Code Authorship and Fault-proneness of Open-Source Android Applications : An Empirical Study

John Businge
Mbarara University of Science and Technology
P.O. Box 1410
Mbarara, Uganda
johnxu21@gmail.com

Simon Kawuma
Mbarara University of Science and Technology
P.O. Box 1410
Mbarara, Uganda
simon.kawuma@must.ac.ug

Engineer Bainomugisha
Makerere University
P.O. Box 7106
Kampala, Uganda
baino@cis.mak.ac.ug

Foutse Khomh
SWAT Lab., École Polytechnique de Montréal
Montréal, Canada
foutse.khomh@polymtl.ca

Evarist Nabaasa
Mbarara University of Science and Technology
P.O. Box 1410
Mbarara, Uganda
enabaasa@must.ac.ug

ABSTRACT

Context: In recent years, many research studies have shown how human factors play a significant role in the quality of software components. Code authorship metrics have been introduced to establish a chain of responsibility and simplify management when assigning tasks in large and distributed software development teams. Researchers have investigated the relationship between code authorship metrics and fault occurrences in software systems. However, we have observed that these studies have only been carried on large software systems having hundreds to thousands of contributors. In our preliminary investigations on Android applications that are considered to be relatively small, we observed that applications systems are not totally owned by a single developer (as one could expect) and that cases of no clear authorship also exist like in large systems. To this end, we do believe that the Android applications could face the same challenges faced by large software systems and could also benefit from such studies.
Goal: We investigate the extent to which the findings obtained on large software systems applies to Android applications.
Approach: Building on the designs of previous studies, we analyze 278 Android applications carefully selected from GitHub. We extract code authorship metrics from the applications and examine the relationship between code authorship metrics and faults using statistical modeling.
Results: Our analyses confirm most of the previous findings, i.e., Android applications with higher levels of code authorship among contributors experience fewer faults.

KEYWORDS

Software faults; Minor Contributors; Major Contributors; Most Values Contributors; Total Contributors

ACM Reference format:
John Businge, Simon Kawuma, Engineer Bainomugisha, Foutse Khomh, and Evarist Nabaasa. 2017. Code Authorship and Fault-proneness of Open-Source Android Applications : An Empirical Study. In *Proceedings of PROMISE , Toronto, Canada, November 8, 2017,* 10 pages.
https://doi.org/10.1145/3127005.3127009

1 INTRODUCTION AND MOTIVATION

In recent years, a number of studies have shown that human factors play a significant role in software quality [2, 3, 7–9, 12, 14, 15, 19–21]. Code authorship metrics were introduced by Bird et al. [3] to capture developers' contributions in large and distributed software development teams, with the aim to establish a clear chain of responsibility (who to blame in case there is a problem) and simplify management (to whom to assign a task or a bug-fix). Using code authorship metrics, researchers have investigated how different levels of developers' activities on components affect the quality of a software. For example, Bird et al. [3] found that a module with weak code authorship (i.e., that is written by many minor authors) is more likely to have faults in the future. Bird et al. used code authorship metrics to split developers of a software component into two distinct groups: major developers, corresponding to people who authored more than 5% of the contributions and minor developers who are people who authored less than 5% of the code of the component. Greiler et al. [9] argue that a lack of clear code authorship is likely to cause a lack of responsibility on the parts of the code that an engineer does not own.

Although there has been some research examining the relationship between code authorship metrics and the quality of software components, all these studies have been performed on a few (maximum of ten) medium to large sized systems, that were developed by hundreds of developers. We are not aware of any study that investigated code authorship in small sized systems, such as mobile applications. Yet, our preliminary analysis of Android applications, revealed that most applications are not totally owned by a single developer (as one could expect) and that cases of no clear authorship also exist like it was reported for large systems (see Bird et al. [3]). While the findings reported by these previous works are actionable in large software systems, the role of (a lack of) clear code authorship is still unclear for small sized systems. We do believe that relatively small software systems like Android applications could face similar challenges as large software systems and could also possibly benefit from the findings of code authorship studies. For example, Bird et al. [3] observed that high values of minor contributors are associated with more faults in a software system. In this paper, in the context of Android applications, *we investigate whether applications with few major contributors are more–or–less fault-prone than applications with larger numbers of developers that do minor contributions.*

The remainder of this paper is organized as follows: Section 2 presents the background information. Section 3 discusses the experimental setup of our study. Section 4 discusses the results and findings of our study. Section 5 presents threats to the validity, while Section 6 provides an overview of the related work. Finally, Section 7 concludes the paper and outlines some avenues for future work.

2 BACKGROUND & DEFINITIONS

In this section, we give a brief overview of code authorship metrics and explain how previous studies related the metrics to software quality.

2.1 Code Authorship Metrics and Software Quality

Previous studies used the authorship of code changes to estimate the code authorship of a developer for a module [3, 7, 20]. Bird et al. [3] computed a developer's code authorship for a module by calculating the proportion of code changes that the developer has authored within that module. Bird et al. showed that weakly owned Windows binaries where many engineers contributed small amounts of code were more likely to be fault-prone than strongly owned Windows binaries, for both Windows Vista and Windows 7. Furthermore, Bird et al. also observed that the more minor code authors contributed to a software module, the more faults it contained. Rahman and Devanbu [20] computed code authorship values at a finer level of granularity, by calculating the proportion of changed lines that each developer has authored. Rahman and Devanbu observed that code implicated in faults were strongly associated with situations of single developer's contribution. Foucault et al. [7] revisited the theory formulated by Rahman and Devanbu [20] on seven open source software systems, using code authorship metrics proposed by Bird et al. [3] and confirmed the existence of a relationship between code authorship and software quality. However, they also argue that the usefulness of code authorship metrics is debatable since in all their studied systems, they found independent variables to be highly collinear. The main difference between these previous studies and our study is the size of the systems that are investigated, i.e., large vs. small sized systems.

