Code Authors Hidden in File Revision Histories: An Empirical Study

Siyi Gong
Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
gongsiyi@sjtu.edu.cn

Hao Zhong
Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
zhonghao@sjtu.edu.cn

***Abstract*—Although many programmers write their names in the comments of a source file, from such comments, it is unreliable to identify code authors, since the modifications of many programmers are not recorded. Even if they are recorded in a code repository, many authors are hidden in revision histories.**

The true authors of source files are important in many research topics. For example, when detecting plagiarism, if the authors of two source are overlapped, it becomes more challenging to determine plagiarism than the source files that are written by individual authors. As it is difficult to determine true authors of a source file, researchers typically use source files whose authors are already known (*e.g.*, the source files from Google Code Jam), but such files are not many and less representative. Meanwhile, although some empirical studies touch code authors, to the best of our knowledge, no prior study has analyzed the characteristics of code authors that are hidden in revision histories. As a result, many research questions along with code authors are still open. For example, how many authors does a source file can have, and what are the proportions of contributions per source file, if they are written by more than one author?

To answer the timely questions, in this paper, we conducted an empirical study on code authors that are hidden in revision histories. To support our study, we implemented a tool called CODA. By comparing the latest code lines with past commits, CODA identifies the true authors of all code lines. With its support, we analyzed 12,092 source files that were written by 506 programmers. Our study answers several interesting questions concerning code authors. For example, we find that 75.4% source files are written by multiple authors, and their contributions follow the famous 80/20 principle. These findings are useful to understand authors of source files in open source communities.

I. INTRODUCTION

Although the comments of a source file typically record its author(s) (*e.g.*, `@author` tags in Java comments), programmers seldom carefully maintain such comments, and author comments often refer to wrong authors. To identify the true authors, researchers [17], [36] have proposed various approaches. This research topic is known as code authorship attribution (see Section V for details). Besides this research topic, the true authors of a source file are important to various other research topics (*e.g.*, bug report assignments [8], software forensics [49], and plagiarism detection [44]).

Despite the importance of code authorship, the real authors of code lines are still largely unknown, and many fundamental questions along with code authors are still open. For example, a recent review [36] complains that most approaches assume that a source file is written by only an author, but source files in the wild are typically written by multiple programmers. As another example, most plagiarism detection tools compare the code similarity to determine plagiarism [11]. Although this setting fits some scenarios (*e.g.*, detecting plagiarism in the homework submitted by students), source files in the real development can be written by intersecting authors, and the definition of plagiarism shall be reconsidered. In the literature, researchers often build theories on their own experiences, instead of solid empirical evidences. For example, Kalgutkar *et al.* [36] list the challenges of identifying code authors. Although we agree that their visions are insightful, they did not provide any empirical evidences to support their listed challenges. To deepen the understanding on code authorship, there is a strong need for an empirical study, but it is challenging to conduct such a study. When programmers check out a project, they see the latest versions of source files, but the authors of source files are often hidden the revision history of source files.

To meet the timely need, we conducted the first empirical study on code authors. To assist our study, we implemented a tool called CODA. It extracts authors and their modifications from code repositories and matches modifications with the latest source files to determine the true author of each line. With CODA, we analyzed 12,092 source files that were written by 506 programmers. In this empirical study, we explore the following research questions:

- **RQ1.** *How many commits of an author are modified by follow-up commits?*
  Motivation. Some code authors are hidden in file revision histories, because the added code lines of their commits are modified by the commits of other authors. To explore the relevance of our target problem, from the perspective of commits, we count how many commits of an author are modified by other commits.
  Protocol. To answer this question, we collected the total commits of an author c_t, and the modified commits of the author c_m. After that, we used the ratio, $\frac{c_m}{c_t}$, to explore modified commits.
  Answer. We find that the commits of transient authors are often either totaly modified or never modified, and for most other authors, half of their commits are typically modified by follow-up authors (Finding 1).

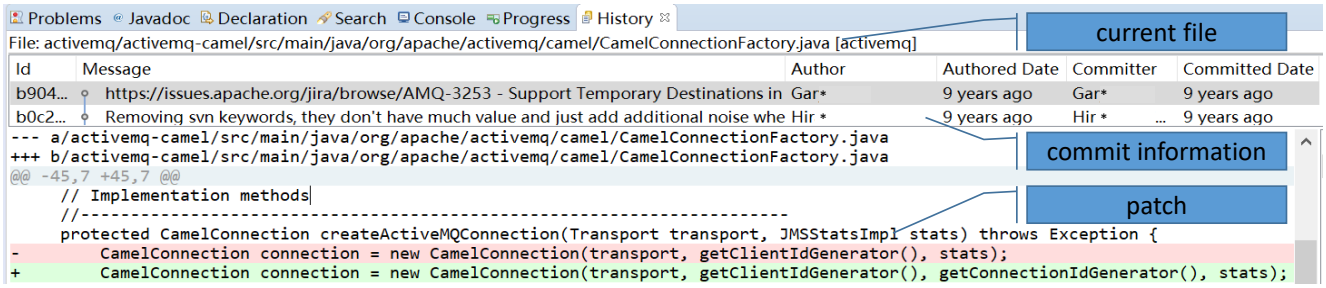


Fig. 1: The revision history of a source file (We anonymized all author names to protect privacy)

- **RQ2.** *How many authors does a source file have?*
  Motivation. The commits of a file build up the latest code lines and the true authors of a file. To explore the impacts of our target problem, from the perspective of the latest files, we count the number of authors for each file. In extreme cases, the modifications of an author can be totally rewritten by other authors. In this research question, we explore how frequently such cases happen.
  Protocol. To answer this question, for each source file, we identify the $real\ authors$ of its code lines by comparing their modified code with the latest code. The added lines of some programmers are totally rewritten by other programmers, and such programmers shall not be considered as real authors. We also identify these hidden authors of each source file.
  Answer. We find that 10% to 60% files have hidden authors (Finding 2), and 75.4% source files have multiple authors (Finding 3). Although so many source files have multiple authors, most files (the medians) are written by no more than five programmers (Finding 3).

- **RQ3.** *What are the proportions of contributed code lines, if a source file is written by more than one author?*
  Motivation. In RQ2, we find that about 75.4% source files are written by more than one author. When a file has more than one author, the contributed code lines of the authors can be quite different. In RQ3, we explore the distributions of those greatest and lest contributors.
  Protocol. To answer this question, for a source file, we calculate the total code lines of each code author l_{total}, and count its ratio over its all code lines l_c.
  Answer. We find that the authors of most files follow the famous 80/20 principle, *i.e.*, 80% of their latest lines are written by single authors. Although one author typically writes most code lines of a file, most files have several code lines that can be contributed by transient programmers or pair programming (Finding 4).

- **RQ4.** *How many modified lines are hidden?*
  Motivation. The development cost is also hidden in file revision histories. The hidden cost is useful to related research topics (*e.g.*, cost estimation [35]).
  Protocol. To answer this question, we count how many invisible lines a file has. Here, a code line is hidden, if it is modified by latter commits.
  Answer. We find that for most files, there are fewer than one hidden line behind a visible line, but in extreme cases, a line can be modified by ten previous lines (Finding 5). The result show that the hidden lines may not have major impacts on estimating overall cost, but their impacts on individual files can be significant.

