rate of 36% on buggy commit labels. Combining corrections for all of these aforementioned items, the reanalysis revealed that only 4 out of the original 11 languages correlated with abnormal defect rates, and even for those the effect size is exceedingly small.

Figure 2 summarizes our results: Not only is it not possible to establish a causal link between programming language and code quality based on the data at hand, but even their correlation proves questionable. Our analysis is repeatable and available in an artifact hosted at: https://github.com/PRL-PRG/TOPLAS19_Artifact.

Repetition

RQ1	RQ2	RQ3	RQ4
✓	✗	✗	✗

Reanalysis (RQ1)

	FSE'14	This paper
positive association	C++ Objective-C C PHP Python JavaScript	C++
negative association	TypeScript Clojure Scala Haskell Ruby	Clojure Haskell Ruby
no statistically significant association	C# Go CoffeeScript Java Perl Erlang	C C# Objective-C Go JavaScript CoffeeScript Java Perl PHP Python Erlang Scala TypeScript

Fig. 2. Result summary

Follow up work. While reanalysis was not able to validate the results of the original study, we stopped short of conducting a reproduction as it is unclear what that would yield. In fact, even if we were to obtain clean data and use the proper statistical methods, more research is needed to understand all the various sources of bias that may affect the outcomes. Section 5 lists some challenges that we discovered while doing our repetition. For instance, the ages of the projects vary across languages (older languages such as C are dominated by mature projects such as Linux), and the data include substantial numbers of commits to test files (how bugs in tests are affected by language characteristics is an interesting question for future research). We believe that there is a need for future research on this topic; we thus conclude our paper with some best practice recommendations for future researchers (Section 6).

2 ORIGINAL STUDY AND ITS CONCLUSIONS

2.1 Overview

The FSE paper by Ray *et al.* [26] aimed to explore associations between languages, paradigms, application domains, and software defects from a real-world ecosystem across multiple years. Its multi-step, mixed-method approach included collecting commit information from GitHub; identifying each commit associated with a bug correction; and using Negative Binomial Regression (NBR) to analyze the prevalence of bugs. The paper claims to answer the following questions.

RQ1. *Are some languages more defect prone than others?*

The paper concluded that "*Some languages have a greater association with defects than others, although the effect is small.*" Results appear in a table that fits an NBR model to the data; it reports coefficient estimates, their standard errors, and ranges of p-values. The authors noted that confounders other than languages explained most of the variation in the number of bug-fixing commits, quantified by analysis of deviance. They reported p-values below .05, .01, and .001 as "statistically significant". Based on these associations, readers may be tempted to conclude that TypeScript, Haskell, Clojure, Ruby, and Scala were less error-prone; and C++, Objective-C, C, JavaScript, PHP, and Python were more error-prone. Of course, this would be incorrect as association is not causation.

RQ2. *Which language properties relate to defects?*

The study concluded that "*There is a small but significant relationship between language class and defects. Functional languages have a smaller relationship to defects than either procedural or scripting languages.*" The impact of nine language categories across four classes was assessed. Since the categories were highly correlated (and thus compromised the stability of the NBR), the paper modeled aggregations of the languages by class. The regression included the same confounders as in RQ1 and represented language classes. The authors report the coefficients, their standard errors, and ranges of p-values. These results may lead readers to conclude that functional, strongly typed languages induced fewer errors, while procedural, weakly typed, unmanaged languages induced more errors.

RQ3. *Does language defect proneness depend on domain?*

The study used a mix of automatic and manual methods to classify projects into six application domains. After removing outliers, and calculating the Spearman correlation between the order of languages by bug ratio within domains against the order of languages by bug ratio for all domains, it concluded that "*There is no general relationship between domain and language defect proneness*". The paper states that all domains show significant positive correlation, except the Database domain. From this, readers might conclude that the variation in defect proneness comes from the languages themselves, making domain a less indicative factor.

RQ4. *What's the relation between language & bug category?*

The study concluded that "*Defect types are strongly associated with languages; Some defect type like memory error, concurrency errors also depend on language primitives. Language matters more for specific categories than it does for defects overall.*" The authors report that 88% of the errors fall under the general Programming category, for which results are similar to RQ1. Memory Errors account for 5% of the bugs, Concurrency for 2%, and Security and other impact errors for 7%. For Memory, languages with manual memory management have more errors. Java stands out; it is the only garbage collected language associated with more memory errors. For Concurrency, inherently single-threaded languages (Python, JavaScript, ...) have fewer errors than languages with concurrency primitives. The causal relation for Memory and Concurrency is understandable, as the classes of errors require particular language features.

2.2 Methods in the original study

Below, we summarize the process of data analysis by the original manuscript while splitting it into the following three phases: data acquisition, cleaning, and modeling.

2.2.1 Data Acquisition. For each of the 17 languages with the most projects on GitHub, 50 projects with the highest star rankings were selected. Any project with fewer than 28 commits was filtered out, leaving 729 projects (86%). For each project, commit histories were collected with `git log --no-merges --numstat`. The data were split into rows, such that each row had a unique combination of file name, project name, and commit identifier. Other fields included committer and author name, date of the commit, commit message, and number of lines inserted and deleted. In summary, the original paper states that the input consisted of 729 projects written in 17 languages, accounting for 63 million SLOC created over 1.5 million commits written by 29 thousand authors. Of these, 566,000 commits were bug fixes.

2.2.2 Data Cleaning. As any project may be written in multiple languages, each row of the data is labeled by language based on the file's extension (TypeScript is `.ts`, and so on). To rule out small change sets, projects with fewer than 20 commits in any single language are filtered out for that language. Commits are labeled as bug fixes by searching for error-related keywords: *error, bug, fix, issue, mistake, incorrect, fault, defect* and *flaw* in commit messages. This is similar to a heuristic introduced by Mockus and Votta [19]. Each row of the data is furthermore labeled with four extra attributes. The Paradigm class is either procedural, functional, or scripting. The Compile class indicates whether a language is statically or dynamically typed. The Type class indicates whether a language admits 'type-confusion', *i.e.*, it allows interpreting a memory region populated by a value of one type as another type. A language is strongly typed if it explicitly detects type confusion and reports it as such. The Memory class indicates whether the language requires developers to manage memory by hand.

2.2.3 Statistical Modeling. For RQ1, the manuscript specified an NBR [7], where an observation is a combination of project and language. In other words, a project written in three languages has three observations. For each observation, the regression uses bug-fixing commits as a response variable, and the languages as the independent variables. NBR is an appropriate choice, given the non-negative and discrete nature of the counts of commits. To adjust for differences between the observations, the regression includes the confounders age, number of commits, number of developers, and size (represented by inserted lines in commits), all log-transformed to improve the quality of fit. For the purposes of RQ1, the model for an observation i is:

$$\begin{gathered}\text{bcommits}_i \sim \mathit{NegativeBinomial}(\mu_i, \theta), \text{ where} \\ \text{E}\{\text{bcommits}_i\} = \mu_i \\ \text{Var}\{\text{bcommits}_i\} = \mu_i + \mu_i^2/\theta \\ \mathit{log}\,\mu_i = \beta_0 + \beta_1 \log(\text{commits})_i + \beta_2 \log(\text{age})_i + \beta_3 \log(\text{size})_i + \beta_4 \log(\text{devs})_i + \textstyle\sum_{j=1}^{16} \beta_{(4+j)} \text{language}_{ij}\end{gathered}$$

The programming languages are coded with *weighted contrasts*. These contrasts are customized in a way to interpret β_0 as the average log-expected number of bugs in the dataset. Therefore, $\beta_5, \ldots, \beta_{20}$ are the deviations of the log-expected number of bug-fixing commits in a language from the average of the log-expected number of bug-fixing commits. Finally, the coefficient β_{21} (corresponding to the last language in alphanumeric order) is derived from the contrasts after the model fit [17]. Coefficients with a statistically significant negative value indicate a lower expected number of bug-fixing commits; coefficients with a significant positive value indicate a higher expected number of bug-fixing commits. The model-based inference of parameters $\beta_5, \ldots, \beta_{21}$ is the main focus of RQ1.