In our study, we use the metrics proposed by Bird et al. [3], i.e., we use the amount of code changes to estimate the code authorship of a developer for an application. Below we state four code authorship metrics that capture the magnitude of the contributions of developers in the applications and one metric that captures software quality. We also provide the rationale for choosing these metrics to assess the potential relation between code authorship and the fault-proneness of applications.

Most valued author (MVA): This metric measures the highest percentage of contributions that a code author has made to a software module. If the MVA of a software project is close to 100%, this means that one author performed almost all the changes in that project. A high value of MVA reveals that the software project has strong authorship while a low value reveals that it has a shared authorship. We expect that an increase in the value of MVA metric may result in a decrease of the number of faults in the project.

Minor Author (Minor-A): The metric counts how many code authors have a ratio of contributions that is lower than a given threshold. If there are lots of minor authors, this implicitly means that many contributions are made by minor contributors and therefore the software project is shared between many code authors. The work is thus fragmented between many code authors with little knowledge of the project they are working on, and therefore overseeing all these contributions becomes an obstacle. We expect that an increase in the value of the Minor-A metric may result in an increase of the number of faults in the project.

Major Author (Major-A): The metric counts how many code authors have a ratio of contributions that is bigger than a given threshold. We rely on Bird et al's. [3] analogy stating that "too many cooks spoil the broth". This means that if there are lots of major authors, they all perform a significant amount of contributions and therefore the software project has a shared authorship which implies that coordinating the work of developers is more difficult. We expect that an increase in the value of the Major-A metric to result in an increase of the number of faults in the project.

Total Authors (Total-A): This metric counts the total number of contributors that have made changes in a software project. It examines the effect of team size on software quality. If the size of a team is too large, coordination may be difficult, which may result in poor quality and faulty code.

Faults: This is the number of faults reported in a given Android project.

Later in our analysis, we will build regression models to identify the relationship between faults and code authorship metrics in the Android applications.

3 EXPERIMENTAL SETUP

This section details the experimental setup. We also define the research goal and questions, descriptions of how we collected the data and how we extracted the metrics.

3.1 Goal & Research Questions

Like the study by Bird et al. [3] we also adopt the Basili's goal question metric approach [1] to frame our study of code authorship and fault-proneness. Our goal is to understand the relationship between code authorship and fault-proneness in Android applications (which are relatively small software systems). In order to reach this goal, we ask two research questions:

RQ1: Are Android applications with *higher values of Minor-A* associated with more *faults* in comparison to those with *lower values of Minor-A*?

RQ2: Are Android applications with *higher values of MVA* less fault-prone than applications with *lower values of MVA*?

To answer the two research questions, we use regression models on a dataset collected from Android open-source applications hosted on Github. Below we present the methodology that was employed to process the dataset. We present the corpus of applications used, as well as how we computed the different metrics used in the models.

3.2 Data Collection and Extraction of Code Authorship and Control Metrics

3.2.1 Data Collection. The data used in the study was carefully extracted from GitHub using the GitHub API. The selection of the GitHub data was based on the following criteria: First, in early March, 2016 when we collected the data, we typed the keyword *Android applications* in the GitHub search engine, which returned about 45,000 repositories. About 40,000 of these were written in the *Java* language. Because we are aware that some of the projects on GitHub are for example written by students as assignments, we wanted to eliminate these repositories as much as we could so as not to pollute our results. For this reason, we carefully collected *repository names* on GitHub with a "*description*" containing the word "*Android application*", having *at least five releases*, *at least three stars* and written in *Java*. Starring a repository allows a developer to keep track of projects that he finds interesting as well as showing appreciation to the repository maintainer for their work[1]. The rationale for using three stars was because we wanted to get as many applications as possible, for statistical significance.

To compute the code authorship metrics, we processed the different JSON objects returned by the Github API. As an illustration example, we use the Android application project `eglaysher/lifecounter`. The Github API `https://api.github.com/repos/eglaysher/lifecounter/commits` returned a JSON object with all the commits of this project. Each commit has got a unique 40 character identifier called `SHA`. For each commit SHA, we use the Github API, for example `https://api.github.com/repos/eglaysher/lifecounter/commits/a807848a5d426aabea59dae4b355706e60228e7a` which returns a JSON object with specific commit details that include: author details (login ID, email address, and full names), all the files that have been modified as well as the number of lines of code (LOC) in each file that have been modified. For each file, we summed up the changed LOC. We thereafter summed up the changed LOC for all the files in a commit. Additionally, for each commit, we also kept track of the author details for the commits. Finally, we summed up the changed LOC of all the commits in a project made by different authors. We also collected other statistics like `longevity`, and `inactivity` to help us summarize the studied projects using descriptive statistics in Section 4.1. `Longevity`–is the time interval in months between the application's "first release date" to the "last commit date" or March 06, 2016 (the last day of collection of the application statistics on GitHub). `Inactivity`—The time interval in months between the application last commit and March 06, 2016. Because we want to build meaningful models, in our data sets we only used projects that have existed for at least one year since we collected this data (i.e., first release dates before March 06, 2015). Using the aforementioned criteria, we retained a total of 278 applications for our study. We share our data set on-line[2], to allow the community to replicate our work.