- **RQ5:** *How do open source projects attract programmers?*
  Motivation: A health open source project shall attract programmers. In RQ5, we analyze the attractions.
  Protocol. To answer this question, we count lines modified by authors, and calculate the distribution over time.
  Answer: We find indicates that it is more challenge to attract long-term programmers (Finding 6). In open source project, most programmers make a commit, and modify about 30 lines of code, in each day (Finding 7).

II. An Example of Study Background

In this section, we use an example to introduce the story of code authorship, and analyzes its impacts. In read development, a file can have a long maintenance history, and many programmers can contribute its content. The maintenance history of a source file can be obtained by IDEs. For example, Figure 1 shows the history of a source file, which is retrieved by Eclipse. While multiple programmers can contribute to a source file, their contributions can twist with each other. For example, Figure 2 shows the modification details of this textual file. Specifically, Figure 2a shows the `createProducer` method of the latest file. Although it has only six lines, these lines are written by two programmers. Figure 2b shows the updated lines of a commit. This commit was submitted by Hir*, and it contributed to the first lines of the method `createProducer` in Figure 2a. Figure 2c shows the updated lines of two commits. The two commits were submitted by Rob*, and they contributed to the other three lines of the `createProducer` method.

As shown in this example, even a small method can have a complicated revision history. The complicated histories can influence various research topics. For example, researchers have proposed various approaches to identify the author(s) of a source file (see Section V for details), and Kalgutkar *et al.* [36] complain that most approaches assume that each file has exact an author. Such approach will fail to identify the true author of our example, since it is written by multiple authors. As another example, Bin-Habtoor and Zaher [11] introduce that some approaches detect copy-and-paste code snippets to detect

```
protected PerfProducer createProducer(...) {
  PerfProducer result = super.createProducer(fac,
      dest, number, payload);
  result.setDeliveryMode(DeliveryMode.
      NON_PERSISTENT);
  result.setSleep(10);
  return result;
}
```

(a) A commit history

```
author:Hir*
commit:230a86c
protected PerfProducer createProducer(...) {
  PerfProducer result = super.createProducer(fac,
      dest, number, payload);
```

(b) A commit of Hir*

```
author:Rob*
commit:6e7e3ab
  result.setDeliveryMode(DeliveryMode.
      NON_PERSISTENT);
  return result;
}
commit:63e3f41
 result.setSleep(10);
```

(c) Two commits of Rob*

Fig. 2: The authors hidden in revisions

plagiarism. In our example, the short method is composed by multiple authors, and each author contributes only one or two code lines. As a result, their mentioned approaches may not effectively detect plagiarism involving our example.

Although the above example illustrate the complexity of code authors, a single example is insufficient to show the relevance of this problem, and many questions are still open. For example, to what degree can a source file be written by multiple authors, and how are their contributed code lines twisted? To fully answer these questions, it is desirable to conduct an empirical study on code authorship. The answers to these questions have far reaching impacts on many research topics (*e.g.*, bug report assignments [8], software forensics [49], and plagiarism detection [44]).

It is challenging to conduct this study, since each single line of source code could be changed, added or deleted by multiple programmers and the contributions of a programmer can scatter across many lines of code. To handle this challenge, we implemented a tool called CODA to extract the accurate authors of each code line from its revision history. We next introduce our data set, CODA, and analysis protocol.

III. METHODOLOGY

In this section, we introduce CODA (Section III-A), our data set (Section III-B), and analysis protocol (Section III-C).

A. CODA

CODA first extracts commit authors and patches from code repositories, and then matches those patches with the latest source files to determine code authors.

1. Extracting commit details and patches. We built CODA on SVNKit [3], a popular library for developing clients of code repositories. For each commit, CODA extracts its commit detail and patch from code repositories. For example, as shown in Figure 1, we use Eclipse to retrieve the revision history of `CamelConnectionFactory`. For each commit of a source file, code repositories record its commit information such as its id, message, author, author date, committer, and committed date. For example, the first row of Figure 1 shows that the author and the committer of the latest commit is Gar*. The author and the committer of a revision is typically the same programmer, but we notice that in some cases, they are different. CODA considers that the author of revision is the true programmer of the revision. Besides its commit information, for each commit of a source file, code repositories store its changes as a patch. For example, in Figure 1, we highlight the patch of the latest commit. In this patch, the first and second rows show the names of the original file and the modified file, respectively. In this example, the commit does not change the file name. The third row shows the line numbers of modified code. In the follow-up lines, "+" denotes added lines; "-" denotes deleted lines; and other lines are unchanged. For each source file (f), we use CODA to extract all the commits C that modify the file. We next introduce how CODA determines the authors of f, according to C.

TABLE I: Subject projects.

Name	File	LOC	Author	Commit	Delta file
activemq	4,103	701,776	83	10,055	69,997
aries	2,113	288,164	35	4,846	26,517
carbondata	773	185,724	107	3,709	41,221
cassandra	1,650	546,857	214	21,995	90,959
derby	2,712	1,234,091	33	8,130	52,342
mahout	741	177,392	34	3,703	29,434
total	12,092	3,134,004	506	52,438	310,470

2. Matching patches to the latest source files. As shown in Figure 1, a patch encodes a modification as an addition (+) and a deletion (-). As only additions can appear in the latest files, CODA extracts only additions from patches, *i.e.*, the lines that start with "+". For each patch, CODA compares its added lines with the lines of the latest code to collect their common lines. Algorithm 1 shows the details of the comparison. Given the added lines of a commit (L_1) and the lines of the latest version (L_2), Line 2 iterates all the lines of L_1. If a line appears in both L_1 and L_2 (Line 6), the line is added to L'; Line 9 breaks the loop; and Line 2 checks the next line of L_1. The process terminates until L_1 reaches its ends. Given L_1 and L_2 as the inputs, if Algorithm 1 finds that L' is not empty, CODA determines that the author of L_1 is the author of $L' \subseteq L_2$. When a source file has multiple commits, CODA matches the commits in the order from the newest one to the oldest one. If a line of the latest version is already matched, CODA ignores the line in the later matches. As a result, it does not match superficial lines that accidentally appear in previous patches.

We use CODA to compare each latest source file with past commits. As shown in Figure 2, CODA is able to determine the author of each code line. Based on the identified authors of all code lines, we can answer the questions listed in Section I.