For RQ2, the study fit another NBR, with the same confounder variables, to study the association between language classes and the number of bug-fixing commits. It then uses Analysis of Deviance to quantify the variation attributed to language classes and the confounders. For RQ3, the paper calculates the Spearman's correlation coefficient between defectiveness by domain and defectiveness overall, with respect to language, to discuss the association between languages versus that by domain. For RQ4, the study once again uses NBR, with the same confounders, to explore the propensity for bugfixes among the languages with regard to bug types.

3 EXPERIMENTAL REPETITION

Our first objective is to repeat the analyses of the FSE paper and to obtain the same results. We requested and received from the original authors an artifact containing 3.45 GB of processed data and 696 lines of R code to load the data and perform statistical modeling steps.

3.1 Methods

Ideally, a repetition should be a simple process, where a script generates results and these match the results in the published paper. In our case, we only had part of the code needed to generate the expected tables and no code for graphs. We therefore wrote new R scripts to mimic all of the steps, as described in the original manuscript. We found it essential to automate the production of all numbers, tables, and graphs shown in our paper as we had to iterate multiple times. The code for repetition amounts to 1,140 lines of R (file `repetition.Rmd` and `implementation.R` in our artifact).

3.2 Results

The data was provided to us in the form of two CSV files. The first, larger file contained one row per file and commit, and it contained the bug fix labels. The second, smaller file aggregated rows with the same commit and the same language. Upon preliminary inspection, we observed that the files contained information on 729 projects and 1.5 million commits. We found an additional 148 projects that were omitted from the original study without explanation. We choose to ignore those projects as data volume is not an issue here.

Developers vs. Committers. One discrepancy was the 47 thousand authors we observed versus the 29 thousand reported. This is explained by the fact that, although the FSE paper claimed to use *developers* as a control variable, it was in fact counting *committers*: a subset of developers with commit rights. For instance, Linus Torvalds has 73,038 commits, of which he personally authored 11,343, the remaining are due to other members of the project. The rationale for using developers as a control variable is that the same individual may be more or less prone to committing bugs, but this argument does not hold for committers as they aggregate the work of multiple developers. We chose to retain committers for our reproduction but note that this choice should be revisited in follow up work.

Measuring code size. The commits represented 80.7 million lines of code. We could not account for a difference of 17 million SLOC from the reported size. We also remark, but do not act on, the fact that project size, computed in the FSE paper as the sum of inserted lines, is not accurate—as it does not take deletions into account. We tried to subtract deleted lines and obtained projects with negative line counts. This is due to the treatments of Git merges. A merge is a commit which combines conflicting changes of two parent commits. Merge commits are not present in our data; only parent commits are used, as they have more meaningful messages. If both parent commits of a merge delete the same lines, the deletions are double counted. It is unclear what the right metric of size should be.

3.2.1 Are some languages more defect prone than others (RQ1). We were able to qualitatively (although not exactly) repeat the result of RQ1. Table 2 (a) has the original results, and (c) has our repetition. Grey cells indicate disagreement with the conclusion of the original work. One disagreement in our repetition is with PHP. The FSE paper reported a p-value <.001, while we

Table 2. Negative Binomial Regression for Languages (grey indicates disagreement with the conclusion of the original work)

	Original Authors				Repetition	
	(a) FSE [26]		(b) CACM [25]		(c)	
	Coef	P-val	Coef	P-val	Coef	P-val
Intercept	-1.93	<0.001	-2.04	<0.001	-1.8	<0.001
log commits	2.26	<0.001	0.96	<0.001	0.97	<0.001
log age	0.11	<0.01	0.06	<0.001	0.03	0.03
log size	0.05	<0.05	0.04	<0.001	0.02	<0.05
log devs	0.16	<0.001	0.06	<0.001	0.07	<0.001
C	0.15	<0.001	0.11	<0.01	0.16	<0.001
C++	0.23	<0.001	0.18	<0.001	0.22	<0.001
C#	0.03	–	-0.02	–	0.03	0.602
Objective-C	0.18	<0.001	0.15	<0.01	0.17	0.001
Go	-0.08	–	-0.11	–	-0.11	0.086
Java	-0.01	–	-0.06	–	-0.02	0.61
Coffeescript	-0.07	–	0.06	–	0.05	0.325
Javascript	0.06	<0.01	0.03	–	0.07	<0.01
Typescript	-0.43	<0.001	0.15	–	-0.41	<0.001
Ruby	-0.15	<0.05	-0.13	<0.01	-0.13	<0.05
Php	0.15	<0.001	0.1	<0.05	0.13	0.009
Python	0.1	<0.01	0.08	<0.05	0.1	<0.01
Perl	-0.15	–	-0.12	–	-0.11	0.218
Clojure	-0.29	<0.001	-0.3	<0.001	-0.31	<0.001
Erlang	0	–	-0.03	–	0	1
Haskell	-0.23	<0.001	-0.26	<0.001	-0.24	<0.001
Scala	-0.28	<0.001	-0.24	<0.001	-0.22	<0.001

observed <.01; per their established threshold of .005, the association of PHP with defects is not statistically significant. The original authors corrected that value in their CACM repetition (shown in Table 2 (b)), so this may just be a reporting error. On the other hand, the CACM paper dropped the significance of JavaScript and TypeScript without explanation. The other difference is in the coefficients for the control variables. Upon inspection of the code, we noticed that the original manuscript used a combination of log and log10 transformations of these variables, while the repetition consistently used log. The author's CACM repetition fixed this problem.

3.2.2 Which language properties relate to defects (RQ2). As we approached RQ2, we faced an issue with the language categorization used in the FSE paper. The original categorization is reprinted in Table 3. The intuition is that each category should group languages that have "similar" characteristics along some axis of language design.

The first thing to observe is that any such categorization will have some unclear fits. The original authors admitted as much by excluding TypeScript from this table, as it was not obvious whether a gradually typed language is static or dynamic. But there were other odd ducks. Scala is categorized as a functional language, yet it allows programs to be written in an imperative manner. We are not aware of any study that shows that the majority of Scala users write functional code. Our experience with Scala is that users freely mix functional and imperative programming. Objective-C is listed as a statically compiled and unmanaged language. However, Objective-C has an object system that is inspired by SmallTalk; its treatment of objects is quite dynamic, and objects are collected by reference counting, so its memory is partially managed. The Type category is the most counter-intuitive for programming language experts as it expresses whether a language allows value of one type to be interpreted as another, e.g. due to automatic conversion. The CACM paper attempted to clarify this definition with the example of the `ID` type. In Objective-C, an `ID` variable can hold any value. If this is what the authors intend, then Python, Ruby, Clojure, and Erlang would be weak as they have similar generic types.

In our repetition, we modified the categories accordingly and introduced a new category of Functional-Dynamic-Weak-Managed to accommodate Clojure and Erlang. Table 4(c) summarizes the results with the new categorization. The reclassification (using zero-sum contrasts introduced in section 4.2.1) disagrees on the significance of 2 out of 5 categories. We note that we could repeat the results of the original classification, but since that classification is wrong, those results are not meaningful.

3.2.3 Does language defect proneness depend on domain (RQ3). We were unable to repeat RQ3, as the artifact did not include code to compute the results. In a repetition, one expects the code to be available. However, the data contained the classification of projects in domains, which allowed us to attempt to recreate part of the analysis described in the paper. While we successfully replicated the initial analysis step, we could not match the removal of outliers described in the FSE paper. Stepping outside of the repetition, we explore an alternative approach to answer the question. Table 5 uses an NBR with domains instead of languages. The results suggest there is no evidence that the application domain is a predictor of bug-fixes as the paper claims. So, while we cannot repeat the result, the conclusion likely holds.