While collecting the data from GitHub, we observed that some of the applications received a lot of commits from certain contributors in their first release, but no commits from these contributors later on. This possibly means that the development of the application started elsewhere and it was just imported into GitHub. Our reasoning is that, the more contributors an application has after its first release, the higher the likelihood that the application will be maintained frequently on GitHub. To compute code authorship metrics for a given application, we collect information about the commits of all the developers that have contributed to the application *after the first release* of the application until *March, 6 2016*. We also downloaded the source files of the applications from Github and extracted the number of Lines of source code (LOC), using the *cloc*[3] tool.

3.2.2 Name Merging. During data collection, we discovered that some contributors of the applications used more than one account, which makes them appear as different contributors. To address this issue, we performed name merging to ensure that our data is not polluted with duplicate informations that would introduce noise. We merged the details of two contributors into one using the following heuristics in the order mentioned: 1) if they possess the same `login ID`, 2) posses different `login ID` but posses the same `full names`, and 3) possess both different `login ID` and `full names` but have the same `e-mail` prefix.

3.2.3 Code Authorship and Control Metric Extraction. We computed code authorship metric values following the definition proposed by Bird *et al.* [3] and used by Rahman and Devanbu [20], which consists in calculating the proportion of contribution of each author. If C^b lines are changed on a repository rp_1 in a time interval t, and there are a total number of m distinct authors, and the number of lines contributed by author a in the time interval t is C_t^b, then the contribution ratio of a is $r_a^C = \frac{C_t^b}{C^b}$. We then sort the ratios in descending order and thereafter we sum them up as illustrated in Equation 1. $sum_{0.8}$ is the summation of the ratios starting with the largest ratio r_1^C (highest ratio) to r_n^C, where r_n^C is the first ratio where $sum_{0.8} \geqslant 0.8$ and $n \leqslant m$.

Bird *et al.* who studied large scale software systems (i.e., Windows Vista and Windows 7) with nearly one hundred code authors, used a threshold value of 5% to categorize major and minor code

[1]https://help.github.com/articles/about-stars/

[2]https://sites.google.com/site/coauthorship2017/dataset

[3]https://github.com/AlDanial/cloc

contributors (i.e., they considered an author to be a major contributor of a module if they contributed more than 5% of the code of that module, otherwise they are considered to be a minor contributor to the module). Since our work focuses on applications, which are of smaller size, and have fewer developers than these Windows projects, computed differently the threshold value used to decide about major and minor contributors.

A threshold of 5% would be inappropriate because, for example, if an application has two code authors with contribution ratios of 0.93 and 0.07, respectively, applying the threshold of 5% used by Bird et al., would characterize both authors as major contributors, which would be misleading. In our study, using the heuristics presented in Equation 1, we label major authors if they have been categorized in $sum_{0.8}$ and minor authors in the remaining $sum_{0.2}$. In terms of thresholds values, we state that the labeling of major and minor authors is considered at threshold–0.8. To compute the `MVA` metric, we consider the author with the highest ratio. For example, if an application has two code authors having contribution ratios of 0.93 and 0.07, the `MVA` value will be 0.93. However, as shown in Figure 1 (in Section 4.1), we do have cases of low MVA metric meaning that there are cases of shared-authorship of the applications.

$$sum_{0.8} = \sum_{i=1}^{n} r_i \qquad (1)$$

We carried out a *sensitivity analysis* on the experiments by varying the thresholds from 0.7, 0.75, 0.8, 0.85 and 0.9 but the results did not yield significant differences. We decided to use the threshold of 0.8.

Also, we would like to emphasize that we computed all the aforementioned metrics at the project level, i.e., we computed the authorship (respectively the level of contribution) of a developer for the whole application. A detailed explanation of this design decision (i.e., project level vs. module level analysis) can be found at the end of Section 3.3.

3.3 Data Collection and Extraction of Code Quality Metrics

We used the `SonarQube` tool[4] to extract information about faults and security vulnerability experienced by the studied applications. `SonarQube` (previously called Sonar) is an open source quality management platform, dedicated to continuously analyze and measure technical quality, from project portfolio to methods. `SonarQube` is a popular code quality measurement tool that has gone through a number evolutionary versions having its first version released in 2007. The tool is actively maintained on Github as of January 16, 2016 having 21,739 `commits`, 107 `releases`, 55 `contributors`, 160 `watches`, 1,461 `stars`, and 566 `forks`[5]. All the statistics of `SonaQube` on Github show that the tool is very popular among developers and is actively being maintained. The `SonarQube` tool analyzed the files in each application looking for faults, and reported specific points in the file where a fault was observed. The tool categorizes the identified faults into five types: blocker, critical, major, minor, and info.

- *Blocker*: A fault of this kind might make the whole application unstable in production. For example, calling garbage collector, not closing a socket, etc.
- *Critical*: A fault of this kind might lead to an unexpected behavior in production without impacting the integrity of the whole application. For example, NullPointerException, badly caught exceptions.
- *Major*: A fault of this kind might have a substantial impact on productivity. For example, too complex methods, package cycles.
- *Minor*: A fault of this kind might have a potential but minor impact on productivity. For example, finalizer does nothing but call superclass finalizer.
- *Info*: Unknown or not yet well defined security risk which can impact productivity.

The tool counts the number of faults reported in each file and aggregates them to obtain the total number of faults contained in the project. Additionally, the tool further rates the fault-proneness of the whole application as follows: A–Zero faults, B–at least one minor fault, C–at least one major fault, D–at least one critical fault and E–at least one blocker fault. For our dependent variable metric, we consider the total number of faults reported for each of the Android application. We extracted the faults reported on the last release of each of the studied applications.