Algorithm 1: the `findCodeLineNumber` Algorithm

Input:
L_1 is the m code lines of a latest source file
L_2 is the n code lines that are rebuilt from a patch
Output:
L_2 is the code lines with line numbers

```
1: arr ← new int[m+1][n+1];
2: for i ← 1 to m + 1 do
3:     for j ← 1 to n + 1 do
4:         l1 ← L1.get(i);
5:         l2 ← L2.get(j);
6:         if l1 = l2 then
7:             arr[i][j] ← arr[i−1][j−1] + 1;
8:         else
9:             arr[i][j] ← max(arr[i−1][j], arr[i][j−1]);
10:        end if
11:    end for
12: end for
13: m ← m − 1;
14: n ← n − 1;
15: while m ≥ 1 and arr[m] = arr[m−1] do
16:    m ← m − 1;
17: end while
18: while m ≥ 0 and n ≥ 0 do
19:    l1 ← L1.get(i);
20:    l2 ← L2.get(j);
21:    if l1 = l2 then
22:        l2.line ← l1.line;
23:        m ← m − 1;
24:        n ← n − 1;
25:    else
26:        if arr[m][n+1] > arr[m+1][n] then
27:            m ← m − 1;
28:        else
29:            n ← n − 1;
30:        end if
31:    end if
32: end while
```

B. Dataset

Table I shows the data set. Column "**Name**" lists names of projects. All the projects are collected from the Apache foundation [1]. To ensure the reliability of our findings, in our study, we checked out thousands of commits and analyzed millions of modified files. As it is infeasible to manually analyze so many files, we implemented CODA to automate the process. Although CODA does not involve complicated analysis, it is built for analyzing the code repositories of Apache. As a result, in this study, we selected only Apache projects. However, to ensure their representativeness, we selected subjects from different types of projects such as messaging servers (activemq), OSGi application programming modes (aries), databases (cassandra and derby), and machine learning frameworks (mahout). Column "**File**" lists the number of Java source files in the latest versions. Column "**LOC**" lists their lines of code. Column "**Author**" lists the number of authors that appear in commits. Column "**Commit**" lists the number of commits. Column "**Delta file**" lists the number of delta files. Here, delta files are calculated as $deletions + additions + 2 \times modifications$, in that a modification can be considered as a deletion and an addition.

C. General Protocol

To explore the research questions in Section I, with the support of CODA, we analyze the data set in Section III-B. Our analysis protocol is as follows:

Step 1. Extracting commits for source files. All the projects in Table I maintain their source files on Github. From their Github code servers, we use CODA to extract all the commits of each project. For each commit, CODA stores its old and modified versions of source files in a local directory. For example, as shown in Table I, CODA extracted 10,055 commits for the `activemq` project.

Step 2. Identifying the revisions of each code line. Given the commits of a source file as its input, we use CODA to identify the author of each code line. CODA determines the author of a code line by comparing the line with the updated code lines of all commits. For example, as shown in Figure 2, Figure 2a shows the latest file, Figure 2b and Figure 2c show the updated lines of two commits. CODA compared the latest file with the updated lines to determine the authors of code lines. During the process, we collected the results to explore RQs 1, 2, and 4.

Step 3. Identifying the contribution of each author. After the author of each code line is identified, we used CODA to rebuild all the updated lines of a programmer. For example, Figure 2c lists the updated lines that were written by Rob* and still appear in the latest source files. Based on the contributions of authors, we explored RQs 3 and 5.

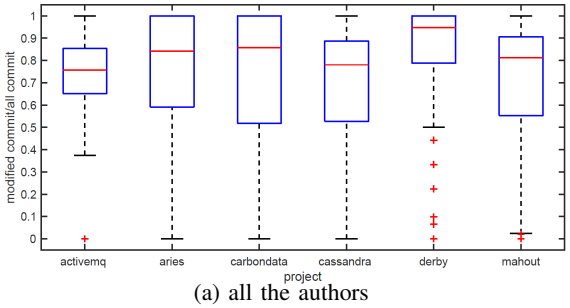
IV. EMPIRICAL RESULT

This section presents our analysis results. More details of our dataset, and our results are listed on our project website: https://github.com/gongsiyi/codeauthor

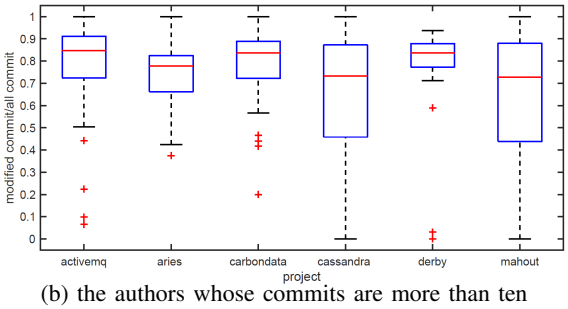
A. RQ1. Modified Commits

1) Protocol: For each commit on a file, we compare its added lines with the latest file. If one added line does not appear in the latest file, we consider that the commit is modified by latter commits. For each project, we use c_t to denote the total commits of an author, and c_m to denote the modified commits of the author. For each author, we use the ratio, $\frac{c_m}{c_t}$, to denote how many commits of an author are modified by latter commits. We draw box plots to present the distribution of modified commits per author.

2) Result: Figure 3a shows the distribution of modifications. The result shows that the medians are around 80%. This result indicates that for most authors, 80% of their commits are modified by latter commits. Figure 3a also shows some extreme cases. For example, in aries, the commits of some authors (*e.g.*, Vio*) are all modified, but the commits of some other authors (*e.g.*, Ala*) are never modified by latter commits. After some inspection on extreme cases, we find that most of such commits are written by transient programmers. In the above example, in total, Ala* made one commit, and Vio* made four commits. In a project, transient programmers are often assigned with less challenging or less important programming tasks. For those tasks, programmers may not produce bugs, and even if they do, their introduced bugs may not cause notable consequences. For example, in total, Krz* made two commits to aries, and the two commits modified only three lines of code in two test cases. As their



(a) all the authors



(b) the authors whose commits are more than ten

Fig. 3: The ratios of modified commits

modifications are minor, the two commits are never modified by latter commits. However, if transient programmers modified more lines of code, their commits are likely to be modified by latter commits. For example, in aries, Pat* also made two commits on two test cases. In particular, in one commit, Pat* implemented a static class, and in another commit, he implemented its most lines of code. As the two commits modified many lines of code, and their modified lines are further modified by latter commits.

We next analyze the commits of those core programmers. In particular, we analyze programmers who made more than ten commits, in that transient programmers made fewer commits than others. Figure 3b shows the distribution of the remaining authors. We find that the lengths of all the boxes in Figure 3b become shorter than those of Figure 3a. This result indicates that more core authors have modified commits than transient authors, but the commits of fewer core authors are totally modified than those of transient authors. Our observations lead to our first finding:

Finding 1. The commits of transient authors are often either totally modified or never modified, but the commits of core authors are more stable. For most core authors, 80% of their commits are modified by latter commits.

As many commits are modified, it is likely that the commits of a programmer are all modified by others. We next investigate its impacts on the identification of code authors.

