

cregit : Token-level Blame Information in git Version Control Repositories

Daniel M. German · Bram Adams ·
Kate Stewart

First revision: Jan 25, 2019

Abstract The blame feature of version control systems is widely used—both by practitioners and researchers—to determine who has last modified a given line of code, and the commit where this contribution was made. The main disadvantage of blame is that, when a line is modified several times, it only shows the last commit that modified it—occluding previous changes to other areas of the same line. In this paper, we developed a method to increase the granularity of blame in git: instead of tracking lines of code, this method is capable of tracking tokens in source code. We evaluate its effectiveness with an empirical study in which we compare the accuracy of blame in git (per line) with our proposed blame-per-token method. We demonstrate that, in 5 large open source systems, blame-per-token is capable of properly identifying the commit that introduced a token with an accuracy between 94.5% and 99.2%, while blame-per-line can only achieve an accuracy between 75% and 91% (with a margin of error of $\pm 5\%$ and a confidence interval of 95%). We also classify the reasons why either blame method fails, highlighting each method’s weaknesses. The blame-per-token method has been implemented in an open source tool called cregit, which is currently in use by the Linux Foundation to identify the persons who have contributed to the source code of the Linux kernel.

D. German
University of Victoria
E-mail: dm@uvic.ca

B. Adams
Polytechnique Montréal
E-mail: bram.adams@polymtl.ca

K. Stewart
Linux Foundation
E-mail: kstewart@linuxfoundation.org

1 Introduction

One feature of version control systems is the ability to know what change (i.e. commit) introduced a specific line of code. This feature is colloquially known as “blame” (and also known as “annotate”). Given a snapshot of the source code (at the present or at any given time) the blame command of version control systems outputs, for each line of code, the commit responsible for modifying it last (including some metadata of this commit, such as its date/time and its author).

Blame has become an important tool in the software development process. One use of blame is to extract information needed to train defect prediction models [27, 29, 62, 60] (specifically, to identify the commit that inserted the bug, using—for example—the SZZ algorithm [37]); another to recommend experts on a given area of the source code [45, 47, 1]. Blame is also used by developers to understand the commits made to a particular area of the system, including who is making them and how the codebase is evolving. Blame has also been used by researchers trying to understand who changes what (eg. [52, 53, 64, 72, 66, 65, 73, 33, 71, 5, 40, 45, 20, 56, 42, 37]).

One of the major disadvantages of blame is that it operates at the line level [43, 22, 58]. If a line has been modified several times, blame only identifies its very last commit—rather than identifying what commits have modified what portions of the line. The typical solution to this problem is to retrieve a snapshot of the system previous to this last commit, and see if the line existed before, and run blame again (to identify the last commit—before this current commit—that change this line). This process is impractical, since it would need to be done for every different commit, and the user would have to—manually—keep track of the commits made to each of these lines (one line at a time). Software projects that tend to reformat source code might be particularly vulnerable to this disadvantage (some version control systems, like git, are able to ignore whitespace changes made to a line while computing blame; others, like SubVersion and ClearCase do not). These issues are exacerbated as the history of a project grows, and the code might be modified over and over again. Even though many methods and tools rely on blame, there are no empirical studies that evaluate its accuracy.

Another aspect that is gaining relevance is the need for legal purposes of identifying who the authors of the source code are. For example, during the last few years Patrick McHardy—a developer of the Linux kernel—has sued companies that distribute hardware products that include the Linux kernel, claiming that he has significant contributions to the kernel [41, 69]. While it is relatively easy to identify what commits he has contributed to the kernel, it is not straightforward to identify what portions of his changes are still present in the current kernel and what code was written by others and subsequently modified by him—specially due to the limitations of blame described above. This is particularly difficult in a system like Linux that has more than 25 years of history and more than 850k commits (as of Dec. 2018, including commits in git and bitkeeper).

In this paper we propose a method that improves blame in git (one of the most popular version control systems today). Rather than tracking the additions and deletions to lines (as typical version control systems do), our method tracks additions and removals of tokens. Doing this, we improve blame by reporting, for every single token in the source code, the commit that has last added/modified that token (rather than the last commit that added/modified the line). This method can be used to improve our understanding of the history of a software system, and in combination with line-based blame, to improve the quality of blame information of a software system. This will, by extension, improve the quality of methods that rely on it (such as manual and automatic origin analysis, defect prediction, copyright ownership analysis). Specifically, our contributions are:

- A method that, given a git repository, creates a new synthetic git repository that has the same commits and metadata as the original repository but tracks a “view” of the source code in the original repository.
- Using this method we create a repository that tracks the source code at the token level (rather than line level). This repository will have the same commits and metadata than the original repository. When running `git-blame` on this repository one will be able to determine the commit that last added/modified a token in the original repository.
- An implementation of this method is an open source tool called `cregit` (<https://github.com/cregit/>) that is already being used to improve blame information at the token level in the Linux kernel [63].
- An empirical evaluation (using five mature open source projects) that compares the accuracy of blame at the line level (using the traditional `git-blame`, which we will refer to `git-by-line`) and blame at the token level (using our method, referred to as `git-by-token`). In this study we show that when tracking the specific commit that adds a token to the source code, line-based blame is accurate between 75 and 91%, while token-based blame is accurate between 95 and 99% of the time. We also categorize the reason why line-based blame and token-based incorrectly identify the commit that adds a given token.

The replication package of this paper can be found at <http://turingmachine.org/2018/cregit>.

2 Motivation and Related Work

In this section, we motivate the need to improve the blame-by-line feature of version control systems. We start by showing an example that highlights the limitations of blame-by-line and follow it with a description of three use cases that require more accurate blame information.

2.1 Motivating Example

<pre> #include <stdio.h> int sum(int i) { int total = 0; while (i-->0) total+=i; return total; } int fact(int i) { if (i == 0) return 1; else return i*fact(i-1); } int main(void) { printf("Fact: %d\n", fact(10)); printf("Sum: %d\n", sum(10)); return 0; } </pre>	<pre> #include <stdio.h> int sum(int i){ int total = 0; while (i-->0) total+=i; return total; } int fact(int i) { if (i == 0) return 1; else return i*fact(i-1); } int main(void){ printf("Fact: %d\n", fact(10)); printf("Sum: %d\n", sum(10)); return 0; } </pre>	<pre> #include <stdio.h> long sum(long i) { long total = 0; while (i-->0) total+=i; return total; } int fact(int i) { if (i == 0) return 1; else return i*fact(i-1); } int main(void) { printf("Fact: %d\n", fact(10)); printf("Sum: %d\n", sum(10)); return 0; } </pre>
Commit c_1 by Dev. A	Commit c_2 by Dev. B	Commit c_3 by Dev. C

Fig. 1 Contents of a simple repository R composed of three commits that track the changes to one file. The color represents the developer who added that code. R is used throughout the paper as running example. *Dev. A* (blue) creates the file in commit c_1 , after which *Dev. B* only changes whitespace. Finally, in c_3 *Dev. C* changes both whitespace and source code (red).