We would like to state that our study differs slightly from the previous studies of Foucault et al. [7] and Bird et al. [3] on the artifact that was considered. While the previous studies investigate the relationship between code authorship metrics and fault-proneness in a module, our study investigates the relationship between code authorship metrics and fault-proneness in a project. Software modules are units of development within a software project for example file or package. There are two main reasons for the above stated difference in the investigated artifact in the software system: 1) As earlier stated, unlike in the previous studies with relatively large software systems having hundreds of developers, it made sense extracting code authorship metrics and building module-level models. However, since we are investigating Android applications that are much smaller as well as having few authors, extracting code authorship metrics and building for example file-level models would not make a lot of sense. 2) Again, because of the large sizes as well as the very few number of the software systems investigated in the previous studies (i.e., Bird et al.–two software systems and Foucault et al.–seven software systems), models were built for each software system. For example, the data points of the independent variables in the models are the number of code authorship metrics per software module in a software system (if a software system has 100 software modules, then 100 code authorship metrics data points were extracted). As opposed to the way models were built in previous studies, in this study because of the limited size and number of authors in the Android applications, we build models where we consider each application as a data point. Another difference with previous works concerns the faults investigated. We rely on SonarQube to identify faults in the code of applications while previous works extracted faults reported in bug tracking systems.

[4]https://www.sonarqube.org/
[5]https://github.com/SonarSource/sonarqube

3.4 Multiple Linear Regression Models Tuning

In this section, we discuss how we build multiple linear regression models to uncover the relationship between faults and code authorship metrics. Similar to previous related works [3, 4, 14, 21, 23], our main goal for building fault-proneness models is not to predict fault-prone applications, but to understand the relationship between the explanatory variables and the fault-proneness of the applications. Specifically, we use linear regression to enable us to examine the effect of one or more code authorship metrics and source code metrics (when controlling the other variables). In the regression models with faults as the dependent variable, one can observe which variables have an effect on faults, how large the effect is, in what direction (i.e., if number of faults go up when a metric goes up or when it goes down), and how much of the variance in the number of failures is explained by the metrics. We compare the amount of variance in failures explained by a model that includes the code authorship metrics to a model that does not include them. In the models, we use size (i.e., LOC) and complexity metrics (Mc Cabe complexity[6]) as control variables.

Before building the models, we conducted a number of model tuning and preparations in order to have model results that can be trusted. The following are some of the diagnostics that we carried out: First, we standardized the variables by subtracting the values from the mean and dividing by the standard deviation. This allowed our variables to be relatively on the same scale (which is very important since we are building models across different applications). Second, because the interpretation of the models' results can be influenced by the presence of redundant variables. We checked for redundant variables using the `redun` function in the `rms` `R` package [11]. However, we found that none of the explanatory variables that survived our correlation analysis were redundant. Third, we also removed the variables that introduced multicollinearity and over-fitting by considering the Variance Inflation Factor (VIF). All variables in the final models had VIFs of under 5, as guided by standard rule of thumb [17]. Fourth, for the ordinary least squares (OLS) regression to be reliably interpreted, we had to look at normally distribution of the residuals. Non-normality of the residuals is attributable to the skewness of the variables. In our data set, variables that are found to be skewed are log transformed to stabilize the variance and improve the model fit, whenever appropriate [17]. Fifth, to overcome the issue of the ordering of regressors, we assess and report the relative importance of regressors in the multiple regression model using the PMVD technique developed by Feldman [6], which is implemented in the R package relimpo [10]. Finally, we take special care to make sure that the most important modeling assumptions of OLS regression are met, namely: 1) homoscedasticity (by examining *residual vs. fitted* plots), 2) linear independence (mentioned above, by removing highly correlated variables according to VIF), and 3) normality of errors (by examining *normal qq plots*). In addition, we also consult the *Cook's distance vs. leverage plots* to identify any potentially overly influential outliers to examine for validity. This resulted in the removal of six points in our data set which improved the model fit (based on R^2 value) while having minimal effect on the estimated model coefficients (*i.e.,* no variable coefficients changed signs or significance).

[6]https://docs.sonarqube.org/display/SONAR/Metrics+-+Complexity

Table 1 – Descriptive statistics of the study variables.

Variable	Mean	Min	1st Quartile	Median	3rd Quartile	Max
MVA	82%	20%	66%	95%	100%	100%
Total-A	12.3	1	2	3	8	465
Minor-A	10.5	0	0.25	2	6	457
Major-A	1.8	1	1	1	2	16
faults	71.8	0	11.3	32.5	78.5	679
Complexity	2053.1	11	350.8	889.0	2397.8	20511
SizeF	5680.6	14	719.5	2194.5	5601.3	82655
SizeL	9604.6	67	1927.25	4437	10928.5	84265
Longevity	24.4	0.7	14	22.8	34.675	79.2
Inactivity	10.1	0	0.525	4.7	16.425	67.6
Cd'LOC	184160.5	15	8071	24459	95337.5	10331255

faults—Number of faults.
SizeF—Size in Lines of Code of the first release of a project.
SizeL—Size in Lines of Code of the last release of a project.
Cd'LOC—Added + Deleted LOC.

To build the models, we follow a traditional hierarchical approach where we start with a base model that contains only control factors. In subsequent models, we add the various independent measures associated with code authorship metrics. This modeling approach allows us to understand the independent and relative impact on faults, of each set of factors. In order to assess the fit of each model, we report the percentage of variance explained by the model (commonly referred to as the R-squared). We examine the improvement in percentage of variance in the dependent variable explained when we add code authorship metrics to the base model.

4 RESULTS AND DISCUSSION

In this section, we present the results of our experiments. As shall be seen in this section, in our analysis we used a number of methods to examine the relationship between code authorship and the fault-proneness of applications.