3.2.4 What's the relation between language & bug category (RQ4). We were unable to repeat the results of RQ4 because the artifact did not contain the code which implemented the heatmap or NBR for bug types. Additionally, we found no single column in the data that contained the bug categories reported in the FSE paper. It was further unclear whether the bug types were disjoint: adding together all of the percentages for every bug type mentioned in Table 5 of the FSE study totaled 104%. The input CSV file did contain two columns which, when combined, matched these categories. When we attempted to reconstruct the categories and compared counts of each bug type, we found discrepancies with those originally reported. For example, we had 9 times as many Unknown bugs as the original, but we had only less than half the number of Memory bugs. Such discrepancies make repetition invalid.

3.3 Outcome

The repetition was partly successful. RQ1 produced small differences, but qualitatively similar conclusions. RQ2 could be repeated, but we noted issues with language classification; fixing these issues changed the outcome for 2 out of 5 categories. RQ3 could not be repeated, as the code was

Table 3. Language classes defined by the FSE paper.

Classes	Categories	Languages
Paradigm	Procedural	C C++ C# Objective-C Java Go
	Scripting	CoffeeScript JavaScript Python Perl PHP Ruby
	Functional	Clojure Erlang Haskell Scala
Compilation	Static	C C++ C# Objective-C Java Go Haskell Scala
	Dynamic	CoffeeScript JavaScript Python Perl PHP Ruby Clojure Erlang
Type	Strong	C# Java Go Python Ruby Clojure Erlang Haskell Scala
	Weak	C C++ Objective-C PHP Perl CoffeeScript JavaScript
Memory	Unmanaged	C C++ Objective-C
	Managed	Others

Table 4. Negative Binomial Regression for Language Classes

	(a) Original		(b) Repetition		(c) Reclassification	
	Coef	P-val	Coef	P-val	Coef	P-val
Intercept	-2.13	<0.001	-2.14	<0.001	-1.85	<0.001
log age	0.07	<0.001	0.15	<0.001	0.05	0.003
log size	0.05	<0.001	0.05	<0.001	0.01	0.552
log devs	0.07	<0.001	0.15	<0.001	0.07	<0.001
log commits	0.96	<0.001	2.19	<0.001	1	<0.001
`Fun Sta Str Man`	-0.25	<0.001	-0.25	<0.001	-0.27	<0.001
`Pro Sta Str Man`	-0.06	<0.05	-0.06	0.039	-0.03	0.24
`Pro Sta Wea Unm`	0.14	<0.001	0.14	<0.001	0.19	0
`Scr Dyn Wea Man`	0.04	<0.05	0.04	0.018	0	0.86
`Fun Dyn Str Man`	-0.17	<0.001	-0.17	<0.001	–	–
`Scr Dyn Str Man`	0.001	–	0	0.906	–	–
`Fun Dyn Wea Man`	–	–	–	–	-0.18	<0.001

Language classes are combined procedural (Pro), functional (Fun), scripting (Scr), dynamic (Dyn), static (Sta), strong (Str), weak (Wea), managed (Man), and unmanaged (Unm). Rows marked – have no observation.

Table 5. NBR for RQ3

	Coef	p-Val
(Intercept)	-1.94	<0.001
log age	0.05	<0.001
log size	0.03	<0.001
log devs	0.08	<0.001
log commits	0.96	<0.001

	Coef	p-Val
Application	0	1.00
CodeAnalyzer	-0.05	0.93
Database	0.04	1.00
Framework	0.01	1.00
Library	-0.06	0.23
Middleware	0	1.00

missing and our reverse engineering attempts failed. RQ4 could not be repeated due to irreconcilable differences in the data.

4 REANALYSIS

Our second objective is to carry out a reanalysis of RQ1 of the FSE paper. The reanalysis differs from repetition in that it proposes alternative data processing and statistical analyses to address what we identify as methodological weaknesses of the original work.

4.1 Methods: Data processing

First, we examined more closely the process of data acquisition in the original work. This step was intended as a quality control, and it did not result in changes to the data.

We wrote software to automatically download and check commits of projects against GitHub histories. Out of 729 projects used in the FSE paper, 618 could be downloaded. The other projects may have been deleted or became private. The downloaded projects were matched by name. As the FSE data lacked project owner names, the matches were ambiguous. By checking for matching SHAs, we confidently identified 423 projects as belonging to the study. For each matched project, we compared its entire history of commits to its commits in the FSE dataset, as follows. We identified the most recent commit *c* occurring in both. Commits chronologically older than *c* were classified as either *valid* (appearing in the original study), *irrelevant* (not affecting language files), or *missing* (not appearing in the original study).

We found 106K missing commits (*i.e.* 19.95% of the dataset). Perl stands out with 80% of commits that were missing in the original manuscript (Fig. 3 lists the ratio of missing commits per language). Manual inspection of a random sample of the missing commits did not reveal any pattern. We also recorded *invalid* commits (occurring in the study but absent from the GitHub history). Four projects had substantial numbers of invalid commits, likely due to matching errors or a change in commit history (such as with the `git rebase` command).

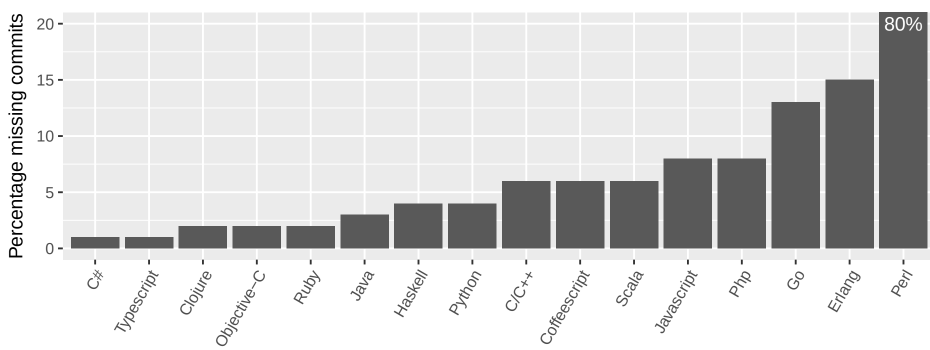

Fig. 3. Percentage of commits identified as missing from the FSE dataset.

Next, we applied three data cleaning steps (see below for details; each of these was necessary to compensate for errors in data acquisition of the original study): (1) *Deduplication*, (2) *Removal of TypeScript*, (3) *Accounting for C and C++*. Our implementation consists of 1323 lines of R code split between files `re-analysis.Rmd` and `implementation.R` in the artifact.

4.1.1 Deduplication. While the input data did not include forks, we checked for project similarities by searching for projects with similar commit identifiers. We found 33 projects that shared one or more commits. Of those, 18 were related to `bitcoin`, a popular project that was frequently copied and modified. The projects with duplicate commits are: litecoin, mega-coin, memorycoin, bitcoin, bitcoin-qt-i2p, anoncoin, smallchange, primecoin, terracoin, zetacoin, datacoin, datacoin-hp, freicoin, ppcoin, namecoin, namecoin-qt, namecoinq, ProtoShares, QGIS, Quantum-GIS, incubator-spark, spark, sbt, xsbt, Play20, playframework, ravendb, SignalR, Newtonsoft.Json, Hystrix, RxJava, clojure-scheme, clojurescript. In total, there were 27,450 duplicated commits, or 1.86% of all commits. We deleted these commits from our dataset to avoid double counting some bugs.