To motivate the concepts introduced by this paper, we use the simple example of Figure 1. It assumes that we have a single file under `git` version control, and shows the contents of three subsequent revisions of the file. The commits responsible for these revisions were done by developers A, B and C (in that order). The first commit adds three functions, the second one only modifies whitespace (merging several pairs of lines into one), while the third one modifies the type signature of the first function (`sum`) and un-does some whitespace changes. The color of the source code reflects the developer who was responsible for adding that code. Commits that reformat source code—frequent in some projects—make it more difficult to determine the origin of code that has been affected by the reformatting [22].

Please note that the purpose of this example is to illustrate the limitations of `git-blame` and in practice the issues described below are rarely rarely this extreme.

2.2 Use Case 1: Who is the expert for this code?

A new developer is hired and, while learning the code base, finds a number of coding decisions that she would like to have more information about. Instead of asking her manager which of the three developers to contact, she uses the built-in `git blame -w` feature of `git` to obtain, for each line in the third revision of the file, who modified it last (the `-w` switch instructs `git` to ignore white space-only changes to a line). She would then contact the resulting

```

c-1 (Dev A 1) #include <stdio.h>
c-1 (Dev A 2)
c-3 (Dev C 3) long sum(long i)
c-3 (Dev C 4) {
c-3 (Dev C 5)     long total = 0;
c-3 (Dev C 6)     while (i-->0)
c-3 (Dev C 7)         total+=i;
c-1 (Dev A 8)     return total;
c-1 (Dev A 9) }
c-2 (Dev B 10) int fact(int i) {
c-2 (Dev B 11)     if (i == 0 ) return 1;
c-2 (Dev B 12)     else return i*fact(i-1);
c-1 (Dev A 13) }
c-3 (Dev C 14) int main(void)
c-3 (Dev C 15) {
c-2 (Dev B 16)     printf("Fact: %d\n", fact(10));
c-2 (Dev B 17)     printf("Sum: %d\n", sum(10));
c-1 (Dev A 18)     return 0;
c-1 (Dev A 19) }

```

Fig. 2 Output of “blame” (`git blame -w`) on the third revision of Figure 1. The “-w” option instructs blame to ignore changes in whitespace, however this option only works when the changes in whitespace are on the same line. Color has been added to ease author identification (blue for Dev A, green for Dev B and red for Dev C).

developers for the code lines about which she has questions. Figure 2 shows the corresponding output of this command.

The figure shows how the “blame” output will give the wrong impression to the new developer (and cause her to ask the wrong person for help—even though this example is small). Although all the lines marked by “blame” as *Dev A* are correct, none of those marked as *Dev B* are, since all developer B really did was merging subsequent lines (by removing a newline character and spaces). Dev A is the actual expert of these code lines, since she wrote most of the initial code; `git`’s “blame” feature (even with the `-w` switch) is not capable of dealing with these types of whitespace changes.

The situation gets worse when considering the lines marked as *Dev C*. It is true that this developer made source code changes. However, it can be argued that lines 2 to 5 are incorrectly attributed to *Dev C*, since all the developer did on function `sum` was to change the argument and return type from `int` to `long` and split one line of code. Again, most of the work, i.e., the actual algorithm of `sum` as well as the decision that this function had to be added, both belong to *Dev A*. In other words, *Dev C* only had a small hand in the code of the function `sum`, but is “blamed” to be responsible for more than 70% (5 out of 7) of its source code lines.

The example suggests that there are two major problems with “blame”: (1) it does not properly handle whitespace changes in case of merging/splitting of lines of code and (2) it attributes the entire line of code to the person who modified it last, regardless of how much it was modified. The first problem is caused by the limitation of “blame” that it only traces back the history of source code by comparing lines one-to-one. It never considers the merging or splitting of lines in its algorithm. The second problem is a well-known limitation of “blame”, for which various researchers have proposed solutions.

```

CURRENT LINE 1:#include <stdio.h>
c-1 Dev A: #include <stdio.h>
CURRENT LINE 2:
c-1 Dev A:
CURRENT LINE 3:long sum(long i)
c-3 Dev C: long sum(long i)
CURRENT LINE 4:{
c-3 Dev C: {
CURRENT LINE 5:    long total = 0;
c-3 Dev C:    long total = 0;
c-1 Dev A:    int total = 0;
CURRENT LINE 6:    while (i-->0)
c-3 Dev C:    while (i-->0)
c-2 Dev B:    while (i-->0) total+=i;
c-1 Dev A:    while (i-->0)
CURRENT LINE 7:        total+=i;
c-3 Dev C:        total+=i;
CURRENT LINE 8:    return total;
c-1 Dev A:    return total;
CURRENT LINE 9:}
c-1 Dev A: }
CURRENT LINE 10:int fact(int i) {
c-2 Dev B: int fact(int i) {
c-1 Dev A: int sum(int i)
CURRENT LINE 11:    if (i == 0 ) return 1;
c-2 Dev B:    if (i == 0 ) return 1;
CURRENT LINE 12:    else return i*fact(i-1);
c-2 Dev B:    else return i*fact(i-1);
c-1 Dev A:    return i*fact(i-1);
CURRENT LINE 13:}
c-1 Dev A: }
CURRENT LINE 14:int main(void)
c-3 Dev C: int main(void)
c-2 Dev B: int main(void) {
c-1 Dev A: int main(void)
CURRENT LINE 15:{
c-3 Dev C: {
CURRENT LINE 16:    printf("Fact: %d\n", fact(10));
c-2 Dev B:    printf("Fact: %d\n", fact(10));
c-1 Dev A:    printf("Fact: %d\n",
CURRENT LINE 17:    printf("Sum: %d\n", sum(10));
c-2 Dev B:    printf("Sum: %d\n", sum(10));
c-1 Dev A:    printf("Sum: %d\n",
CURRENT LINE 18:    return 0;
c-1 Dev A:    return 0;
CURRENT LINE 19:}
c-1 Dev A: }

```

Fig. 3 Output of `git-author` after the third revision of Figure 1. The output shows, for each line, its previous versions (and their corresponding authors). The colors have been added to ease identification of the authors. Note that this algorithm still works at line level, and cannot properly deal with lines that are split (e.g., line 7 is not attributed to its original author, Dev A; while line 6 is attributed to all authors). It also incorrectly thinks that line 10 (function `int fact(int i)`) was derived from the line `int sum(int i)`.

Most recently, Meng et al. developed a git extension (`git-author`) [43] that improves on an earlier algorithm by Canfora et al. to try to match lines more accurately from one revision to another [8]. The output of `git-author` is depicted in Figure 3. As can be seen, `git-author` still tracks only one line at a time and is unable to consistently track lines as they are split or merged. For example, line 7 is incorrectly attributed to *Dev C* and line 6 to all three developers.