4.1 Descriptive Statistics

The first step in our analysis consisted of examining various descriptive statistics of the measures described earlier. Table 1 presents the descriptive statistics of the variables used in the models as well as other variables describing the studied applications. One can draw a number of insights from these descriptive statistics, about the common characteristics exhibited by the studied applications. First, looking at the column values of the *Median* and *Mean* for the variables `Size-F` and `Size-L`, we observe that these are small software systems; as the values are below 10,000 LOC. Second, looking at values of the variables `Size-F` and `Size-L` in the *3rd Quartile* column we also observe that very few applications have a size above 10,000 LOC. Third, looking at values in the column of the *3rd Quartile* for the variables `Total-A`, the values of the column *Median* for `Major-A`, and `Minor-A`, we observe that the software applications considered, indeed comprise few contributors. Although the values of code authorship metrics are low (expected for relatively small systems), studying such a situation is still important in the sense that there are still authors with low contributions, hence with limited knowledge of certain parts of the applications.

Table 2 – Spearman correlation between the different variables used in the study.

Variable	MVA	Minor-A	Major-A	Total-A		SizeF	SizeL	faults	Complexity	Cd'LOC
MVA	1.00	-0.33	-0.75	-0.33		-0.15	-0.22	-0.23	-0.21	-0.19
Minor-A	-0.33	1.00	0.46	0.68		0.31	0.28	0.34	0.28	0.32
Major-A	-0.75	0.46	1.00	0.59		0.11	0.13	0.17	0.12	0.33
Total-A	-0.33	0.68	0.59	1.00		0.19	0.19	0.17	0.18	0.44
SizeF	-0.15	0.31	0.11	0.19		1.00	0.78	0.43	0.80	0.10
SizeL	-0.22	0.28	0.13	0.19		0.78	1.00	0.56	0.99	0.09
faults	-0.23	0.34	0.17	0.17		0.43	0.56	1.00	0.54	0.19
Complexity	-0.21	0.28	0.12	0.18		0.80	0.99	0.54	1.00	0.09
Cd'LOC	-0.19	0.32	0.33	0.44		0.10	0.09	0.19	0.09	1.00

These low contributors could introduce regressions in the applications if they modify areas of the code for which they have little knowledge. Fourth, comparing the values of size (SizeF and SizeL) and Cd'LOC, gives us an indication that the applications have gone through multiple releases along their lifetime, which reduces the possibility that they are student class assignments and not real products. Fifth, looking at the values of variables of `Longevity` and `Inactivity`, we can also observe that the applications have been in existence for a fairly good amount of months. Lastly, looking at the column values of Min, 1st Quartile, Median, 3rd Quartile and Max for all other variables apart from MVA, we observe that the distribution of the variables is heavily rightly skewed. We confirmed this observation on the distribution by drawing box-plots (cf. Figure 1) using the R software. We also observe that the values of code authorship obtained on our studied apps at the project level is similar to the values of code authorship reported by Greiler et al. [9] for the files contained in the four Microsoft projects. This similarity reinforced our decision to conduct our study at the project level instead of component or file levels.

4.2 Correlations

Here we computed Spearman rank correlations presented in Table 2. From the table, we observe that the metric MVA is negatively correlated with most of the other metrics including the number of faults. This implies that the more the code authorship is shared among multiple developers of an application, the higher is the likelihood that the application will contain faults. This observation is reinforced by the fact that for all the studied applications, Total-A and Minor-A metrics are positively correlated with the number of faults (the more people are involved in the development of an application, the higher is the risk of faults). Looking at the individual trends of relationships between code authorship metrics and number of faults, we observe that Minor-A–(0.34) correlates highest with the number of faults, followed by MVA–(-0.23), Major-A, and Total-A–(0.17). Additionally, we also observe high correlations between code attributes like SizeF, SizeL, and complexity with number of faults. The high correlations of size and complexity is not surprising since previous studies have shown that the two variables have a very strong correlation with fault-proneness [5].

Considering the correlations of both code authorship metrics and those of code attributes, as discussed by Bird et al. [3], it is not clear if the increase in number of faults in the applications is attributable to more Minor-A or to measures such as size and

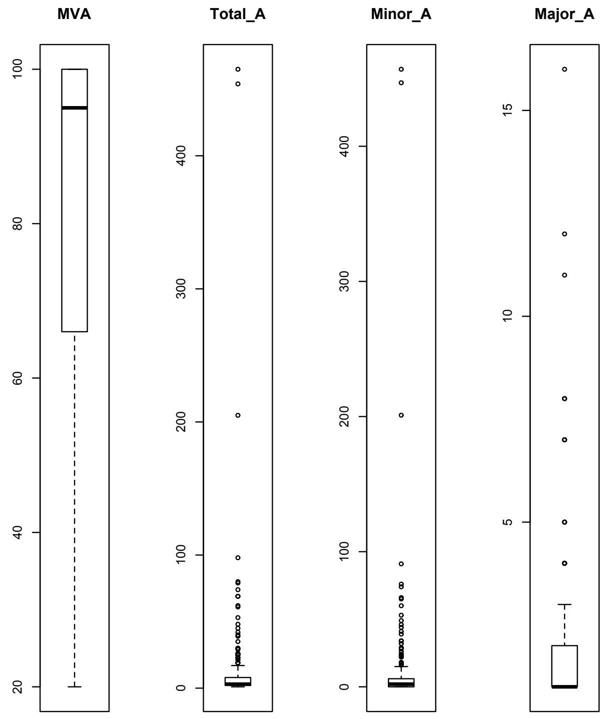

Fig. 1 – Box plots showing the distribution of the code authorship metrics used in the study.

complexity that are also known to be related to faults. To clear this dilemma, we build models of multiple linear regression to observe the effect of each metric on the number of faults when the code authorship and code attribute metrics are used together. Furthermore, prior research has shown that when characteristics such as size are not considered, the omission can affect the validity of observations made for other software metrics [5]. In the next section, we discuss how we used regression models to overcome the anticipated problem. Furthermore, looking at the correlations from Table 2, we observe high correlations between number of `faults` and `SizeL` (> 0.5). In building the models in the next section, we decided to use SizeF instead of SizeL as the control variable since using SizeL introduced over-fitting in the models.