4.1.2 Removal of TypeScript. In the original dataset, the first commit for TypeScript was recorded on 2003-03-21, several years before the language was created. Upon inspection, we found that the file extension `.ts` is used for XML files containing human language translations. Out of 41 projects labeled as TypeScript, only 16 contained TypeScript. This reduced the number of commits from 10,063 to an even smaller 3,782. Unfortunately, the three largest remaining projects (`typescript-node-definitions`, `DefinitelyTyped`, and the deprecated `tsd`) contained only declarations and no code. They accounted for 34.6% of the remaining TypeScript commits. Given the small size of the remaining corpus, we removed it from consideration as it is not clear that we have sufficient data to draw useful conclusions. To understand the origin of the classification error, we checked the tool mentioned in the FSE paper, GitHub Linguist.[2] At the time of the original study,

[2]https://github.com/github/linguist

that version of Linguist incorrectly classified translation files as TypeScript. This was fixed on December 6th, 2014. This may explain why the number of TypeScript projects decreased between the FSE and CACM papers.

4.1.3 Accounting for C++ and C. Further investigation revealed that the input data only included C++ commits to files with the `.cpp` extension. However, C++ compilers allow many extensions, including `.C`, `.cc`, `.CPP`, `.c++`, `.cp`, and `.cxx`. Moreover, the dataset contained no commits to `.h` header files. However, these files regularly contain executable code such as inline functions in C and templates in C++. We could not repair this without getting additional data and writing a tool to label the commits in the same way as the authors did. We checked GitHub Linguist to explain the missing files, but as of 2014, it was able to recognize header files and all C++ extensions.

The only correction we applied was to delete the `V8` project. While V8 is written mostly in C++, its commits in the dataset are mostly in JavaScript (Fig. 4 gives the number of commits per language in the dataset for the V8 project). Manual inspection revealed that JavaScript commits were regression test cases for errors in the missing C++ code. Including them would artificially increase the number of JavaScript errors. The original authors may have noticed a discrepancy as they removed V8 from RQ3.

	Commits
C	16
C++	7
Python	488
JavaScript	2,907

Fig. 4. V8 commits.

At the end of the data cleaning steps, the dataset had 708 projects, 58.2 million lines of code, and 1.4 million commits—of which 517,770 were labeled as bug-fixing commits, written by 46 thousand authors. Overall, our cleaning reduced the corpus by 6.14%. Fig. 5 shows the relationship between commits and bug fixes in all of the languages after the cleaning. As one would expect, the number of bug-fixing commits correlated to the number of commits. The figure also shows that the majority of commits in the corpus came from C and C++. Perl is an outlier because most of its commits were missing from the corpus.

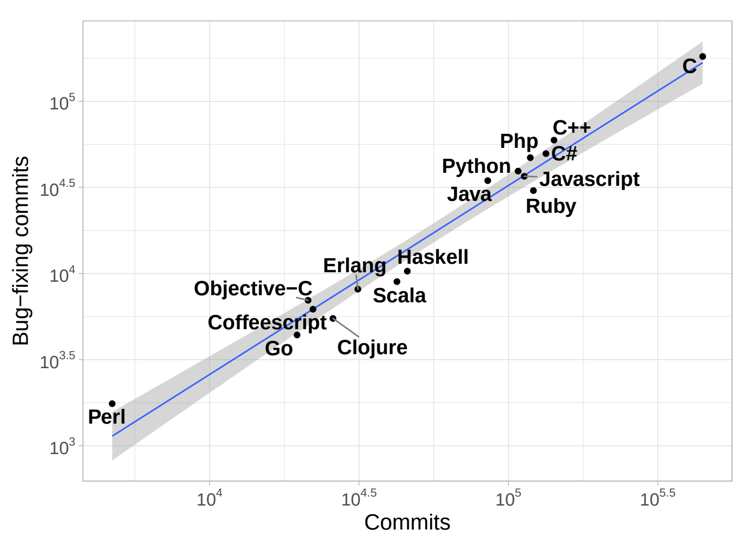

Fig. 5. Commits and bug-fixing commits after cleaning, plotted with a 95% confidence interval.

4.1.4 Labeling Accuracy. A key reanalysis question for this case study is: *What is a bug-fixing commit?* With the help of 10 independent developers employed in industry, we compared the manual labels of randomly selected commits to those obtained automatically in the FSE paper. We selected a random subset of 400 commits via the following protocol. First, randomly sample 20 projects. In these projects, randomly sample 10 commits labeled as bug-fixing and 10 commits not labeled as bug-fixing. Enlisting help from 10 independent developers employed in industry, we omitted the commits' bugfix labels and divided them equally among the ten experts. Each commit was manually given a new binary bugfix label by 3 of the experts, according to their best judgment. Commits with at least 2 bugfix votes were considered to be bug fixes. The review suggested a false positive rate of 36%; *i.e.*, 36% of the commits that the original study considered as bug-fixing were in fact not. The false negative rate was 11%. Short of relabeling the entire dataset manually, there was nothing we could do to improve the labeling accuracy. Therefore, we chose an alternative route and took labeling inaccuracy into account as part of the statistical modeling and analysis.

We give five examples of commits that were labeled as bug fixing in the FSE paper but were deemed by developers not to be bug fixes. Each line contains the text of the commit, underlined emphasis is ours and indicates the likely reason the commit was labeled as a bug fix (when apparent), and the URL points to the commit in GitHub:

- `tabs to spaces formatting fixes.`
  https://langstudy.page.link/gM7N
- `better error messages.`
  https://langstudy.page.link/XktS
- `Converted CoreDataRecipes sample to MagicalRecordRecipes sample application.`
  https://langstudy.page.link/iNhr
- `[core] Add NIError.h/m.`
  https://langstudy.page.link/n7Yf
- `Add lazyness to infix operators.`
  https://langstudy.page.link/2qPk

Unanimous mislabelings (when all three developers agreed) constituted 54% of the false positives. To control for random interrater agreement, we compute Cohen's Kappa coefficient. We calculate kappa coefficients for all pairs of raters on the subset of commits they both reviewed. All values were positive with a median of 0.6. Within the false positives, most of the mislabeling arose because words that were synonymous with or related to bugs (e.g., "fix" and "error") were found within substrings or matched completely out of context. A meta-analysis of the false positives suggests the following six categories:

(1) *Substrings*;
(2) *Non-functional:* meaning-preserving refactoring, e.g. changes to variable names;
(3) *Comments:* changes to comments, formatting, etc.;
(4) *Feature:* feature enhancements;
(5) *Mismatch*: keywords used in an unambiguous non-bug context (e.g., "this isn't a bug");
(6) *Hidden features:* new features with unclear commit messages.

The original study clarified that its classification, which involved identifying bugfixes by only searching for error-related keywords came from [19]. However, that work classified modification requests with an iterative, multi-step process, which differentiates between six different types of code changes through multiple keywords. It is possible that this process was planned but not completed in the FSE publication.

It is noteworthy that the above concerned are well known in the software engineering community. Since the Mockus and Votta paper [19], a number of authors have observed that using keywords appearing in commit message is error prone, and that biased error messages can lead to erroneous conclusions [2, 12, 28] (paper [2] has amongst its authors two of the authors of FSE'14). Yet, keyword based bug-fix detection is still a common practice [3, 6].

4.2 Methods: Statistical Modeling

The reanalysis uncovered several methodological weaknesses in the statistical analyses of the original manuscript.

4.2.1 Zero-sum contrasts. The original manuscript chose to code the programming languages with weighted contrasts. Such contrasts interpret the coefficients of the Negative Binomial Regression as deviations of the log-expected number of bug-fixing commits in a language from the average of the log-expected number of bug-fixing commits *in the dataset*. Comparison to the dataset average is sensitive to changes in the dataset composition, makes the reference unstable, and compromises the interpretability of the results. This is particularly important when the composition of the dataset is subject to uncertainty, as discussed in Sec. 4.1 above. A more common choice is to code factors such as programming languages with zero-sum contrasts [17]. This coding interprets the parameters as the deviations of the log-expected number of bug-fixing commits in a language from the average of log-expected number of bug-fixing commits *between the languages*. It is more appropriate for this investigation.