Hence, both problems seem to have the same root cause: blame’s line-level “diff” is too coarse-grained. Indeed, while *Dev C* changed only two tokens in the signature of function `sum` and one token in the declaration of variable `total`, the *entire two lines* are considered as one unit by “blame”. A change in even one character of a line will make the entire line appear as removed and then added, obfuscating the origin of the portions of the line that were not changed.

2.3 Use Case 2: Who authored this code?

The problem of ascertaining authorship goes beyond finding out who are experts in the current development team. Patrick McHardy, one of the developers of the netfilter module of the Linux kernel has been actively suing distributors of products that contain the Linux kernel [59, 4, 69]. In [41] Meeker provides a detailed description of McHardy’s actions. In a nutshell, by claiming to be an author of the Linux kernel, he has been attempting to enforce its license; his goal appears not to bring those sued into compliance, but to financially gain from these lawsuits. A major challenge in this case is to determine—using currently available tools—McHardy’s contributions to the kernel and, more specifically, what portions of the current source code were authored by him. This is further complicated by the fact that Linux is 26 years old and has more than 24 million SLOCs.

Meeker used `git-blame` and `cregit` (the tool we propose herein) to identify McHardy’s contributions to the kernel. and emphasizes that “blame might not always tell the whole story. For example, a committer can check in many lines of code having made only minor changes...”. She is particularly concerned with areas of the source code where a developer makes minor changes to a line of code, and blame reports this developer as being responsible for the entire line. Also, the blame feature of Github (build on top of `git-blame`) does not ignore whitespace, and its output can be easily affected by commits that change whitespace (such as reindentations).

2.4 Use Case 3: Where did this code come from?

Blame information can also be used to understand how the source code has evolved. In this scenario, a developer is interested to know why a specific part of the code was introduced. For example, why was the function `fact` added? `git-blame` (as shown in Figure 2) will incorrectly indicate that the second commit was the one that added this function. Debugging is a task that frequently requires this information; for example, Chacon et al. [9, Debugging with Git] emphasize the use of `git-blame` to find the change that introduces a bug. Improving `git-blame` would correspondingly help in the identification of the correct change that introduced the bug. In [35], Shuah Khan, at the time a Samsung developer who contributes to the Linux Kernel, describes

how `git-blame` and the method we propose in this paper (blame-by-token) are useful in the identification (and eventual fix) of Linux bugs. Research has also used `git-blame` to try identify the origin of failing code (e.g. [23]). Any improvement to the accuracy of line-based `git-blame` will benefit any of these methods.

2.5 Related Work

Here we discuss work related to the topics of origin analysis (e.g., “blame” / “diff”) and source code expertise and ownership.

2.5.1 Origin Analysis

Origin analysis focuses on tracking code functionality across multiple revisions of a code base [25, 15]. While traditional “diff” and “blame” tools form the basis of most of the existing approaches, several heuristics have been proposed to deal with the effects of code refactoring activities such as renaming of identifiers or splitting/merging of functions. `git`’s implementation of “dif” and “blame” has adopted a substantial number of these heuristics, for example allowing to track the movement of code snippets between files [10]. Instead of improving line-level “blame” using heuristics, we propose to use finer-grained, token-level “blame”.

Given that “blame” functionality in modern version control systems (such as `git` and Mercurial) is implemented in terms of “diff”, most of the existing work in the area has focused on improving “diff”. Most of this work either focused on line-level “diff” or tree-based “diff”. More recently, finer-grained “diff” has been explored as well. The fundamental “diff” algorithm dates back to the work of Miller et al. [44, 48] and Ukkonen et al. [67]. The algorithm aimed to “compute a shortest sequence of insertion and deletion commands that converts one given file to another” by searching for the longest common subsequences of code lines between file revisions [44]. This initial “diff” algorithm still forms the basis of today’s implementations such as GNU diff and `git`’s “diff” command, even though these added a variety of heuristics and parameterization to improve performance and configuration.

One of the reasons why traditional “diff” still is so popular is its focus on lines of text, independent from any programming language. Reiss [54], in his seminal empirical evaluation of 18 approaches for tracking the location of source code across code revisions, found that language-independent techniques based on Levenshtein distance (improved by also considering as context a small number of lines of code before/after the line to be tracked) improved upon more powerful, tree-based techniques.

For this reason, Canfora et al. [8] and Asaduzzaman et al. [2] decided to leverage base “diff” to find unchanged lines of code, then use more powerful heuristics on the lines that did change. Canfora et al.’s Ldiff [8] first finds matching (changed) code snippets (“hunks”) between file revisions using cosine

similarity on the sets of words inside the snippets, then finds the best candidate by sorting based on Levenshtein edit distance between the original hunk and each candidate hunk. LHdiff [2] instead uses simhashing of each hunk to find sufficiently similar hunk candidates, followed by Levenshtein edit distance to find the best candidate (and a heuristic to track lines that are split). LHdiff especially focuses on cases where both the code line that is being tracked as well as the line’s context change substantially.

In contrast, most of the language-specific “diff” algorithms are tree-based. By exploiting the syntax of source code or the general structure of some other kind of document, these approaches are able to track changes in, say, a for-loop or function body instead of generic lines. The fundamental concept of most of the tree-based approaches is to derive an “edit script” that contains a sequence of operations that can be applied on a tree structure of a given file revision to obtain the next revision [11]. While a wide variety of tree-based approaches exist [3, 6, 16, 18, 26, 32, 46, 49, 51], even for other documents than source code [39, 70], the most commonly used approaches are ChangeDistiller [19] and GumTree [17].

ChangeDistiller [19] uses bigram string similarity to match AST nodes between file revisions, followed by similarity measures at the level of subtrees [11]. GumTree [17] aims to provide scalable tree tracking able to deal with code movement across files. For this, it combines a top-down greedy search for the largest possible, isomorphic subtrees with a bottom-up phase analyzing nodes within the matched subtrees that have no corresponding node. Hata et al. [30] extract methods into separate files, then use `git`’s file-level tracking of code to obtain method-level code tracking. Finally, Hassan et al.’s C-REX [28] and Kim et al.’s LSdiff [36] focus on structural differences between file revisions such as “calls to function `f()` have been added in 3 files”. Both of them extract low-level facts of a given file revision (e.g., “function `g()` calls `f()`”), then use a custom algorithm [28] or logic rules [36] to abstract up and compare these facts between revisions. While the work on tree-based “diff” aims to not only find edit scripts that are as concise as possible, but also to obtain scripts that are as close as possible to the developers’ original intent, we instead aim to track code as accurately as possible across file revisions, with major use cases in terms of code authorship and expertise.