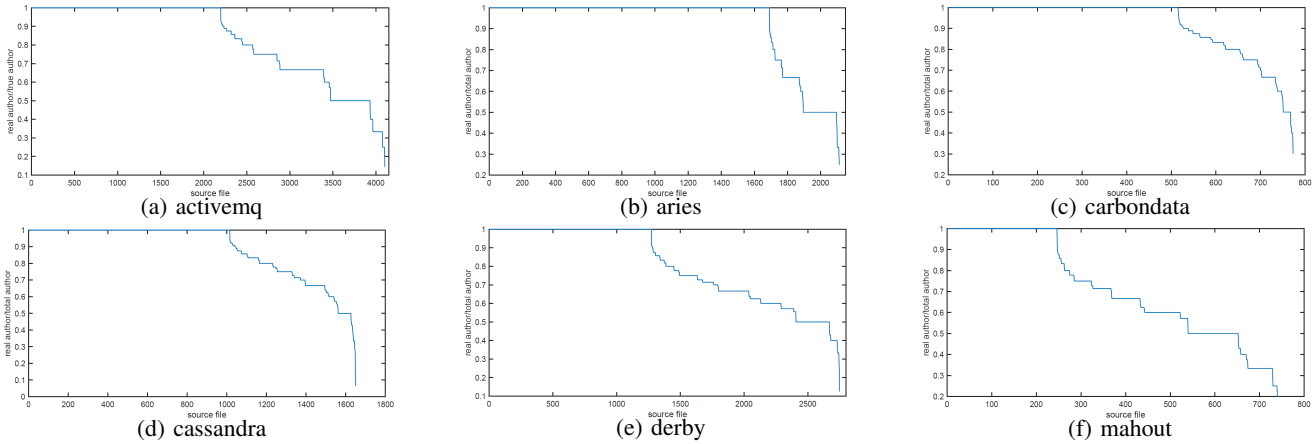
B. RQ2. Multiple Authors

1) Protocol: If a programmer made a commit on a file, the prior approaches [40] determine that the programmer is an author of the file. As a comparison, we use their strategy to determine the authors of a file. We refer to such authors as the total authors of a file. However, even if a programmer appears in the total-author list of a file, it can be unreasonable to determine that the programmer is an author of the file, because all the commits of the programmer can be totally rewritten by other authors. If a programmer appears in a commit of a file, Meng *et al.* [40] determine that the program is an author of the file. Although the strategy is more accurate that the authors under the `@author` tags, in Section IV-A, Finding 1 shows that for most programmers, half of their commits are modified by latter ones. We used CODA to compare the latest files with the modifications of past commits.

As shown in Figure 2, CODA is able to determine the true author of each code line. For each source file, based on the identified author of each code line, we calculate the number of real authors. To understand the impacts of modified commits, we use CODA to identify real authors of all files. Here, if at least one line of her modifications appear in the latest version of a file, we consider the programmer as a real author of the file. When counting real authors, we order the commits of a file in the chronological order. If a code line already appears in a commit, we do not compare it with other commits. Some code lines can be quite common (*e.g.*, braces). Our strategy does not identify superficial authors who made only such common modifications as real authors. Figure 5b shows the distributions of real authors. Comparing with Figure 5a, we find that except aries, the medians of all the projects decrease. This result indicates that modified commits influence the identification of code authors, and all the projects are influenced. To further understand the influence, for each file, we calculate its ratio of $\frac{real\ author}{total\ author}$. Here, the total authors of a file are the authors that appear in the commits of the file. We draw plots to show the distributions.

2) Result: We find that in some cases, the commits of a programmer are totally rewritten by other programmers. For example, in activemq, the class, `JMSMappingOutboundTransformer`, was created by Timothy Bish. After it was created, another programmer, Robert Gemmell, fixed two bugs of this file (`AMQ-5637` and `AMQ-5592`). The fixed file is still buggy. When Timothy Bish fixed latter bugs (*e.g.*, `AMQ-5648`), he revised the modifications of Robert Gemmell. As a result, none of Robert Gemmell'modifications appears in the latest version of this file, and he is in fact not an author of this file. However, by the criteria of the prior studies (*e.g.*, [40]), Robert Gemmell is still a valid author of this file.

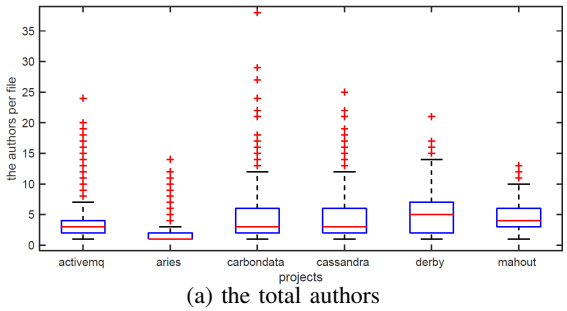
Figure 4 shows the distributions of such cases. This result shows that the distributions vary among different projects. As one extreme case, in aries, about 10% files have authors whose modifications are totally rewritten by other authors, but as another extreme case, the percent is about 60% for mahout. When a ratio of a file is less than 100%, at least one programmer is not a real author. A lower ratio indicates that more authors are not real. Based on where the plots turn down, for activemq, cassandra, derby, and mahout, about half of their files have authors whose modifications are totally rewritten. The plots of all the projects end at a low ratio of about 0.2. This result indicates that for all the projects, there are some extreme cases. In such cases, programmers rewrite many lines of files, and the modifications of about 80% authors are totally



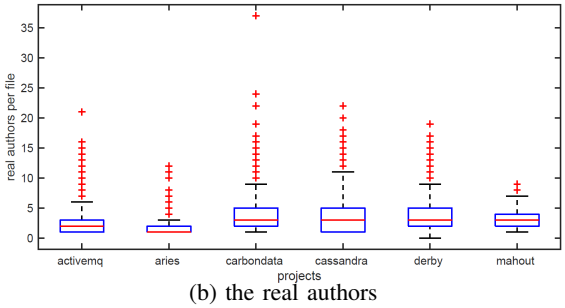
(a) activemq (b) aries (c) carbondata

(d) cassandra (e) derby (f) mahout

Fig. 4: The ratios of hidden authors



(a) the total authors



(b) the real authors

Fig. 5: The number of authors

rewritten. The observations lead to another finding:

Finding 2. Although some programmers appear in the commits of a file, they shall not be considered as a true author of the file, because their modifications do not appear in the latest files. The percents of such authors varies from 10% (aries) to 60% (mahout).

We calculate the percents of files that are written by multiple authors. The results are activemq (75.6%), aries (46.3%), carbondata (83.7%), cassandra (77.3%), and derby (88.3%). In total, 75.4% source files are written by more than one programmer. Figure 5a shows the distributions of authors per file. In some extreme cases, a file can have more than 30 authors. However, most maximum (the largest data point excluding any outliers) are below ten, and the medians are around five. The result leads to a finding:

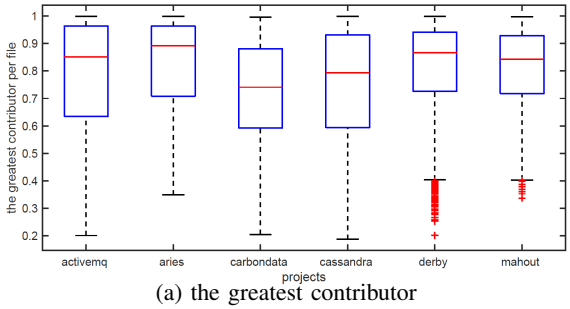
Finding 3. In total, 75.4% source files are written by multiple programmers, and most files have five authors.

In summary, we find that 75.4% files are written by a team of authors, and most teams have five programmers. The modifications of a programmer can be totally rewritten by other programmers, which influences from 10% files of aries to 60% files of mahout.