4.2.2 Multiplicity of hypothesis testing. A common mistake in data-driven software engineering is to fail to account for multiple hypothesis testing [27]. When simultaneously testing multiple hypotheses, some p-values can fall in the significance range by random chance. This is certainly true for Negative Binomial Regression, when we simultaneously test 16 hypotheses of coefficients associated with 16 programming languages being 0 [17]. Comparing 16 independent p-values to a significance cutoff of, say, 0.05 in absence of the associations implies the family-wise error rate (*i.e.*, the probability of at least one false positive association) FWER $= 1 - (1 - 0.05)^{16} = 0.56$. The simplest approach to control FWER is the method of Bonferroni, which compares the p-values to the significance cutoff divided by the number of hypotheses. Therefore, with this approach, we viewed the parameters as "statistically significant" only if their p-values were below $0.01/16 = .000625$.

The FWER criterion is often viewed as overly conservative. An alternative criterion is the False Discovery Rate (FDR), which allows an average pre-specified proportion of false positives in the list of "statistically significant" tests. For comparison, we also adjusted the p-values to control the FDR using the method of Benjamini and Hochberg [1]. An adjusted p-value cutoff of, say, 0.05 implies an average 5% of false positives in the "statistically significant" list.

As we will show next, for our dataset, both of these techniques agree in that they decrease the number of statistically significant associations between languages and defects by one (Ruby is not significant when we adjust for multiple hypothesis testing).

4.2.3 Statistical significance versus practical significance. The FSE paper focused on the statistical significance of the regression coefficients. This is quite narrow, in that the p-values are largely driven by the number of observations in the dataset [11]. Small p-values do not necessarily imply practically important associations [4, 30]. In contrast, *practical significance* can be assessed by examining model-based *prediction intervals* [17], which predict future commits. Prediction intervals are similar to confidence intervals in reflecting model-based uncertainty. They are different from confidence intervals in that they characterize the plausible range of values of the future individual data points (as opposed to their mean). In this case study, we contrasted confidence intervals and prediction intervals derived for individual languages from the Negative Binomial Regression. As above, we used the method of Bonferroni to adjust the confidence levels for the multiplicity of languages.

4.2.4 Accounting for uncertainty. The FSE analyses assumed that the counts of bug-fixing commits had no error. However, labeling of commits is subject to uncertainty: the heuristic used to label commits has many false positives, which must be factored into the results. A relatively simple approach to achieve this relies on parameter estimation by a statistical procedure called the bootstrap [17]. We implemented the bootstrap with the following three steps. First, we sampled with replacement the projects (and their attributes) to create resampled datasets of the same size. Second, the number of bug-fixing commits bcommits^*_i of project i in the resampled dataset was generated as the following random variable:

$$\text{bcommits}^*_i \sim \text{Binom}(\text{size} = \text{bcommits}_i,\ \text{prob} = 1 - \text{FP}) + \text{Binom}(\text{size} = (\text{commits}_i - \text{bcommits}_i),\ \text{prob} = \text{FN})$$

where FP=36% and FN=11% (Section 4.1). Finally, we analyzed the resampled dataset with Negative Binomial Regression. The three steps were repeated 100,000 times to create the histograms of estimates of each regression coefficients. Applying the Bonferroni correction, the parameter was viewed as statistically significant if 0.01/16th and (1-0.01)/16th quantiles of the histograms did not include 0.

4.3 Results

Table 6(b-e) summarizes the re-analysis results. The impact of the data cleaning, without multiple hypothesis testing, is illustrated by column (b). Grey cells indicate disagreement with the conclusion of the original work. As can be seen, the p-values for C, Objective-C, JavaScript, TypeScript, PHP, and Python all fall outside of the "significant" range of values, even without the multiplicity adjustment. Thus, 6 of the original 11 claims are discarded at this stage. Column (c) illustrates the impact of correction for multiple hypothesis testing. Controlling the FDR increased the p-values slightly, but did not invalidate additional claims. However, FDR comes at the expense of more potential false positive associations. Using the Bonferroni adjustment does not change the outcome. In both cases, the p-value for one additional language, Ruby, loses its significance.

Table 6, column (d) illustrates the impact of coding the programming languages in the model with zero-sum contrasts. As can be seen, this did not qualitatively change the conclusions. Table 6(e) summarizes the average estimates of coefficients across the bootstrap repetitions, and their standard errors. It shows that accounting for the additional uncertainty further shrunk the estimates closer to 0. In addition, Scala is now out of the statistically significant set.

Prediction intervals. Even though some of the coefficients may be viewed as statistically significantly different from 0, they may or may not be practically significant. We illustrate this in Fig. 6. The panels of the figure plot model-based predictions of the number of bug-fixing commits as function of commits for two extreme cases: C++ (most bugs) commits) and Clojure (least bugs). Age, size, and number of developers were fixed to the median values in the revised dataset. Fig. 6(a) plots model-based confidence intervals of the *expected values*, *i.e.*, the estimated average numbers of bug-fixing commits in the underlying population of commits, on the log-log scale considered by the model. The differences between the averages were consistently small. Fig. 6(b) displays the model-based *prediction intervals*, which consider individual observations rather than averages, and characterize the plausible future values of projects' bug-fixing commits. As can be seen, the prediction intervals substantially overlap, indicating that, despite their statistical significance, the practical difference in the future numbers of bug-fixing commits is small. Fig. 6(c)-(d) translate the confidence and the intervals on the original scale and make the same point.

4.4 Outcome

The reanalysis failed to validate most of the claims of [26]. As Table 6(d-f) shows, the multiple steps of data cleaning and improved statistical modeling invalidated the significance of 7 out of 11 languages. Even when the associations are statistically significant, their practical significance is small.

Table 6. Negative Binomial Regression for Languages (grey indicates disagreement with the conclusion of the original work)

	Original Authors		Reanalysis							
	(a) FSE [26]		(b) cleaned data		(c) pV adjusted		(d) zero-sum		(e) bootstrap	
	Coef	P-val	Coef	P-val	FDR	Bonf	Coef	Bonf	Coef	sig.
Intercept	-1.93	<0.001	-1.93	<0.001	–	–	-1.96	–	-1.79	*
log commits	2.26	<0.001	0.94	<0.001	–	–	0.94	–	0.96	*
log age	0.11	<0.01	0.05	<0.01	–	–	0.05	–	0.03	
log size	0.05	<0.05	0.04	<0.05	–	–	0.04	–	0.03	*
log devs	0.16	<0.001	0.09	<0.001	–	–	0.09	–	0.05	*
C	0.15	<0.001	0.11	0.007	0.017	0.118	0.14	0.017	0.08	
C++	0.23	<0.001	0.23	<0.001	<0.01	<0.01	0.26	<0.01	0.16	*
C#	0.03	–	-0.01	0.85	0.85	1	0.02	1	0	
Objective-C	0.18	<0.001	0.14	0.005	0.013	0.079	0.17	0.011	0.1	
Go	-0.08	–	-0.1	0.098	0.157	1	-0.07	1	-0.04	
Java	-0.01	–	-0.06	0.199	0.289	1	-0.03	1	-0.02	
Coffeescript	-0.07	–	0.06	0.261	0.322	1	0.09	1	0.04	
Javascript	0.06	<0.01	0.03	0.219	0.292	1	0.06	0.719	0.03	
Typescript	-0.43	<0.001	–	–	–	–	–	–	–	–
Ruby	-0.15	<0.05	-0.15	<0.05	<0.01	0.017	-0.12	0.134	-0.08	*
Php	0.15	<0.001	0.1	0.039	0.075	0.629	0.13	0.122	0.07	
Python	0.1	<0.01	0.08	0.042	0.075	0.673	0.1	0.109	0.06	
Perl	-0.15	–	-0.08	0.366	0.419	1	-0.05	1	0	
Clojure	-0.29	<0.001	-0.31	<0.001	<0.01	<0.01	-0.28	<0.01	-0.15	*
Erlang	0	–	-0.02	0.687	0.733	1	0.01	1	-0.01	
Haskell	-0.23	<0.001	-0.23	<0.001	<0.01	<0.01	-0.2	<0.01	-0.12	*
Scala	-0.28	<0.001	-0.25	<0.001	<0.01	<0.01	-0.22	<0.01	-0.13	

5 FOLLOW UP WORK

We now list several issues that may further endanger the validity of the causal conclusions of the original manuscript. We have not controlled for their impact; we leave that to follow up work.