In parallel with the coarser-grained, tree-based approaches, various approaches have been developed that focus on a finer granularity, similar to our work. Spacco et al.’s SDiff [61] focuses at the statement level. They represent each statement as a sequence of tokens, then use diff to compare the corresponding token strings (statements) between file revisions, followed by minimum weight bipartite graph matching (with statements as nodes) to find the most likely matching code hunks. Similar to tree-based matching, SDiff is language-specific since it uses a code tokenizer. LHdiff [2] also experimented with tokenization. Each code line was tokenized as is (without putting each token on a separate line) and LHdiff was applied on the resulting tokenized lines (instead of `git`’s “diff”). No improvement in performance was found compared

to the standard LHdiff. In contrast to SDiff and LHdiff, we split each line at token level, then apply regular line-level “blame” on the resulting files.

2.5.2 Source Code Expertise and Ownership

Determining code expertise and ownership are major use cases of origin analysis approaches, and the “blame” feature of version control systems is widely used for this purpose. “Blame” typically is built on top of “diff” in the sense that “blame” starts with the first revision of a file, and iteratively compares it to the next revision (using “diff”), in order to determine what changes have been done at each revision. While “diff” has been an active area of research, significantly less work has been done in attempting to improve “blame” with this research.

The most extreme “blame” implementation is the PCC approach by Tsikerdekis [66], which tries to track the survival of individual characters in order to obtain more accurate code ownership. Inspired by work on the editing of Wikipedia pages [50], PCC tries to follow characters across revisions, then uses the survival period of all characters contributed by a project member as additional data source to measure the member’s expertise (instead of just considering all characters as equal). The use of characters as indivisible units makes the approach language-independent, since no language-specific parser nor tokenizer is required. However, this also leads to unexpected traceability, such as characters in comments evolving into source code entities.

Servant et al. [57] (history slicing) and Meng et al. [43] explored line-level recursive “blame” approaches that do not stop at the most recent commit but continue going back in time until the initial addition of a given line of code. Meng et al.’s `git-author` tool enhances the recursive “blame” information by assigning weights to authors based on the number of characters that they contributed to the current revision of a file (and how long those survived). For about 10% of the lines that Meng et al. evaluated, `git-author` was more accurate than `git-blame`, and it also allowed to build effective line-level bug prediction models. While `git-author` is capable of identifying the different changes made to a given line, it still tracks source code at the line level, and incorrectly deals with lines of code that are split or merged. It also tends to be confused when two lines are too similar to each other (as demonstrated in Figure 3).

Since it is not always trivial to determine whether a line is newly added or a (heavily) mutated version of another line [8], Servant et al. evolved their concept of history slicing with fuzzy elements. This means that instead of saying “this line IS a changed version of that line”, the fuzzy slices would say “there is X% chance that this line is a changed version of that line”. Their approach represents a compromise in precision and recall between line-level “diff” approaches (high recall, low precision) and approaches that use optimization techniques to map lines between file revisions (high precision, low recall). Tsikerdekis et al. [66] compared their PCC approach for recommending code experts to heuristics based on the percentage of commits made or files

touched [7], `git-blame`, `git-author` [43] and approaches based on effort estimation, and found that PCC provides additional information not provided by those approaches.

Substantial research has focused on determining or estimating software expertise and knowledge. Similar to a file-level `git-blame`, McDonald [40] recommends the developer who most recently committed to a file as expert for that file. This approach was improved by Mockus et al. [47], who consider the number of changes made by a developer, and by Girba et al. [24], who also consider the churn of these changes. The relation between expertise and both commit frequency and churn was empirically validated by Fritz et al. [20] through 19 interviews with Java developers. However, Bhattacharya et al. [5] argue that such change-level metrics, unaware of the developer’s actual role, can lead to inaccurate results.

Hence, later expertise models explored different kinds of expertise, such as usage expertise. Their assumption is that one does not only gain knowledge about a code base by making changes to it, but also by using it through method calls [56]. In later work [38], these authors empirically found that both usage and implementation expertise recommenders perform similarly. Fritz et al. [21] used this insight to create a degree of knowledge model to recommend experts by combining both usage and implementation metrics.

Finally, Hattori et al. [31] take into account insights from psychology in which memory, and hence expertise, is not a constant concept, but decays over time if knowledge is not refreshed. Hence, they leverage IDE interaction data to determine the recency of interacting with source code files (and hence the freshness of a developer’s knowledge about these files).

3 repo Views

As we described in the previous section, the main limitation of `git-blame` (and other tools created to determine the origin of source code) is that they attempt to track the origin of lines of code, which, conceptually, are divisible and mergeable. This limitation becomes amplified when a project is composed of many different files modified a large number of times. For example, version 4.15 of the Linux kernel is composed of more than 62k files, and is the result of more than 750k commits.

Our proposal to address this limitation is to improve the granularity of “blame”. Instead of tracking lines of source code, “blame” should track tokens of source code. A token (in the syntactic programming language sense) can be considered as a non-divisible unit of source code. Under this assumption, a token cannot be modified; it can only be removed or replaced by another token. A “blame” by token would identify, for every single token in the source code, the commit that inserted it, including the ability of tracking its movement in the file and across files, similar to the way that `git-blame` does today.

Conceptually, one could obtain a token-level “blame” from line-level “blame” by creating a mirror `git` repository that tracks a given `git` repository, but in

```

c-1 (Dev A 1) #
c-1 (Dev A 2) include
c-1 (Dev A 3) <stdio.h>
c-3 (Dev C 4) long
c-1 (Dev A 5) sum
c-1 (Dev A 6) (
c-3 (Dev C 7) long
c-1 (Dev A 8) i
c-1 (Dev A 9) )
c-1 (Dev A 10) {
c-3 (Dev C 11) long
c-1 (Dev A 12) total
c-1 (Dev A 13) =
c-1 (Dev A 14) 0
c-1 (Dev A 15) ;
c-1 (Dev A 16) while
c-1 (Dev A 17) (
c-1 (Dev A 18) i
c-1 (Dev A 19) --
c-1 (Dev A 20) >
c-1 (Dev A 21) 0
c-1 (Dev A 22) )
...

```

Fig. 4 Excerpt of “blame” (`git blame -w`) on the third revision of Figure 1, assuming that the source code has been formatted such that each line contains at most one token. As can be seen, blame correctly identifies the changes as described in Figure 1.

which each file in a commit has been reformatted such that there is only one token per line. One could then use regular line-level `git-blame` on this reformatted version of the source code, since tokens are indivisible and cannot be partially modified. Figure 4 illustrates part of such a tokenized commit for the example in Figure 1.

This section introduces the concept of a “view” of a version control repository (or *repoView*). A *repoView* is a repository that has been created from the contents of an existing repository by mapping every commit in the original repository to one commit in the *repoView*. The latter commits, instead of “committing” regular files as in the original repository, commit their “view”, which is simply a filter that takes as input a source code file and outputs a textual view. Hence, every commit in the *repoView* corresponds to one and only one commit in the original repository, and every revision of a file is a “view” of its corresponding revision of the same file in the original repository.