Table 3 – Linear Models coefficients and the sum of squares (ANOVA) for the different variable combinations. The models include standard metrics of size and complexity, as well as the models with code authorship metrics added. An asterisk* in column–*Variance* denotes that a model showed statistically significant improvement when the additional variable was added. Significance codes: 0.000 (***), 0.001 (**), 0.01 (*), 0.05 (.), $>= 0.1$ ()

Model	Description	Variable	Coeff	$Pr(>\|t\|)$	Sum Sq	Importance	Variance
			Base Model				
1	Base	SizeF	0.049	0.473	1.258***	29.6%	39.5%
		Complexity	0.121	0.000***	0.876***	70.4%	
			Base Model + code authorship Heuristics				
2	Base + MVA	SizeF	0.053	0.426	1.129***	26.9%	42.0% (+2.5%)
		Complexity	0.115	0.000***	0.777***	63.0%	
		MVA	-0.023	0.000***	0.364***	10.1%	
3	Base + Total-A	SizeF	0.041	0.538	1.258***	26.5%	41.3%* (+1.8%)
		Complexity	0.115	0.000***	0.780***	63.4%	
		Total-A	0.056	0.005**	0.193***	10.1%	
4	Base + Minor-A	SizeF	0.033	0.619	1.258***	23.5%	43.8%* (+4.3%)
		Complexity	0.109	0.000***	0.680***	56.1%	
		Minor-A	0.056	0.000***	0.429***	20.4%	
5	Base + Minor-A + Major-A	SizeF	0.034	0.606	1.258***	23.0%	44.0%* (+0.2%)
		Complexity	0.109	0.000***	0.679***	55.1%	
		Minor-A	0.048	0.001**	0.429***	16.9%	
		Major-A	0.005	0.369	0.008	5.0%	

4.3 OLS Model Results

In this section, we present the results of the multivariate linear regression analysis for the 5 models we built in our experiments. Table 3 illustrates the results of our analysis. The table presents the values of the different constructs in the models that we built, which includes: 1) *Model*–the models comprising different variable combinations. 2) *Description*–how the models were incrementally built, 3) *Variable*–the different variable combinations considered in each of the models. 4) *Coeff* –the values of the regression coefficients corresponding to each of the variables in the models. The values of the coefficients tell us the change in the number of faults for every unit increase in the independent variable. For example, a value of 0.049 for the variable SizeF in Model–1 tells us that the predicted number of faults in an application will increase by 0.049 for every increase of one LOC in SizeF. 5) $Pr(>|t|)$–The $p - value$ showing the statistical significance of that variable given all the other variables have been entered into the model. The significant codes indicate: (***)–$p = 0.000$, (**)–$p = 0.01$, (*)–$p = 0.05$, and (.)–$p = 0.1$. 6) *Sum Sq*–Sum of Squares associated with the sources of variance of the variables (total variance partitioned into the variance which can be explained by the independent variables) and the residual. The significance code for the *Sum Sq*, like $Pr(>|t|)$, indicate the variables that significantly contribute to the estimate of the dependent variable. 7) *Importance*–This is the relative importance for each of the predictors in the model provided by R `relaimpo` package. For example, in Model–1, Complexity has the highest importance in estimating the dependent variable with a value of 70.4%. 8) *Variance*–This measures the proportion of variance of the dependent variable (i.e., number of faults) explained by the regressors (i.e., code authorship and control metrics) in the model. For example, in Model-1, the proportion of variance explained by the regressors `SizeF` and `Complexity` is 39.5%.

Table 3, column–*Variance*, the asterisk* denotes cases where the goodness-of-fit F-test indicated that the addition of variable improved the model by a statistically significant degree. The value in parenthesis indicates the percentage of increase in variance explained over the model without the added variable. For example, in Model-5–Base + Minor-A + Major-A explains 44.0% of the variance in the number of faults which is 0.2% more than Model-4–Base + Minor-A which explains 43.8%. As stated in Section 3.4, our model building followed a traditional hierarchical approach where we started with a base model containing only control factors–SizeF and Complexity (we refer to this model as the Base model). From Table 3, (*Model-1*:*Importance*), the Base model shows that SizeF and Complexity both have significant effects on the number of faults. In addition, these metrics are able to explain 39.5% of the variance in the number of faults in the software projects we considered. Looking at the *p-values* of SizeF in the column–$Pr(>|t|)$, we observe that they are all > 0.05. This seems to suggest that SizeF do not contribute to the model.

The reason for the insignificant contribution of SizeF in terms of *p-values* is because SizeF and Complexity are highly correlated (c.f., Table 2). However, when considered in isolation of Complexity, the contribution of SizeF on the number of faults is very significant. This can also be observed from column–Importance in Table 3. Furthermore, in order to determine the relative importance of the code authorship metrics, one needs at least two variables in the *Base model* to run the `relaimpo` package in R. In subsequent models (models 2–5), we incrementally added the various independent variables associated with the different research questions. This modeling approach allowed us to understand the importance of the different independent variables on the number of faults in the applications.

RQ1: Are Android applications with *higher values of Minor-A* associated with more *faults* in comparison to those with *lower values of Minor-A*?