5.1 Regression Tests

Tests are relatively common in large projects. We discovered that 16.2% of files are tests (801,248 files) by matching file names to the regular expression "`*(Test|test)*`". We sampled 100 of these files randomly and verified that every one indeed contained regression tests. Tests are regularly modified to adapt to changes in API, to include new checks. Their commits may or may not be relevant, as bugs in tests may be very different from bugs in normal code. Furthermore, counting tests could lead to double counting bugs (that is, the bug fix and the test could end up being two commits). Overall, more study is required to understand how to treat tests when analyzing large scale repositories.

5.2 Distribution of Labeling Errors

Given the inaccuracy of automated bug labeling techniques, it is quite possible that a significant portion of the bugs being analyzed are not bugs at all. We have shown how to accommodate for that uncertainty, but our correction assumed a somewhat uniform distribution of labeling errors across languages and projects. Of course, there is no guarantee that labeling errors have a uniform distribution. Error rates may be influenced by practices such as using a template for commits. For instance, if a project used the word `issue` in their commit template, then automated tools would classify all commits from that project as being bugs. To take a concrete example, consider

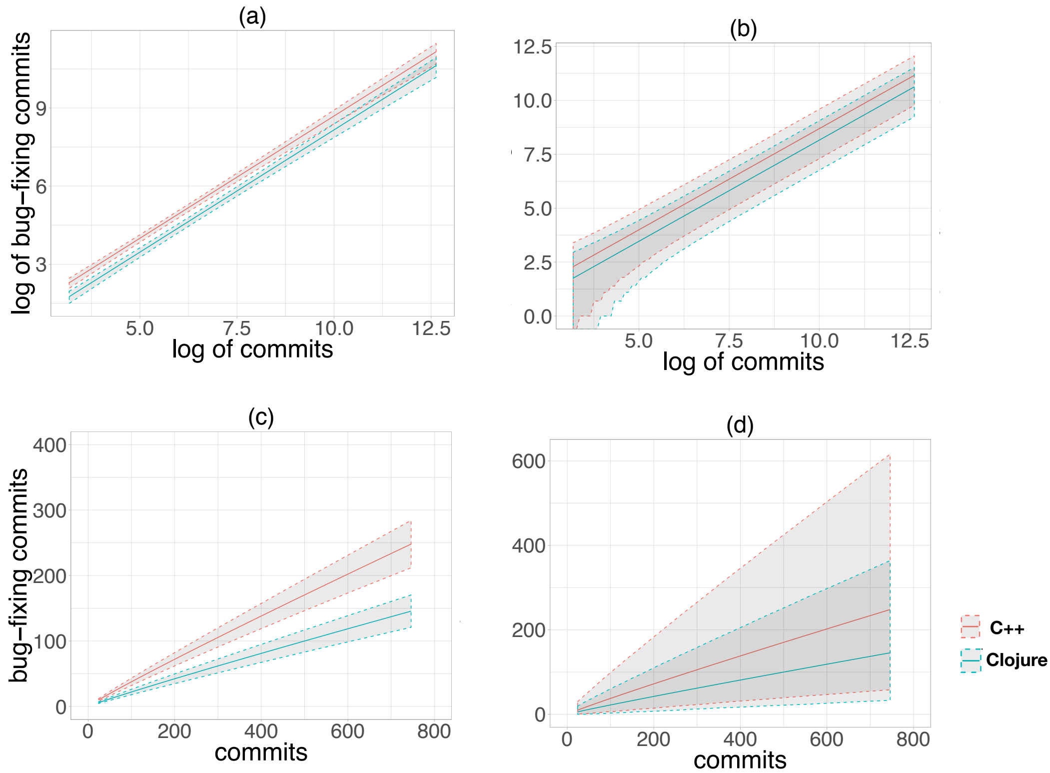

Fig. 6. Predictions of bug-fixing commits as function of commits by models in Table 6(c-d) for C++ (most bugs) and Clojure (least bugs). (a) (1-0.01/16%) confidence intervals for expected values on log-log scale. (b) Prediction intervals for a future number of bug-fixing commits, represented by 0.01/16 and $1 - 0.01/16$ quantiles of the NB distributions with expected values in (a). (c)–(d): translation of the confidence and prediction intervals to the original scale.

the `DesignPatternsPHP` project: it has 80% false positives, while more structured projects such as `tengine` have only 10% false positives. Often, the indicative factor was as mundane as the wording used in commit messages. The `gocode` project, the project with the most false negatives, at 40%, "closes" its issues instead of "fixing" them. Mitigation would require manual inspection of commit messages and sometimes even of the source code. In our experience, professional programmers can make this determination in, on average, 2 minutes. Unfortunately, this would translate to 23 person-months to label the entire corpus.

5.3 Project selection

Using GitHub stars to select projects is fraught with perils as the 18 variants of `bitcoin` included in the study attest. Projects should be representative of the language they are written in. The `PHPDesignPatterns` is an educational compendium of code snippets; it is quite likely that is does represent actual PHP code in the wild. The `DefinitelyTyped` TypeScript project is a popular list of type signatures with no runnable code; it has bugs, but they are mistakes in the types assigned to function arguments and not programming errors. Random sampling of GitHub projects is not an appropriate methodology either. GitHub has large numbers of duplicate and partially duplicated projects [18] and too many throwaway projects for this to yield the intended result. To mitigate this threat, researchers must develop a methodology for selecting projects that represent the population of interest. For relatively small numbers of projects, less than 1,000, as in the FSE paper, it is conceivable to curate them manually. Larger studies will need automated techniques.

5.4 Project provenance

GitHub public projects tend to be written by volunteers working in open source rather than by programmers working in industry. The work on many of these projects is likely done by individuals (or collections of individuals) rather than by close knit teams. If this is the case, this may impact the likelihood of any commit being a bug fix. One could imagine commercial software being developed according to more rigorous software engineering standards. To mitigate for this threat, one should add commercial projects to the corpus and check if they have different defect characteristics. If this is not possible, then one should qualify the claims by describing the characteristics of the developer population.

5.5 Application domain

Some tasks, such as system programming, may be inherently more challenging and error prone than others. Thus, it is likely that the source code of an operating system has different characteristics in terms of errors than that of a game designed to run in a browser. Also, due to non-functional requirements, the developers of an operating system may be constrained in their choice of languages (typically unmanaged system languages). The results reported in the FSE paper suggest that this intuition is wrong. We wonder if the choice of domains and the assignment of projects to domains could be an issue. A closer look may yield interesting observations.