3.1 Illustration of a *repoView*

Figure 5 illustrates a *repoView* for our running example. For this example, we have used a tokenizer filter to create “views”. This tokenizer converts the original source code file into a version in which each token occupies one line (we have used `srcml` [14,13] for this purpose). Note that each line (token) contains both the type and value of a C-language token. For example, the line `#include <stdio.h>` has been divided into three tokens: `#`, a preprocessor directive; `include`, the include directive of the preprocessor; and `<stdio.h>`,

preprocessor #	preprocessor #	preprocessor #
directive include	directive include	directive include
file <stdio.h>	file <stdio.h>	file <stdio.h>
file <stdio.h>	file <stdio.h>	file <stdio.h>
name int	name int	name long
name sum	name sum	name sum
parameter_list (parameter_list (parameter_list (
name int	name int	name long
name i	name i	name i
parameter_list)	parameter_list)	parameter_list)
block {	block {	block {
name int	name int	name long
name total	name total	name total
init =	init =	init =
literal 0	literal 0	literal 0
decl_stmt ;	decl_stmt ;	decl_stmt ;
...
... view(c ₁ , token)	... view(c ₂ , token)	... view(c ₃ , token)

Fig. 5 Excerpts of the token view of the 3 revisions of the file in the example of Figure 1. Each token in the source code is separated onto one line and annotated with its type. The colour has been added—as in Figure 1—to easily identify what has been changed in each commit.

the file name. We call this view the *token view of the source code*. As in Figure 1 we have coloured the source code according to the authors of the change.

Because each file in the *repoView* has one feature of the source code (i.e., one token) per line, and because `git-blame` is designed to track changes in lines between commits, blame can easily detect the changes in these features and output the correct attribution for each token. This is illustrated in Figure 6.

This *repoView* addresses each of the three use cases described in the previous section. First, it addresses use cases 1 and 2 by correctly crediting each token to its corresponding author: we can see that *Dev C* is only credited with the tokens `long`, and the rest is attributed to *Dev A*. *Dev B* is not attributed any token (since she only modified whitespace). Second, it also addresses the third use case because each token is correctly associated with the commit that introduced it.

3.2 Model

In order to formalize the concept of *repoView* illustrated in the previous subsection, we need to introduce some concepts. First, we define a view of a file f as a mapping ϕ , such that $\phi(f)$ should satisfy the following properties:

1. for any text file f , $\phi(f)$ is also a text file;
2. if f_1 and f_2 are two consecutive revisions of a file, the textual diff between $\phi(f_1)$ and $\phi(f_2)$ represents the difference between f_1 and f_2 .

These two properties guarantee that if we track (using version control) the changes to the views of a file (instead of the changes to the file itself),

```

c-1 (dev A 1) preprocessor|#
c-1 (dev A 2) directive|include
c-1 (dev A 3) file|<stdio.h>
c-3 (dev C 4) name|long
c-1 (dev A 5) name|sum
c-1 (dev A 6) parameter_list|(
c-3 (dev C 7) name|long
c-1 (dev A 8) name|i
c-1 (dev A 9) parameter_list|)
c-1 (dev A 10) block|{
c-3 (dev C 11) name|long
c-1 (dev A 12) name|total
c-1 (dev A 13) init|=
c-1 (dev A 14) literal|0
c-1 (dev A 15) decl_stmt|;
c-1 (dev A 16) while|while
c-1 (dev A 17) condition|(
c-1 (dev A 18) name|i
c-1 (dev A 19) operator|--
c-1 (dev A 20) operator|>
c-1 (dev A 21) literal|0
c-1 (dev A 22) condition|)
...

```

Fig. 6 “blame” output on the third revision of Figure 5, showing only the changes to the first function, due to space constraints.

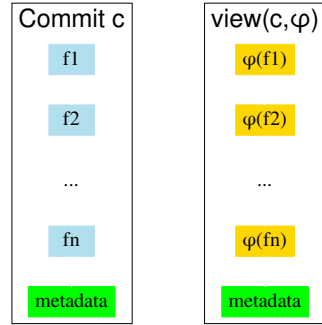


Fig. 7 Anatomy of a commit in terms of its file revisions and metadata, and its corresponding view according to the ϕ mapping. The metadata of the commit remains unchanged through the mapping.

the output of “diff” (the lines added and removed by a change) have some meaning to a developer who is inspecting the change.

A view of a repository R , based on the mapping ϕ , is created by replacing each commit in its DAG with an equivalent commit (with equivalent ancestors), where the contents of this new commit are the views of each of the files (using ϕ) modified in the commit. In other words, the view repo tracks the views of the files (instead of the original files).

This process is illustrated in Figure 7 and Figure 8. Figure 7 shows how a commit in the view repo is created. Each of these commits tracks the same

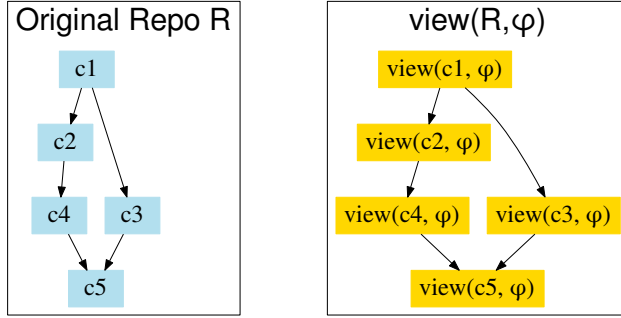


Fig. 8 DAG of the original repository R and of its view according to mapping ϕ . The two DAGs show the bijective nature of views.

files, but instead of containing the original files, it contains their views (using ϕ). Note that for each commit in the repo R , there is only one view commit in the view repository. After all the new view commits have been created, the view repo is created by reproducing the DAG of R , but replacing each commit in it with the commit's view; this is shown in Figure 8. Finally, the tags and branch names of the repo R must be created in the view repo (they should point to their corresponding view commit).

4 Implementation

We have implemented *repoViews* in an open source tool called **cregit** (available at <http://github.com/cregit>). Given a filter that implements a view of the source code (such as the token view described above), **cregit** can create the *repoView* of a git repository using this filter. In this section we describe the details of our implementation.

4.1 Creating a *repoView* repository

As described in Section 3, a *repoView* of a repository is created by using a filter ϕ to replace each of its commits with their *view* commits.