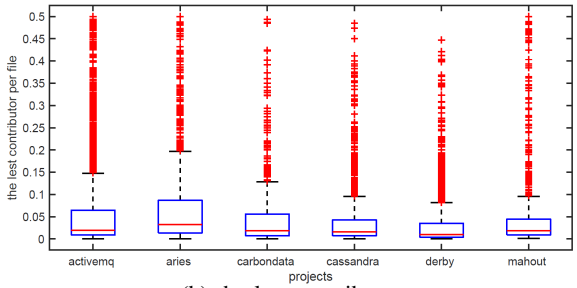
C. RQ3. Author Contributions

1) Protocol: In Section IV-B, we find that a notable portion (75.4%) of files are written by multiple authors, and the modifications of an author can be totally rewritten by other authors. The real authors of a file can have different contributions to the file. In this section, we use the numbers of code lines to measure the contributions of real authors. A commit on a file can delete lines (l_d) and add lines (l_a). For each author, we compare all the added lines (l_a) of her commits with the the latest version of a file, and count the number of the common lines as l_c. As we did in Section IV-B, when counting l_c of real authors, we rank the commits of a file in the chronological order. If a code line already appears in a commit, we do not compare it with previous commits. If a file has l_{total} lines of code and an author has l_c common lines, we calculate the contribution of the author as $\frac{l_c}{l_{total}}$. For each file with multiple authors, we calculate the contributions of all its authors. Among the authors, we call the one who write the most lines of code as the greatest contributor, and the one who write the fewest lines of code as the lest contributor. We draw box plots to show the distributions of contributions for the greatest contributors and least contributors.

2) Result: Figures 6a and 6b show distributions of the greatest contributors and the lest contributors, respectively. In Figure 6a, the medians are around 0.8. This result indicates that for most files, a single author (*i.e.*, the greatest contributor) writes 80% lines of code. This contribution shows that the contributors of authors follow the famous 80/20 principle [27]. In Figure 6b, the medians are fewer than 0.05. This result shows that most files are touched by a programmer, and



(a) the greatest contributor



(b) the lest contributor

Fig. 6: The contributors of files

their added lines are fewer than 5% of their total lines of code. Zhou and Mockus [61] report that open source projects have many transient programmers. Such programmers can be detected as the least contributors, because their added lines are few. Meanwhile, we notice that all the projects of Figure 6b have many outliers. In the modern software development, pair programming has been widely used [55], and researchers [15] are advocating this practice in distributed environments (*e.g.*, open source projects). The outliers may be a side evidence for the practice of pair programming in open source communities, because in such a programming paradigm, two programmers typically make equivalent contributions to a file. Our observations lead to the following finding:

Finding 4. The contributions of authors follow the famous 80/20 principle, and most files have minor modifications, which can be contributed by transient programmers or the practice of pair programming.

D. RQ4. Invisible Added Code Lines

1) Protocol: Although each commit can add some lines of code to a file, an added code can be modified by latter commits. As a result, the latest version of a file has fewer lines of code than those added lines of past commits. Although such lines are invisible, they reflect the true effort of implementing and maintaining a file.

For each file, we calculate the sum of added lines in all its past commits, and denote the sum as l_s. If the latest version of the file has l_{total} lines of code, we calculate the ratio, $\frac{l_s}{l_{total}}$, to measure how many invisible added code lines a file has.

2) Result: Figure 7 shows the distribution of the ratios. The medians of all the projects are between one and two. This result indicates that for most files, their total lines of code do not change much. However, as shown in Figure 7, all the

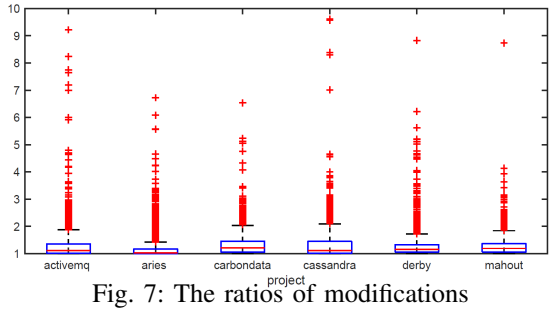


Fig. 7: The ratios of modifications

projects have many outliers. In some extreme cases, a file can have ten times more added lines than what is visible in the latest versions. For each project, we ordered their files in the descending order of the ratio, and manually inspected the top five files. We identified four causes:

1. Functions are moved (40.0%). For example, in activemq, the `AmqpProtocolConverter` interface was initially a concrete class. After its implementation code was moved to its subclasses, its current version has much fewer lines, and the ratio becomes high.

2. Functions are rewritten (20.0%). For example, in derby, most commits of the `TriggerOldTransitionRows` class added only minor lines, but to support `BLOB` and `CLOB` types (`DERBY-438`), the class is totally rewritten, which leads to a high ratio.

3. Test code is changed (10.0%). In total, three test cases are intensively modified, and the changes can be caused by requirement changes.

4. Constants are changed (6.7%). For example, in derby, the `MessageId` interface stores error ids, and the ids were constantly changed over time.

We find no patterns for the remaining cases. Most cases have a long maintenance history, and their added lines were accumulated over time. Our observations lead to a finding:

Finding 5. For most files, the added lines over total visible lines are between one and two. However, there are some extreme cases, in which the ratios are more than ten.

E. RQ5: Attraction to Programmers

1) Protocol: We use three metrics such as programming days, commits per day, and modified lines per day to measure the attraction of open source projects. For each programmer, we collect the dates (d_1 and d_2) of the first commit and the last commit. We calculate the programming day (day) of a programmer as $d_2 - d_1$. For each programmer, we collect all her commits ($commit$), and calculate her commits per day as $\frac{commit}{day}$. We collect all the commits of a programmer, and calculate the modified lines of a commit as the sum of its added lines and deleted lines. Here, a modification is counted as a deletion and an addition. For each programmer, we count her total modified lines (tl), and calculate her modified lines per day as $\frac{tl}{day}$. We draw box plots to show the distributions of authors, as far as the three metrics are concerned.

2) Result: Figure 8a shows the distributions of programming days. The medians of aries, derby, and mahout are more

than those of activemq, carbondata, and cassandra. For aries, derby, and mahout, most authors have about 400 programming days, but for the other three projects, most authors have fewer than 100 programming days. Table I shows that activemq, carbondata, and cassandra have more programmers than aries, derby, and mahout. We count the programmers whose programming days are more than 100. The results are activemq (36), aries (27), carbondata (47), cassandra (83), derby (25), and mahout (22). In total, 47.4% programmers have more than 100 programming days, but the percents are lower for those crowded projects, *i.e.*, activemq (43.4%), carbondata (43.9%), and cassandra (38.8%). As a comparison, the percents of aries, derby, and mahout are 77.1%, 75.8%, and 64.7%, respectively. Although the long-term programmers of activemq, carbondata, and cassandra are still more than those of aries, derby, and mahout, the differences are less significant. The observation leads to another finding:

Finding 6. Although some projects attract more programmers than others, it is more difficult to attract long-term programmers. The three projects such as activemq, carbondata, and cassandra attract more programmers than the other three projects, but in activemq, carbondata, and cassandra, about 30% more programmers are not long-term *i.e.*, having fewer than 100 programming days.

Figure 8b shows the distributions of commits per day, and Figure 8c shows the distributions of modified lines per day. The medians are consistent, and lead to a finding:

Finding 7. Most programmers make a commit, and modify about 30 lines each programming day.