From the detailed results—Table 3, we use Model-3, Model-4 and Model-5 to answer RQ1. In Model-3 we add the metric Total-A to the set of predictor variables of the base model to examine the effect of team size on fault-proneness an application. In Model-4 we add the metric Minor-A to the set of predictor variables of the base model to examine the effect of minor contributors on the fault-proneness in an application. In Model-5 we add the metric Major-A to the variables in Model-4 to examine what effect the major contributors have on the fault-proneness of an application. We compare the results of pairs of Model-3 and Model-4 to determine if the total number of code authors has a different effect on the number of faults than the number of minor code authors in the studied applications. The statistics show that the predictor Minor-A has a higher relative importance of 20.4% compared to that of Total-A–10.1%. We also observe that the addition of the Minor-A metric increases the variance explained by 4.3%, whereas the addition of the Total-A metric increases the variance explained by 1.8% only. The gains shown by Minor-A are stronger than those shown by Total-A in estimating the number of faults in the Android applications. This indicates that the number of minor contributors in Android applications have a stronger effect on the fault-proneness of the applications, in comparison to the total number of contributors. Furthermore, the addition of Major-A in Model-5 showed smaller gains, but was still statistically significant. We left out the results of the model Base + Minor-A + Major-A + Total-A since this model produced VIF > 5. Overall, the most significant contributor to the variance explained is Minor-A, followed by Total-A, and lastly by Major-A.

Conclusion RQ1: In Section 2.1 we stated that we expected that an increase in the values of the minor Author metric may result in an increase of the number of faults. The modeling discussed above has revealed that the number of minor code authors have a strong positive relationship with the number of faults even when controlling for classical metrics such as size and complexity. This implies that Android applications with few major contributors are more reliable than applications with larger numbers of contributors where developers do minor contributions. These findings obtained on relatively small sized open source software projects (i.e., the applications) concur with the findings of Bird et al. [3] which were obtained on large commercial software projects. However, while we agree with the finding of Foucault et al. [7] (also conducted on large open source projects) that Minor-A and Total-A metrics are highly collinear, we differ in the conclusion that Minor-A is highly redundant and could be ignored. As we have discussed in our results above, we do state that although Total-A and Minor-A are collinear, we have shown that the individual contribution of each of these variables to the variance of the number of faults, is statistically significant. The difference between our results and those reported by Foucault et al. [7] could possibly be attributed to the differences in the artifact and the experimental design used in the two studies (i.e., module vs project level granularity).

RQ2: Are Android applications with *higher values of MVA* less fault-prone than applications with *lower values of MVA*?

From the results presented in Table 3, we use Model-2 to address RQ2. We observe from the table that the addition of MVA variable in the Base model, i.e., Model-2, significantly improves the variance explained. The added MVA metric examines the impact of the most valuable author on the number of faults. We can also observe that MVA has a significant relative importance of about 10.1%. This statistic tells us that a change in the value of authorship levels in an Android application relates to the number of faults in that Android application. However, we also observe that the MVA metric has a negative coefficient. This tells us that Android applications that have a high value of MVA (i.e., less shared code authorship) are less fault-prone than those with low values of code authorship.

Conclusion RQ2: In Section 2.1 we stated that we expect an increase in the value of the MVA metric to result in a decrease of the number of faults. From the discussion of the results presented above, we can conclude that Android applications with higher levels of code authorship are less fault-prone than applications with lower levels of code authorship. Again, our study's findings concur with the findings of Bird et al. [3] which were obtained on large commercial software projects. Regarding the work of Foucault et al. [7] which was conducted on seven large open source projects, our findings still disagree with their claim that the contribution of the MVA metric seem to be incidental. Our analysis shows that the metric MVA indeed has a relationship with faults in the applications.

From the results of RQ1 and RQ2 discussed above, we make the following recommendations to the developers of Android applications regarding the development process:

(1) *Changes made by minor contributors should be reviewed with more scrutiny before being committed on respective Github projects.* Development teams in Android applications should apply additional scrutiny to files authored by minor developers before these files are committed to the respective Android projects on Github. Since Github differentiates the author and the committer of a software change, we recommend that before a commit is performed, the committer should request major contributors on the project to perform an inspection on the changes made by minor contributors.
(2) *Android applications with lower levels of code authorship should be reviewed with more scrutiny.* Android application project teams on Github can make use of the MVA metric to scrutinize applications with low values of MVA as they may contain many faults and hence experience many failures.

5 THREATS TO VALIDITY

Though we have sought to make sure that all our data was gathered and linked correctly, and that our models are statistically robust, we note some potential threats to validity.

There are *construct validity* threats related to our data collection approach. In the description of the software project contributors in Section 3, we mentioned that some projects might have been started elsewhere, therefore GitHub does not provide the full history of these projects. We therefore decided to extract the code authorship metrics from GitHub in the project, by collecting only the contributions that were made *after the first release* of the project until *March, 6 2016*. This design decision is likely to affect the authorship metrics values obtained on some projects. However, since we analyzed a large number of projects (including projects started on Github), we believe the effect on our conclusions to be minimal. Another *construct validity* threat concern our use of heuristic to merge the names of contributors, in order to avoid polluting our data set with duplicate informations that would introduced noise. It is possible that some names that we merged represent different contributors. There is also a *construct validity* threat related to the fact that we considered faults on the entire projects and not at commit level. It is possible that one/few developer(s) may be writing most of the faulty code. Nevertheless, since we analyzed a large number of projects from diverse domains and diverse team sizes, we believe the effect of this skewness on our conclusions to be minimal.

There is an *internal validity* threat related to the tool (i.e., SonarQube) used to extract faults information, and compute size and complexity metrics. SonarQube is a well maintained tool that is extensively used by practitioners. However, as most static analysis tools, it doesn't have a 100% precision. It is possible that some of the faults considered in this study will never be experienced either by developers or the users of the applications.