Figure 9 shows the concrete structure of a commit in **git**: a) its metadata, b) an ordered list of ancestor commits and c) the top-level blob identifier of the resulting root directory after the commit is applied. The blob identifier of such a directory is built, recursively, from the SHA1 of the blobs of the files and directories it contains. In turn, the blob identifier of a file is the SHA1 of the contents of the file. As such, a **git** repository is effectively a database of blobs indexed by their blob identifier, a set of commits that reference these blobs and each other (a commit references its parents) and a set of labels that map

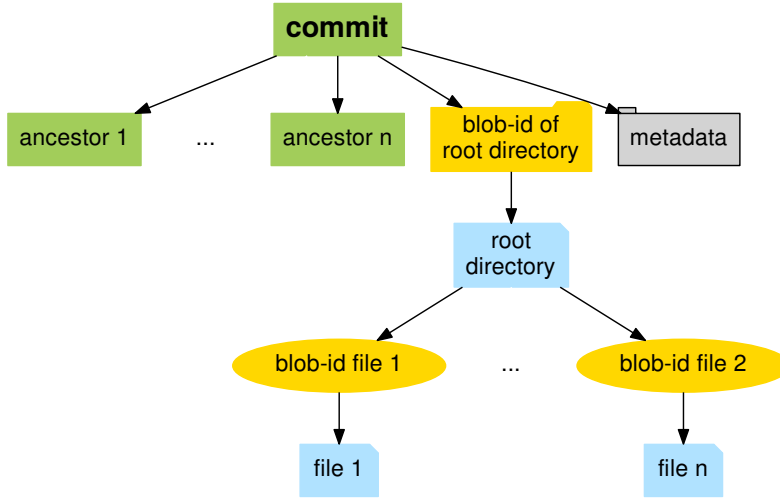


Fig. 9 Structure of a commit as stored in git.

a string to a given commit identifier (the tags). These labels correspond to a) the name of each branch, including HEAD, in the repository (each mapping to the commit identifier that corresponds to their head) or b) an explicit “tag” name (mapping to a given commit identifier that corresponds to that tag).

Algorithm 1 lists our algorithm used to map a given `git` repository R to a `git repoView` V . It basically consists of four phases. In the first phase, we create the blobs of the views of each blob. In the second phase, we traverse the DAG of the repository from its root(s) and replace each commit in it with its corresponding view. To facilitate going from a commit in the `repoView` to the original repository we also append to the log of the commit its original commit identifier (since the commit and its view will have a different commit identifier). In the third phase, the commit identifiers of the labels are replaced with the commit id of their corresponding view. The fourth phase removes all dangling commits and blobs, i.e., the commits and blobs of the original repository that are no longer reachable from any label (branch or tag).

The resulting `repoView` has one `view` commit for each commit in the original repository, which will have as ancestors the views of the ancestors of its original commit. Its blob-id will be replaced with the blob-id that corresponds to the `view` of the filesystem in the original commit. This relationship is depicted in Figure 10. Since our algorithm replaces the current repository with its corresponding view, it should be performed on a clone of the original repository.

```

Data:
   $R \leftarrow$  git repo to process
  a mapping  $\phi : file \rightarrow file$ 
Result:
  the repoView  $V$  of  $R$ 
# Phase 1: create views of blobs
for each blob  $b \in R$  do
  | create a new blob  $b' = view(b, \phi)$ ;
  | insert  $b'$  into  $V$ 
end
# Phase 2: Create views of commits
for traverse each commit  $c \in R$  starting at the roots of the DAG do
  | create a new commit  $c' = view(c, \phi)$  such that
  |    $metadata(c') = metadata(c) + \text{original commit-id}$ ;
  |   for every blob  $b \in c$  do
  |     |  $b' \leftarrow view(b, \phi)$ ;
  |     | add  $b'$  to  $c'$ ;
  |   end
  |   for every  $i$ -th ancestor commit  $a_i$  of  $c$  do
  |     | make  $view(c_i, \phi)$  the  $i$ -th ancestor of  $c'$ 
  |   end
  | add  $c'$  to  $V$ 
end
# Phase 3: Replace the commit-ids of labels with their views
for every label  $b \in R$  do
  | replace its commit  $c_l$  with  $view(c_l, \phi)$ 
end
# Phase 4: Remove blobs and commits of original repo
Remove all dangling commits and blobs from  $R$ ;

```

Algorithm 1: Algorithm to convert a git repository to a *repoView*.

Our implementation of algorithm 1 is based on BFG Repo-Cleaner¹. BFG Repo Cleaner is a tool used to replace strings in every blob in a repository and/or remove blobs in a git repository that contain certain strings. We have extended BFG with the ability to replace the blobs of each commit with the output of a dispatcher program. This program reads the contents of a file-blob from standard input, and dispatches the actual filter command that must be run based on the filetype of the blob (e.g., a C tokenizer for C source files), and prints the resulting *view* to standard output (from which BFG reads it).

4.2 Views

The core of our method relies on filters that implement the desired view mapping (breaking the code into atomic entities, one per line). The simplest view to implement would be to break the source code into one word or special character per line, without any consideration to the syntax of the language. However, we discovered that comments contribute a significant portion of the text in the

¹ <https://rtyley.github.io/bfg-repo-cleaner/>

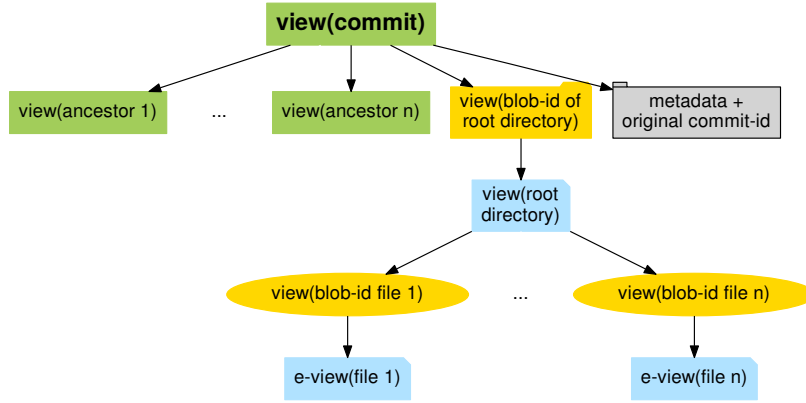


Fig. 10 Structure of the *view* of the *git* commit of Figure 9).

Linux kernel; according to `cloc`², 17.5% of the non-empty lines of the kernel are comments; also, comments lines tend to have more words than source code lines. We considered that splitting all source code (including comments) into each of its words would have added too much noise (specially because one of our original goals was to track who are the authors of the source code, not who comments it). Another issue that would arise when splitting codes by words without a tokenizer is that it would not be possible to distinguish the use of an identifier in the source code from a comment that mentions it.

For these reasons we decided to implement a view that breaks each token into one line (which we have presented in section 3). The advantage of this view is that we can record in each line the token itself and its type (when the tokenizer outputs that data). This allows us, using diff, to distinguish additions or removals of tokens by type (eg. we can distinguish if a comma being added is a parameter separator or a separator of variable declarations); in addition, by looking at the context of an identifier we can see whether it is its declaration or its use. The main disadvantages of using a tokenizer is that it is slower than simply breaking words, and we require one for each programming language. Fortunately we were able to leverage `srcML` [14] for our needs. Also, many scripting languages are capable of tokenizing their own languages (e.g. perl and go).