In summary, it is more difficult to attract long-term programmers, and most programmers make limited changes each programming day. Our result reflects the state of the practice for open source development, but it is not an indicator of the inefficiencies of open source development. We further discuss this issue in Section VI.

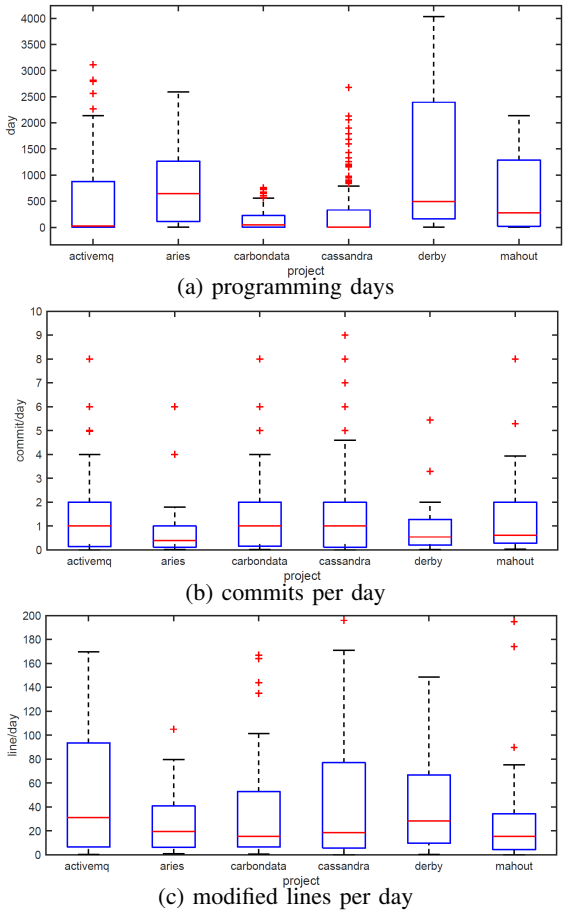
F. Threat to Validity

The threats to internal validity include that a programmer can wrongly fill in the two columns. As most Apache projects are carefully maintained, such errors shall be rare. The threat could be reduced by manually inspecting our identified authors. The threats also include that a programmer can use different names when committing changes. This threat could be reduced by the data sanitization steps [13].

The threats to external validity include our subjects. Although we analyzed thousands of files of six popular projects, they are limited and all in Java and all the projects belong to Apache. The threat could be reduced with more subjects [43]. However, our major findings may not change much, since we have selected different types of projects.

V. Code Authorship Attribution

To better interpret our findings, in this section, we introduce the research on code authorship attribution. Identifying the authors for a piece of code has been a long research topic,



(a) programming days

(b) commits per day

(c) modified lines per day

Fig. 8: The attractions to authors

and this research topic is often called as code authorship attribution. Table II shows some representative approaches. We include Halstead [29], since it is first one to identify authors of source files as reported by Kalgutkar *et al.* [36]. To show the state of the art, our other selected approaches are published after 2010.

In Table II, Column "Input" lists the input data of code authorship attribution. Most approaches (*e.g.*, [16]) analyze source files through static analysis. Besides static analysis, a few approaches (*e.g.*, [32]) dynamically execute compiled source files to analyze their details. As an interesting research topic, researchers (*e.g.*, [45]) have proposed approaches to identify authors of binaries. For example, Rosenblum *et al.* [45] use the Paradyn tools [4] to analyze compiled C/C++ code.

Column "Feature" lists extracted features. The prior approaches mainly extract two types of features such as metrics and ngrams. A software metric is a measurable attribute of a program. For example, two extracted features of Yang *et al.* [56] are as follows:

1) The ratio of blank lines to code lines
2) The ratio of comment lines to code lines

In natural language processing, ngram models are widely used to identify the author of a text [19]. As Hindle *et al.* [33]

TABLE II: The typical approaches of code authorship attribution.

Author	Year	Input	Feature	Technique	Dataset
Halstead [29]	1972	source code	metric	statistic	5 small programs
Hayes and Offutt [32]	2010	source code	metric	statistic	15 programs by professionals and 60 by students
Rosenblum *et al.* [45]	2011	binary	n-gram	svm and k-means	2,579 GCJ programs and 203 from a course
Bandara and Wijayarathna [10]	2013	source code	metric	logistic regression	5 programs from SourceForge
Chouchane *et al.* [18]	2013	binary	n-gram	k-nn classifier	7 morphing engines
Caliskan *et al.* [17]	2015	source code	metric	neural network	GCJ programs by 1,850 authors
Alsulami *et al*. [7]	2017	source code	AST	neural network	700 GCJ files and 200 GitHub files
Yang *et al.* [56]	2017	source code	metric	neural network	3,022 GitHub source files
Caliskan *et al.* [16]	2018	source code	n-gram	random forest	GCJ programs by 100 authors
Abuhamad *et al.* [5]	2020	source code	word2vec and TF-IDF	RNN with random forest	26,607 GitHub source files

show that programming languages and natural languages share similarities, some approaches (*e.g.*, [16]) use ngram models to capture the habits of programmers. For example, Caliskan *et al.* [16] extract a bigram, which indicates that `func` is followed by `if`. Some approaches (*e.g.*, [7]) bypass the extraction of metrics or ngrams, and take ASTs or CFGs directly as their inputs. These approaches typically rely on existing techniques to implicitly extract metrics or ngrams. For example, Alsulami *et al* [7] use a recurrent neural network [23] to analyze ASTs, and such a technique extracts code features as ngrams.

Column "Technique" lists the techniques that identify authors based on extracted features. The prior approaches are roughly divided into two research lines. The first line of approaches creates hypotheses of distinguishing authors, and use statics testing to validate whether their hypotheses hold. For example, Hayes and Offutt [32] report that three metrics are significant discriminators for identifying programmers: (1) the average occurrence of operands, (2) the average occurrence of operators, and (3) the average occurrence of constructs. The other line of approaches reduces the problem of identifying authors to clustering or classification problems, and use data mining or machine learning techniques [58] to resolve the problems. For example, Rosenblum *et al.* [45] use svm and k-means, and Caliskan *et al.* [17] use neural networks.

In early years, researchers often evaluate their approaches on small programs. For example, Halstead [29] list all the evaluated five programs. Tennyson and Mitropoulos [51] select code samples from textbooks because they can determine the authors of books as the authors of code samples. As shown in Table II, some recent approaches (*e.g.*, [45]) use GCJ programs [2]. GCJ is the abbreviation of Google Code Jam. It is a code computation that is hosted by Google. Although the author of a GCJ program is recorded, as criticized by Kalgutkar *et al.* [36], in the GCJ benchmark, each author typically has quite limited code samples, and its languages are limited to several popular ones (*e.g.*, C++). Researchers recently use source files of open source projects as their benchmarks. For example, as shown in Table II, Yang *et al.* [56] collected 3,022 files from Github.