Finally, although the observed correlations and the model results indicate that the phenomenons are related, i.e., low authorship and fault introduction, we cannot claim causation.

6 RELATED WORK

As mentioned earlier, the work presented in this paper builds on previous studies. To this end, throughout the paper we have discussed a number of studies that relate to ours. In this section, we shall discuss other studies related to our study that we have not yet discussed.

A number of other studies have investigated the relationship shared between code authorship metrics and fault-proneness. Matsumoto et al. [13] studied the effects of developer characteristics on software reliability. The authors proposed developer metrics such as the number of code churns made by each developer and the number of developers for each module. The authors analyzed the relationship between the number of faults and developer metrics. The authors reported that modules touched by more developers contained more faults.

Nagappan et al. [18] proposed a metric scheme to quantify organizational complexity, in relation to the product development process. They conducted a case study to identify if the metrics impact failure-proneness. For the organizational metrics, the number of developers was one of the metrics. The authors found that the precision and recall measures for identifying failure-prone binaries, using the organizational metrics, was significantly high.

Weyuker et al. [22] investigated the impact on predictive accuracy of using data about the number of developers who accessed individual code units. The authors found only moderate improvements of fault prediction models that included the cumulative number of developers as prediction factor. Meneely et al. [15] studied the effects of the number of contributors on security vulnerabilities focusing on the Linux software system. They reported that files with changes from nine or more developers were 16 times more likely to have a vulnerability than files changed by fewer than nine developers. In our study we also use the number of contributors as one authorship metric.

Mockus et al. [16] observed two code authorship patterns in the open source projects Apache and Mozilla. In the Apache project, they found that almost every source code file with more than 30 changes had several contributors who authored more than 10% of the changes. In the Mozilla project they found that code authorship decisions were enforced by project development guidelines, which stated that all contributions should be reviewed and approved by the module owner. The authors investigated authorship but did not attempted to examine the connection between the authorship patterns and fault-proneness.

7 CONCLUSIONS

In this study, we investigated whether applications with few major contributors are more reliable than applications larger number of contributors where developers do minor contributions. We carefully selected 278 Android applications from GitHub, from which we extracted metrics related to code authorship. We measured the reliability of the applications using information about fault occurrences, i.e., the number of faults. Using statistical modeling, we examined the relationship between code authorship metrics and faults. We observed that Android applications with higher levels of code authorship among contributors experience fewer faults.

We formulate the following two recommendations to development teams of Android applications projects:

(1) Changes made by minor contributors should be reviewed with more scrutiny before being committed.
(2) Android applications with lower levels of code authorship should be reviewed with more scrutiny as they may contain faults.

To generalize our findings, in the future we plan to expand this work to investigate other relatively small sized projects in other domains.

REFERENCES

[1] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley.

[2] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. 2009. Does Distributed Development Affect Software Quality?: An Empirical Case Study of Windows Vista. *Communication ACM* 52, 8 (2009), 85–93.

[3] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don'T Touch My Code!: Examining the Effects of Ownership on Software Quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.*

[4] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. 2009. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions Software Engineering* 35, 6 (Nov. 2009), 864–878.

[5] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. 2001. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering* 27, 7 (2001), 630–650.

[6] Barry E. Feldman. 2005. Relative Importance and Value. *Available at SSRN* (2005).

[7] Matthieu Foucault, Cédric Teyton, David Lo, Xavier Blanc, and Jean-Rémy Falleri. 2015. On the Usefulness of Ownership Metrics in Open-source Software Projects. *Inf. Softw. Tech.* 64, C (Aug. 2015), 102–112.

[8] Thomas Fritz, Gail C. Murphy, and Emily Hill. 2007. Does a Programmer's Activity Indicate Knowledge of Code?. In *Pro. of the 6th Joint Meeting of the European Soft. Eng. Conf. and ACM SIGSOFT Symposium on The Foundations of Software Engineering.*

[9] Michaela Greiler, Kim Herzig, and Jacek Czerwonka. 2015. Code Ownership and Software Quality: A Replication Study. In *Pro. of the 12th Working Conference on Mining Software Repositories.*

[10] Ulrike Grömping. 2006. Relative Importance for Linear Regression in R: The Package relaimpo. *Journal of Statistical Software* 17, 1 (2006), 1–27.

[11] F. E. Harrell Jr. 2015. *Regression Modeling Strategies.*

[12] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. 2015. Investigating Code Review Quality: Do People and Participation Matter?. In *Proceedings of International Conference on Software Maintenance and Evolution.*

[13] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. 2010. An Analysis of Developer Metrics for Fault Prediction. In *Proeedings of the 6th International Conference on Predictive Models in Software Engineering.*

[14] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.

[15] Andrew Meneely and Laurie Williams. 2009. Secure Open Source Collaboration: An Empirical Study of Linus' Law. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*.

[16] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.* 11, 3 (July 2002), 309–346.

[17] Christopher J Nachtsheim, John Neter, Michael H Kutner, and William Wasserman. 2004. Applied linear regression models. *McGraw-Hill Irwin* (2004).

[18] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. 2008. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Pro. of the 30th Int. Conf. on Software Engineering*.

[19] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can Developer-module Networks Predict Failures?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[20] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, Experience and Defects: A Fine-grained Study of Authorship. In *Pro. of the 33rd International Conference on Software Engineering*.

[21] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Pro. of the 38th International Conference on Software Engineering*.

[22] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. 2008. Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. *Empirical Software Engineering* 13 (2008), 539–559.

[23] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. 2012. Characterizing and Predicting Which Bugs Get Reopened. In *Proceedings of the 34th International Conference on Software Engineering*.