Throughout our experiments with the *token* view, we found that it is beneficial to not only output the tokens (as they appear in the source code), but also to add annotations that convey some structural information that can be used to identify where the token appears. Such annotations add contextual information that makes it easier to match a location in the original source code with

² `cloc` is an OSS tool to count lines of code and comments <https://github.com/AlDanial/cloc>

<code>function fact</code>	<code>function fact</code>	<code>function fact</code>
<code>function main</code>	<code>function main</code>	<code>function main</code>
<code>function sum</code>	<code>function sum</code>	<code>function sum</code>
<code>view(declarations, c₁)</code>	<code>view(declarations, c₂)</code>	<code>view(declarations, c₃)</code>

Fig. 11 *declarations* view of the source code from Figure 1. All declarations are in blue because they were added by Dev. A.

its tokenized view. They are markers inserted into the tokenized source code that inform of some structural information. Currently there are three types of annotations: *begin*, *end*, and *global declaration* annotations. *Begin* and *end* annotations delimit a global entity in the source code (for example, where a function starts and ends), and the declaration annotations document the location and name where a global identifier is defined. Figure 12 demonstrates their use.

```

begin_unit
include|#
file|<stdio.h>
begin_function
DECL|function|sum
name|long
name|sum
parameter_list|(
name|long
...
end_function

```

Fig. 12 Excerpt of the output of the token view of `cregit`. The lines in grey represent annotations, while the lines in black represent tokens that exist in the source code.

We have implemented the *token* filters for the following languages: C, C++, Java, Go and Python. For C/C++ and Java, we use `srcml` [14,13] and process its output with a custom Xerces XML filter. For Go and Python, we use their own built-in parser modules. The parsing information provides the token records, and we use exuberant ctags to insert the annotations to the output of the parsers.

The second view that we implemented is called *declarations*. This view is a filter that outputs the list of global identifiers (variables, constants, methods, classes, functions, etc) defined in the source code file in lexicographical order. The purpose of this filter is to permit tracking of the structure of the source code. Figure 11 shows the output of this view for the file revisions in Figure 1.

For the *declarations* mapping, we exclusively used exuberant ctags. We keep only the type and name of global identifiers and sort them by type and name (to make this view resilient to movement of entities in the same source code file).

Name	Date	Lang	Commits	Files	SLOCs	Tokens
Linux 4.11	30-04-17	C	728,152	46,165	14,764,301	88,077,930
git 2.13	09-05-17	C	49,519	562	182,427	1,192,235
Guava 22.0	20-06-17	Java	5,390	3,079	313,411	3,550,115
ElasticSearch 5.4.1	19-06-17	Java	27,916	5,352	671,159	5,652,039
Lucene-Solr 6.6.0	29-05-17	Java	48,070	6,766	985,231	7,419,783

Table 1 Characteristics of the five projects used in this study. We only count files and slocs of Java and C source code files. For Linux, we have concatenated the BitKeeper histories of Linux before release 2.4.0 (64,469 commits) with the current history.

5 Evaluation

The initial goal of **cregit** was to increase the accuracy of “blame” information. For this reason, we performed an empirical study aimed towards comparing the accuracy of blame-by-line (regular “blame” using the original repository) versus blame-by-token (regular “blame” on the *token* view of the original repository). The goal of the study is to identify, for a given token, which commit originally inserted it into the source code, and determine whether (and why) blame per line or blame per token (or both) were correct in identifying that commit.

Specifically, our study addresses the following research question:

RQ1 Is blame-by-token more accurate than blame-by-line?

The results of this research question showed significant differences in both methods. We were particularly interested to see why each method was unable to correctly identify certain correct commits. We also discovered that in some cases, there was more than one commit that could be considered to have added the token (we refer to these tokens as having ambiguous origin). For this reason, we added two extra research questions to our study:

RQ2 Why do blame-by-token and blame-by-line fail?

RQ3 Why can the origin of a token be ambiguous?

5.1 Case Study Setup

We selected five large, mature open source projects, each developed by a large community of developers: Linux, Git, ElasticSearch, Guava, and Lucene-solr. Their statistics regarding size, number of commits and number of contributors (commit authors) are shown in Table 1. The first two projects are written in C and the latter three in Java.

For each of these projects, we randomly selected 384 tokens that were not comments. We ignored comment tokens—our tokenizer converts a comment into a single token (even multiline comments) because we are mainly concerned with the origin of source code. Given the number of tokens of each project, and assuming a binomial distribution with unknown probability [12] and a confidence interval of $\pm 5\%$ with a confidence level of 95%, we obtained a

random sample of 384 tokens per project. In other words, if, for example, we find that a given “blame” technique is correct in 90% of the sampled tokens, this technique is correct for any token in the project $90 \pm 5\%$ of the time (with a confidence of 95%).

To address the research questions, we first created—using `cregit`—the token-view repository of each of the five repositories. The next step was to identify, for each token in the random sample, the commit that blame-per-line and blame-per-token identified as the responsible for inserting the token. The time it took to process the repositories in a computer with 4 cores and an SSD was: ElasticSearch 3.5hrs, Git 1.25 hrs, Guava .5 hrs, Linux 35.5 hrs, and Lucene-Solr 8.25 hrs. Most of this time was spent tokenizing every revision of every source code file ever recorded in each repo.

5.2 Identifying whether either blame is correct

Every token t in the sample has a location $\langle file, line_number, column_number \rangle$ in the line repo and a location $\langle file, line_number_t \rangle$ in the token repo. Using blame in each repo we can easily find the commit C_L in the line repo responsible for $\langle file, line_number, column_number \rangle$; similarly we can find the commit C_K responsible for $\langle file, line_number_t \rangle$. We inspect the patch of both commit (line repo and token repo) and evaluate which of two patches correctly inserts t . For this task, we used Github’s blame and the diff features of Github. Github’s diff is specially useful because it not only shows the lines that were changed in the commit, but also highlights the words changed in each line (in the line-based repository).

Because we also want to understand the reasons why blame (in both line-based and token-based repos) fails, we try to identify why blame reports an incorrect commit. When blame in one repo is incorrect (in some cases it is incorrect in both repos) we start traversing the history of the repo to try to find the commit that truly inserted t . We do this with the help of Github, since Github makes it easy to navigate to the state of the source code before any given commit. For example, if commit C did not introduce t , we go back to the first parent commit of C (i.e., the moment immediately before C) and locate the line of interest. This line likely is in a different line number, and sometimes in a different file, but the line must exist; otherwise C would have been correct. For the line-based repo, we also locate the column number containing t .