VI. Interpretation of Our Findings

In this section, we interpret our findings:

1. The challenges of identifying authors. Findings 2 and 3 show that most source files are written by multiple code authors. Based on our results, we briefly discuss some potential opportunities and challenges of identifying multiple code authors. First, as shown in Figure 2, even a short method can have multiple authors, and the modifications of a programmer can be discontinuous. Second, Finding 4 shows that the contributions of authors are quite different, and some programmers are transient, they only write a small portion of lines. The above findings show that researchers encounter the data imbalance problem when identifying real authors. Third, as shown in Figure 2, two lines are added by the same programmer, their commits are different and the time interval between the commits can be quite long. Kalgutkar *et al.* [36] point out that the code style of a programmer can evolve over time. Even if two lines are written by the same author, their styles can be different. Fourth, Finding 3 shows that even if a programmer is the author of a file's commits, she may not be an author of the file, since all her added code are modified by other programmers. The above findings show that researchers encounter the uneven data distribution problem when identifying real authors. We have released our dataset on our website. With more advanced statistical analysis techniques (*e.g.*, Bayesian analysis [24]), researchers can derive more beneficial findings.

2. The invisible authors of code lines. Finding 5 shows that for a file, some lines of invisible, because they only appeared in past versions. Typically, invisible lines are as many as or even more than visible lines. For source files, especially for those extreme cases, the relation between the visible code and the invisible code can be described as the tip over an iceberg. Although programmers work on the tip, the stories under the water are often more revealing and interesting. For example, Kim *et al.* [37] analyze how clones evolve over time. Compared with clones in the latest source files (the tip), their study reveals the evolution of clones (the iceberg), which present more interesting details (*e.g.*, why a clone is created, and how a clone changes from one type to another).

Besides clones, the phenomenon of the tip over an iceberg can influence many other research topics. For example, researchers have proposed various metrics to measure the complexity of code (*e.g.*, the cyclomatic complexity density [25]). Compared with metrics on latest files, researchers (*e.g.*, [30]) have proposed metrics that are calculated from code changes, which can be more reasonable to measure the complexity of code. As another example, it has been a hot research topic to estimate the development cost of a software [14]. Typically, these approaches use lines of code to measure the development cost. Figure 7 shows that there are many invisible lines of code behind those visible ones, and ratio can be high. Instead of those visible lines, those invisible lines can be a better indicator of the development cost. After CODA identifies those hidden lines, other researchers can evaluate and modify their cost prediction approaches accordingly.

3. The comparison with commercial projects. Open source and commercial software are two major paradigms for developing software. Researchers [41] have conducted empirical studies to compare the efficiency of the two paradigms. Figure 8 shows the activities of programmers in open source development, and some numbers look low. However, our results are not indicators for the efficiency of either paradigm. Indeed, in the development of commercial software, some roles (*e.g.*, testers) may not write many lines of code either. In addition, in different stages of development, programmers typically make different numbers of commits. Our results show the averages, but we do not consider their development stages. Finally, as most programmers are volunteers in open source development, they may work on weekends. As a result, when comparing with commercial development, the activities of open source development shall be adjusted.

VII. Related Work

Besides the code authorship attribution, our work is related to the following research topics:

The analysis on code ownership. As defined by Hattori *et al.* [31], the ownership of a programmer on a file quantifies the amount of knowledge the programmer has on this file. Greiler *et al.* [26] mine the relation between code ownership and software quality. Diaz *et al.* [21] use code ownership to assist the recovery of links between source files and high-level designs (*e.g.*, use cases). Other researchers [12], [52] analyze the relationship between code ownership and software quality. The concepts of code owners and code authors are related. We notice that some approaches use similar techniques to determine the owner of a file. For example, Corley *et al.* [20] extract all the commits on a file and add all their authors to the owners of the file. Their strategy is identical to Meng *et al.* [40]. As a result, most criticisms on identifying authors still hold on the identification of code owners. In our study, we propose CODA to extract accurate authors of source files, whose results can be also useful to identify code owners.

The empirical studies on commits. Researchers have conducted various empirical studies on commits. Guzman *et al.* [28] analyze the emotion changes of commit messages. Alali *et al.* [6] analyze what is a typical commit (*e.g.*, how many files are added in a commit). Tufano *et al.* [53] analyze the factors of a commit that can introduce bugs. Tufano *et al.* [54] show that many commits are no longer compilable, when they are checked out. Zhong and Su [60] analyze to what degree bug-fixing commits overlap with previous ones. Zhong and Meng [59] analyze the repetitiveness of bug-fixing commits. Eyolfson *et al.* [22] analyze the correlation between bugs and commits. Song *et al.* [48] analyze the commits that are related to workarounds. Our study analyzes commits from another perspective of commits, their authors, which are not explored by the prior studies.

Identifying the expertise of developers. As advocated by Mei and Zhang [39], researchers have learnt models from large scale software engineering data to identify experts. Matter *et al.* [38] build a term-to-developer matrix with word frequencies. Moin *et al.* [42] use an n-gram algorithm on the granularity of characters. Tamrawi *et al.* [50] leverage the fuzzy set theory to reduce the negative effect brought by less discriminative words. Shokripour *et al.* [47] report that more recent activities have higher impacts on the expertise of a developer. Based on this observation, they design a time-based approach to determine experts. Zhang *et al.* [57] construct a heterogeneous network from bug reports, and search the network for experts. Baltes and Diehl [9] survey the criteria of experts, and test to what degree such criteria hold. Researchers also identify experts from source files and commits. Hossen *et al.* [34] report that if a programmer appears in the code comments of a file, the programmer is often an expert to fix its bug. Servant *et al.* [46] identify experts by their changes in commits. The findings of our empirical study can be useful to determine the expertise of developers.

VIII. Conclusion and Future Work

The authors of source files are important in many applications, but they are poorly documented in source code. Even if they are recorded by code repositories, the modifications of code authors are often hidden in file revision histories. As a result, many research questions along with code authors are still open. To deepen the knowledge on code authors, we conduct an empirical study and explore open questions concerning code authors. We summarize our results into seven findings, and interpret them from three perspectives.

In future work, we plan to conduct in-depth investigations on more challenges of identifying authors and their author stories behind source files. In addition, due to the availability of data sources, we analyzed only open source projects in this study, but we plan to investigate the difference between open source and commercial software projects, and we can thus draw more general conclusions.

Acknowledgement

We appreciate the anonymous reviewers for their insightful comments. Hao Zhong is the corresponding author. This work is sponsored by the National Key R&D Program of China No. 2018YFC083050.

REFERENCES

[1] The Apache foundation. http://www.apache.org/, 2018.

[2] Google Code Jam . https://codingcompetitions.withgoogle.com/codejam, 2019.

[3] SVNKit. https://svnkit.com, 2019.

[4] Paradyn. http://www.paradyn.org/html/overview.html, 2020.

[5] M. Abuhamad, T. Abuhmed, D. Nyang, and D. Mohaisen. Multi-χ : Identifying multiple authors from source code files. In *The 20th Privacy Enhancing Technologies Symposium (PETS)*, 2020.

[6] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? a characterization of open source software repositories. In *Proc. ICPC*, pages 182–191, 2008.

[7] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, and R. Greenstadt. Source code authorship attribution using long short-term memory based networks. In *Proc. ESORICS*, pages 65–82, 2017.

[8] J. Anvik, L. Hiew, and G. Murphy. Who should fix this bug? In *Proc. 28th ICSE*, pages 361–370, 2006.