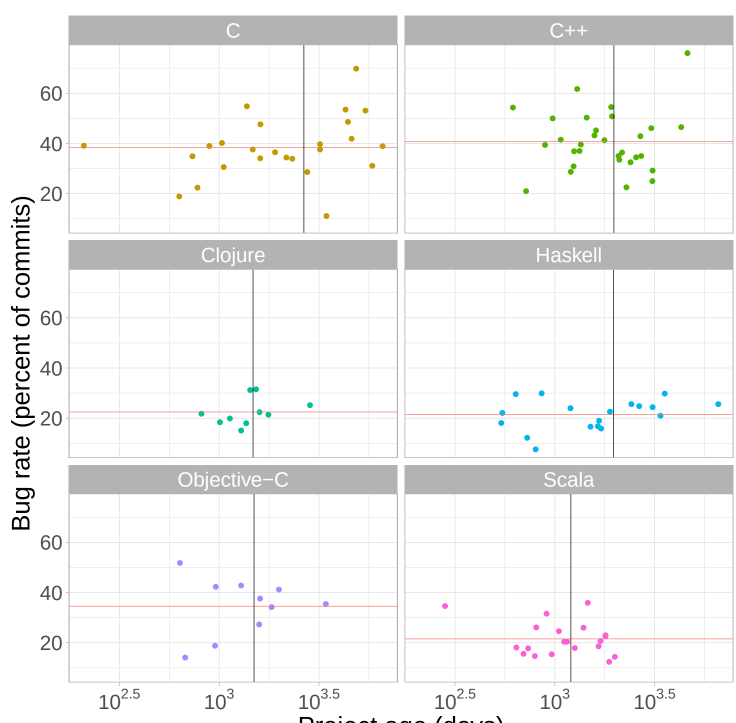

Fig. 7. Bug rate vs. project age. Lines indicate means of project age (x-axis) and bug rate (y-axis).

5.6 Uncontrolled influences

Additional sources of bias and confounding should be appropriately controlled. The bug rate (number of bug-fixing commits divided by total commits) in a project can be influenced by the project's culture, the age of commits, or the individual developers working on it. Consider Fig. 7, which shows that project ages are not uniformly distributed: some languages have been in widespread use longer than others. The relation between age and its bug rate is subtle. It needs to be studied, and age should be factored into the selection of projects for inclusion in the study. Fig. 8 illustrates the evolution of the bug rate (with the original study's flawed notion of bugs) over time for 12 large projects written in various languages. While the projects have different ages, there are clear trends. Generally, bug rates decrease over time. Thus, older projects may have a smaller ratio of bugs, making the language they are written in appear less error-prone. Lastly, the FSE paper did not control for developers influencing multiple projects. While there are over 45K developers, 10% of these developers are responsible for 50% of the commits. Furthermore, the mean number of projects that a developer commits to is 1.2. This result indicates that projects are not independent. To mitigate those threats, further study is needed to understand the impact of these and other potential biases, and to design experiments that take them into account.

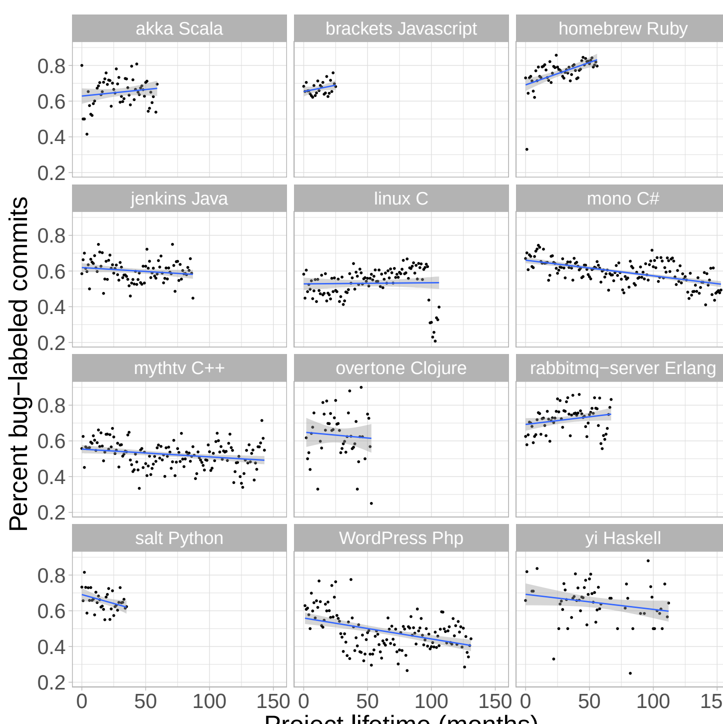

Fig. 8. Monthly avg. bug rate over lifetime. Points are % of bug-labeled commits, aggregated over months.

5.7 Relevance to the RQ

The FSE paper argues that programming language features are, in part, responsible for bugs. Clearly, this only applies to a certain class of programming errors: those that rely on language features. It is unclear if bugs related to application logic or characteristics of the problem domain are always affected by the programming language. For example, setting the wrong TCP port on a network connection is not a language-related bug, and no language feature will prevent that bug,whereas passing an argument of the wrong data type may be if the language has a static type system. It is eminently possible that some significant portion of bugs are in fact not affected by language features. To mitigate this threat, one would need to develop a new classification of bugs that distinguishes between bugs that may be related to the choice of language and those that are not. It is unclear what attributes of a bug would be used for this purpose and quite unlikely that the process could be conducted without manual inspection of the source code.

6 BEST PRACTICES

The lessons from this work mirror the challenges of reproducible data science. While these lessons are not novel, they may be worth repeating.

6.1 Automate, document, and share

The first lesson touches upon the process of collecting, managing, and interpreting data. Real-world problems are complex, and produce rich, nuanced, and noisy datasets. Analysis pipelines must be carefully engineered to avoid corruption, errors, and unwarranted interpretations. This turned out to be a major hurdle for the FSE paper. Uncovering these issues on our side was a substantial effort (approximately 5 person-months).

Data science pipelines are often complex: they use multiple languages, and perform sophisticated transformations of the data to eliminate invalid inputs and format the data for analysis. For instance, this paper relies on a combination of JavaScript, R, shell, and Makefiles. The R code contains over 130 transformation operations over the input table. Such pipelines can contain subtle errors—one of the downsides of statistical languages is that they almost always yield a value. Publications often do not have the space to fully describe all the statistical steps undertaken. For instance, the FSE paper did not explain the computation of weights for NBR in sufficient detail for reproduction. Access to the code was key to understanding. However, even with the source code, we were not able to repeat the FSE results—the code had suffered from bit rot and did not run correctly on the data at hand. The only way forward is to ensure that all data analysis studies be (a) automated, (b) documented, and (c) shared. Automation is crucial to ensure repetition and that, given a change in the data, all graphs and results can be regenerated. Documentation helps understanding the analysis. A pile of inscrutable code has little value.

6.2 Apply domain knowledge

Work in this space requires expertise in a number of disparate areas. Domain knowledge is critical when examining and understanding projects. Domain experts would have immediately taken issue with the misclassifications of V8 and bitcoin. Similarly, the classification of Scala as a purely functional language or of Objective-C as a manually managed language would have been red flags. Finally, given the subtleties of Git, researchers familiar with that system would likely have counseled against simply throwing away merges. We recognize the challenge of developing expertise in all relevant technologies and concepts. At a minimum, domain experts should be enlisted to vet claims.

6.3 Grep considered harmful

Simple bug identification techniques are too blunt to provide useful answers. This problem was compounded by the fact that the search for keywords did not look for words and instead captured substrings wholly unrelated to software defects. When the accuracy of classification is as low as 36%, it becomes difficult to argue that results with small effect sizes are meaningful as they may be indistinguishable from noise. If such classification techniques are to be employed, then a careful *post hoc* validation by hand should be conducted by domain experts.

6.4 Sanitize and validate

Real-world data is messy. Much of the effort in this reproduction was invested in gaining a thorough understanding of the dataset, finding oddities and surprising features in it, and then sanitizing the dataset to only include clean and tidy data [10]. For every flaw that we uncovered in the original study and documented here, we developed many more hypotheses that did not pan out. The process can be thought of as detective work—looking for clues, trying to guess possible culprits, and assembling proof.