[9] S. Baltes and S. Diehl. Towards a theory of software development expertise. In *Proc. ESEC/FSE*, pages 187–200, 2018.

[10] U. Bandara and G. Wijayarathna. Source code author identification with unsupervised feature learning. *Pattern Recognition Letters*, 34(3):330–334, 2013.

[11] A. Bin-Habtoor and M. Zaher. A survey on plagiarism detection systems. *International Journal of Computer Theory and Engineering*, 4(2):185, 2012.

[12] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proc. ESEC/FSE*, pages 4–14, 2011.

[13] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Proc. MSR*, pages 1–10, 2009.

[14] B. Boehm, C. Abts, and S. Chulani. Software development cost estimation approaches—a survey. *Annals of software engineering*, 10(1-4):177–205, 2000.

[15] K. Braithwaite and T. Joyce. Xp expanded: distributed extreme programming. In *Proc. XP*, pages 180–188, 2005.

[16] A. Caliskan, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *NDSS*, 2018.

[17] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt. De-anonymizing programmers via code stylometry. In *Proc. USENIX Security*, pages 255–270, 2015.

[18] R. Chouchane, N. Stakhanova, A. Walenstein, and A. Lakhotia. Detecting machine-morphed malware variants via engine attribution. *Journal of Computer Virology and Hacking Techniques*, 9(3):137–157, 2013.

[19] R. Clement and D. Sharp. Ngram and bayesian classification of documents for topic and authorship. *Literary and linguistic computing*, 18(4):423–447, 2003.

[20] C. S. Corley, E. A. Kammer, and N. A. Kraft. Modeling the ownership of source code topics. In *Proc. ICPC*, pages 173–182, 2012.

[21] D. Diaz, G. Bavota, A. Marcus, R. Oliveto, S. Takahashi, and A. De Lucia. Using code ownership to improve ir-based traceability link recovery. In *Proc. ICPC*, pages 123–132, 2013.

[22] J. Eyolfson, L. Tan, and P. Lam. Correlations between bugginess and time-based commit characteristics. *Empirical Software Engineering*, 19(4):1009–1039, 2014.

[23] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural Computation*, 12(10):2451–2471, 2000.

[24] J. K. Ghosh, M. Delampady, and T. Samanta. *An introduction to Bayesian analysis: theory and methods*. Springer Science & Business Media, 2007.

[25] G. K. Gill and C. F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering*, 17(12):1284, 1991.

[26] M. Greiler, K. Herzig, and J. Czerwonka. Code ownership and software quality: a replication study. In *Proc. MSR*, pages 2–12, 2015.

[27] A. Grosfeld-Nir, B. Ronen, and N. Kozlovsky. The pareto managerial principle: when does it apply? *International Journal of Production Research*, 45(10):2317–2325, 2007.

[28] E. Guzman, D. Azócar, and Y. Li. Sentiment analysis of commit comments in github: an empirical study. In *Proc. MSR*, pages 352–355, 2014.

[29] M. H. Halstead. Natural laws controlling algorithm structure? *ACM Sigplan Notices*, 7(2):19–26, 1972.

[30] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. ICSE*, pages 78–88, 2009.

[31] L. P. Hattori, M. Lanza, and R. Robbes. Refining code ownership with synchronous changes. *Empirical Software Engineering*, 17(4-5):467–499, 2012.

[32] J. H. Hayes and J. Offutt. Recognizing authors: an examination of the consistent programmer hypothesis. *Software Testing, Verification and Reliability*, 20(4):329–356, 2010.

[33] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proc. 34th ICSE*, pages 837–847, 2012.

[34] M. K. Hossen, H. Kagdi, and D. Poshyvanyk. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 130–141, 2014.

[35] M. Jorgensen and M. Shepperd. A systematic review of software development cost estimation studies. *IEEE Transactions on software engineering*, 33(1):33–53, 2006.

[36] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhanova, and A. Matyukhina. Code authorship attribution: Methods and challenges. *ACM Computing Surveys*, 52(1):3, 2019.

[37] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. ESEC/FSE*, pages 187–196, 2005.

[38] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proc. MSR*, pages 131–140, 2009.

[39] H. Mei and L. Zhang. Can big data bring a breakthrough for software automation? *Science China Information Sciences*, 61(5):1–3, 2018.

[40] X. Meng, B. P. Miller, and K.-S. Jun. Identifying multiple authors in a binary program. In *Proc. ESORICS*, pages 286–304, 2017.

[41] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the apache server. In *Proc. ICSE*, pages 263–272, 2000.

[42] A. Moin and G. Neumann. Assisting bug triage in large open source projects using approximate string matching. In *Proc. ICSEA*, 2012.

[43] M. Nagappan, T. Zimmermann, and C. Bird. Diversity in software engineering research. In *Proc. ESEC/FSE*, pages 466–476, 2013.

[44] A. Parker and J. O. Hamblen. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, 32(2):94–99, 1989.

[45] N. Rosenblum, X. Zhu, and B. P. Miller. Who wrote this code? identifying the authors of program binaries. In *Proc. ESORICS*, pages 172–189, 2011.

[46] F. Servant and J. A. Jones. Whosefault: automatic developer-to-fault assignment through fault localization. In *Proc. ICSE*, pages 36–46, 2012.

[47] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. A time-based approach to automatic bug report assignment. *The Journal of Systems & Software*, 102:109–122, 2015.

[48] D. Song, H. Zhong, and L. Jia. The symptom, cause and repair of workaround. In *Proc. ASE*, pages 1264–1266, 2020.

[49] E. H. Spafford and S. A. Weeber. Software forensics: Can we track code to its authors? *Computers & Security*, 12(6):585–595, 1993.

[50] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Proc. ESEC/FSE*, pages 365–375, 2011.

[51] M. F. Tennyson and F. J. Mitropoulos. Choosing a profile length in the scap method of source code authorship attribution. In *Proc. SOUTHEASTCON*, pages 1–6, 2014.

[52] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proc. ICSE*, pages 1039–1050, 2016.

[53] M. Tufano, G. Bavota, D. Poshyvanyk, M. Di Penta, R. Oliveto, and A. De Lucia. An empirical study on developer-related factors characterizing fix-inducing commits. *Journal of Software: Evolution and Process*, 29(1):e1797, 2017.

[54] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4), 2017.

[55] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *IEEE software*, 17(4):19–25, 2000.

[56] X. Yang, G. Xu, Q. Li, Y. Guo, and M. Zhang. Authorship attribution of source code by using back propagation neural network based on particle swarm optimization. *PloS one*, 12(11), 2017.

[57] W. Zhang, S. Wang, and Q. Wang. Ksap: An approach to bug report assignment using knn search and heterogeneous proximity. *Information and software technology*, 70:68–84, 2016.

[58] H. Zhong and H. Mei. Learning a graph-based classifier for fault localization. *Science China Information Sciences*, 63:1–22, 2020.

[59] H. Zhong and N. Meng. Towards reusing hints from past fixes -an exploratory study on thousands of real samples. *Empirical Software Engineering*, 23(5):2521–2549.

[60] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proc. ICSE*, pages 913–923, 2015.

[61] M. Zhou and A. Mockus. What make long term contributors: Willingness and opportunity in OSS community. In *Proc. ICSE*, pages 518–528, 2012.