6.5 Be wary of p-values

Our last advice touches upon data modeling, and model-based conclusions. Complicated problems require complicated statistical analyses, which in turn may fail for complicated reasons. A narrow focus on statistical significance can undermine results. These issues are well understood by the statistical community, and are summarized in a recent statement of the American Statistical Association [30]. The statement makes points such as "*Scientific conclusions should not be based only on whether a p-value passes a specific threshold*" and "*A p-value, or statistical significance, does not measure the importance of a result.*" The underlying context, such as domain knowledge, data quality, and the intended use of the results, are key for the validity of the results.

7 CONCLUSION

The Ray *et al.* work aimed to provide evidence for one of the fundamental assumptions in programming language research, which is that language design matters. For decades, paper after paper was published based on this very assumption, but the assumption itself still has not been validated. The attention the FSE and CACM papers received, including our reproduction study, directly follows from the community's desire for answers.

Unfortunately, our work has identified numerous and serious methodological flaws in the FSE study that invalidated its key result. Our intent is not to blame. Statistical analysis of software based on large-scale code repositories is challenging. There are many opportunities for errors to creep in. We spent over 6 months simply to recreate and validate each step of the original paper. Given the importance of the questions being addressed, we believe it was time well spent. Our contribution not only sets the record straight, but more importantly, provides thorough analysis and discussion of the pitfalls associated with statistical analysis of large code bases. Our study should lend support both to authors of similar papers in the future, as well as to reviewers of such work.

After data cleaning and a thorough reanalysis, we have shown that the conclusions of the FSE and CACM papers do not hold. It is not the case that eleven programming languages have statistically significant associations with bugs. An association can be observed for only four languages, and even then, that association is exceedingly small. Moreover, we have identified many uncontrolled sources of potential bias. We emphasize that our results do not stem from a lack of data, but rather from the quality of the data at hand.

Finally, we would like to reiterate the need for automated and reproducible studies. While statistical analysis combined with large data corpora is a powerful tool that may answer even the hardest research questions, the work involved in such studies—and therefore the possibility of errors—is enormous. It is only through careful re-validation of such studies that the broader community may gain trust in these results and get better insight into the problems and solutions associated with such studies.

Acknowledgments. We thank Baishakhi Ray and Vladimir Filkov for sharing the data and code of their FSE paper; had they not preserved the original files and part of their code, reproduction would have been more challenging. We thank Derek Jones, Shiram Krishnamurthi, Ryan Culppeper, Artem Pelenitsyn for helpful comments. We thank the members of the PRL lab in Boston and Prague for additional comments and encouragements. We thank the developers who kindly helped us label commit messages. This work received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (awards 1518844, 1544542, and 1617892), and the Czech Ministry of Education, Youth and Sports (grant agreement CZ.02.1.010.00.015_0030000421).

REFERENCES

[1] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995). https://doi.org/10.2307/2346101

[2] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. 2009. Fair and Balanced?: Bias in Bug-fix Datasets. In *Symposium on The Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/1595696.1595716

[3] Casey Casalnuovo, Yagnik Suchak, Baishakhi Ray, and Cindy Rubio-González. 2017. GitcProc: A Tool for Processing and Classifying GitHub Commits. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3092703.3098230

[4] David Colquhoun. 2017. The reproducibility of research and the misinterpretation of p-values. *R. Soc. open sci.* 4, 171085 (2017). https://doi.org/10.1098/rsos.171085

[5] Premkumar T. Devanbu. 2018. Research Statement. www.cs.ucdavis.edu/~devanbu/research.pdf

[6] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2013.6606588

[7] J.J. Faraway. 2016. *Extending the Linear Model with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models*. CRC Press. https://books.google.cz/books?id=XAzYCwAAQBAJ

[8] Dror G. Feitelson. 2015. From Repeatability to Reproducibility and Corroboration. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015). https://doi.org/10.1145/2723872.2723875

[9] Omar S. Gómez, Natalia Juristo Juzgado, and Sira Vegas. 2010. Replications types in experimental disciplines. In *Symposium on Empirical Software Engineering and Measurement (ESEM)*. https://doi.org/10.1145/1852786.1852790

[10] Garrett Grolemund and Hadley Wickham. 2017. *R for Data Science*. O'Reilly.

[11] Lewis G. Halsey, Douglas Curran-Everett, Sarah L. Vowler, and Gordon B. Drummond. 2015. The fickle P-value generates irreproducible results. *Nature Methods* 12 (2015). https://doi.org/10.1038/nmeth.3288

[12] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2013.6606585

[13] John Ioannidis. 2005. Why most published research findings are false. *PLoS Med* 2, 8 (2005). https://doi.org/10.1371/journal.pmed.0020124

[14] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3243734.3243804

[15] Paul Krill. 2014. Functional languages rack up best scores for software quality. *InfoWorld* (November 2014). https://www.infoworld.com/article/2844268/application-development/functional-languagesrack-up-best-scores-software-quality.html

[16] Shriram Krishnamurthi and Jan Vitek. 2015. The real software crisis: repeatability as a core value. *Commun. ACM* 58, 3 (2015). https://doi.org/10.1145/2658987

[17] Michael H. Kutner, John Neter, Christopher J. Nachtsheim, and William Li. 2004. *Applied Linear Statistical Models*. McGraw-Hill Education. https://books.google.cz/books?id=XAzYCwAAQBAJ

[18] Crista Lopes, Petr Maj, Pedro Martins, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. Déjà Vu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* OOPSLA (2017). https://doi.org/10.1145/3133908

[19] Audris Mockus and Lawrence Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *International Conference on Software Maintenance (ICSM)*. https://doi.org/10.1109/ICSM.2000.883028

[20] Martin Monperrus. 2014. A Critical Review of "Automatic Patch Generation Learned from Human-written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2568225.2568324

[21] Sebastian Nanz and Carlo A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code. In *International Conference on Software Engineering (ICSE)*. http://dl.acm.org/citation.cfm?id=2818754.2818848

[22] Roger Peng. 2011. Reproducible Research in Computational Science. *Science* 334, 1226 (2011). https://doi.org/10.1126/science.1213847

[23] Dong Qiu, Bixin Li, Earl T. Barr, and Zhendong Su. 2017. Understanding the syntactic rule usage in Java. *Journal of Systems and Software* 123 (2017), 160 – 172. https://doi.org/10.1016/j.jss.2016.10.017

[24] B. Ray and D. Posnett. 2016. A large ecosystem study to understand the effect of programming languages on code quality. In *Perspectives on Data Science for Software Engineering*. Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-804206-9.00023-4

[25] Baishakhi Ray, Daryl Posnett, Premkumar T. Devanbu, and Vladimir Filkov. 2017. A large-scale study of programming languages and code quality in GitHub. *Commun. ACM* 60, 10 (2017). https://doi.org/10.1145/3126905

[26] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar T. Devanbu. 2014. A large scale study of programming languages and code quality in GitHub. In *International Symposium on Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/2635868.2635922

[27] Rolando P. Reyes, Oscar Dieste, Efraín R. Fonseca, and Natalia Juristo. 2018. Statistical Errors in Software Engineering Experiments: A Preliminary Literature Review. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3180155.3180161

[28] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux Bug Fixing Patches. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2012.6227176

[29] Jan Vitek and Tomas Kalibera. 2011. Repeatability, reproducibility, and rigor in systems research. In *International Conference on Embedded Software (EMSOFT)*. 33–38. https://doi.org/10.1145/2038642.2038650

[30] Ronald L. Wasserstein and Nicole A. Lazar. 2016. The ASA's Statement on p-Values: Context, Process, and Purpose. *The American Statistician* 70, 2 (2016). https://doi.org/10.1080/00031305.2016.1154108

[31] Jie Zhang, Feng Li, Dan Hao, Meng Wang, and Lu Zhang. 2018. How Does Bug-Handling Effort Differ Among Different Programming Languages? *CoRR* abs/1801.01025 (2018). http://arxiv.org/abs/1801.01025