At this point we recursively use the same method above (using blame and diff) to identify the commit that added this line, and if this commit is the one that added t . We do this until we identify the correct commit. We also assist our search in one repository with the information we gather in the other repository. While we do most of these operations using the GUI in Github, we also run git commands directly in a local repository and use shell commands (such as `grep`) to help find the location of the token in previous commits.

This process was tedious, but it allowed us to traverse the history of each token. With this information we are able to state if a method is correct, and

when it is not, we documented a brief explanation of why a method was wrong. We then grouped these explanations into a set of categories that explain each failure. All this information is in the replication package.

Note that our goal is not to identify how a token is copied or refactored, but simply, what commit was the one that inserted the token in its final location. For instance, if a commit copies (moves) a token from another location, we are interested in the commit that made the copy (move), not the one that added the original token. Note that `git-blame` is capable of tracking renames of files (we found, however, some limitations to this feature, which we report in subsection 5.4.4), hence both blame-by-line and blame-by-token are usually capable of following the location of a token even when its filename changed.

For both methods, we ran `git-blame` without any extra parameters. One of the main reasons is that this is how `git-blame` in GitHub is computed, and we used GitHub to inspect and navigate the history of the projects. In the next section we will discuss the impact in our study of running `git-blame` ignoring whitespace.

For each token in the random sample, there are five potential outcomes:

- **Both Ok:** blame-by-token and blame-by-line point to the same, correct commit;
- **Line Ok:** blame-by-line points to the correct commit, while blame-by-token does not;
- **Token Ok:** blame-by-token points to the correct commit, while blame-by-line does not;
- **Both Wrong:** both methods point to incorrect commits;
- **Ambiguous:** the provenance of the token is ambiguous and both methods point to a commit that could be considered as the one that added the token.

Please note that all the results in this section have a margin of error of $\pm 5\%$.

5.3 RQ1. Is blame-by-token more accurate than blame-by-line?

Both methods agree for in between 65% and 89% of the tokens in the random sample. This can be seen in Table 2, which shows the results of each method in text form, and Figure 13, which plots the same results graphically. For the tokens without agreement, blame-by-token is correct between 8.6% and 22.7%, while blame-by-line is correct between 0.3% and 3.1% of the cases. Furthermore, a significant number of tokens yielded an ambiguous answer (both methods arguably identified a proper commit) between 1.3% and 6.8%. Only in up to 2.6% of the cases, both approaches were wrong.

Overall, blame-by-token provided correct results for 94.5% to 99.2% of the cases, compared to 74.8% to 90.9% for blame-by-line. These aggregate results are depicted in Table 3. Blame-by-line finds the correct

Project	Both Ok	Ambiguous	Line Ok	Token Ok	Both Wrong
Elastic-search	65.1	6.8	2.9	22.7	2.6
Git	73.4	2.6	3.1	18.5	2.3
Guava	89.3	1.3	0.3	8.6	0.5
Linux	72.1	2.6	2.9	20.6	1.8
Lucene-solr	71.2	7.0	2.6	18.1	1.0

Table 2 Results of the classification of tokens in the five studied projects. All numbers are percentages; the total number of tokens analyzed per project was 384. The five categories are mutually exclusive.

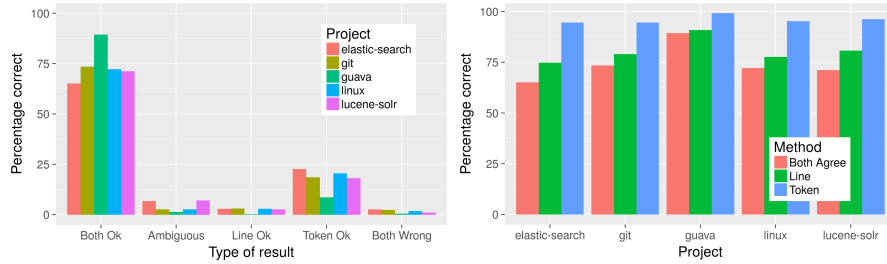


Fig. 13 Breakdown of the accuracy (see Table 2 for details) and correctness of each blame method (Table 3), for each project.

Project	TOKEN	LINE	Difference
Elastic-search	94.6	74.8	19.8
Git	94.5	79.1	15.4
Guava	99.2	90.9	8.3
Linux	95.3	77.6	17.7
Lucene-solr	96.3	80.8	15.5

Table 3 Results of the accuracy of each method in the five studied projects (all numbers represent percentages). The columns LINE and TOKEN correspond to the proportion of tokens in which each method is correct (LINE=Both Ok \cup Ambiguous \cup Line Ok, and TOKEN=Both Ok \cup Ambiguous \cup Token Ok). The total number of tokens analyzed per project was 384.

commit that inserts a token when either both methods agree (Both Ok), blame-by-line is correct (Line Ok) or the provenance of the token is ambiguous (both methods can be considered correct). Similarly, blame-by-token is correct when both methods agree, blame-by-token is correct or the provenance of the token is ambiguous. The median accuracy of blame-by-token in the five projects is 95.3% compared to 79.1% for blame-by-line.

As can be seen, the proportion of correct tokens in each category is fairly consistent. For both “blame” techniques, Guava has the highest percentage of accuracy. We inspected the history of the project and discovered that Guava was imported into its current version control repository long after its development started (the import occurred at version 9.09.15). As such, the current `git` repository does not include all its history and hence many tokens are blamed back to the (large) import commit.

Reason	Linux	Luc.	Guava	Git	Elas.	Total	Ratio
Blame-by-token							
<code>diff</code> has incorrect change alignment.	18	14	3	21	21	77	100%
Blame-by-Line							
Line was modified after the token is inserted.	64	59	23	68	91	305	82.2%
Whitespace was modified after token is inserted.	15	7	10	5	3	40	10.8%
<code>diff</code> has incorrect change alignment.	7	7	2	7	0	23	6.2%
File rename not detected.	0	0	0	0	3	3	0.8%

Table 4 Breakdown per project regarding the reasons why identifying the origin of a token yielded wrong results.

When considering the Line Ok and Token Ok columns of Table 2, we noticed that in between 8.6% to 22.7% of the samples were correctly classified only by blame-by-token (median of 18.5% across the five projects), compared to 0.3% to 3.1% only by blame-by-line. This difference is the main responsible for the difference between TOKEN and LINE. Furthermore, a median of 2.6% (1.3% to 7.0%) of tokens had ambiguous roots. We explore the reasons for misclassification and ambiguous tokens in RQ2 and RQ3.

Using blame-by-token in the *token* view of a repository significantly increases the accuracy of blame with respect to blame-by-line on the original repository, an improvement between 8.3 and 19.8% among the studied projects (with a margin of error of $\pm 5\%$ with confidence 95%).

5.4 RQ2. Why do blame-by-token and blame-by-line fail?

To understand the reasons why each “blame” approach would fail in its identification of the origin commit of a token, we manually analyzed the reasons why each method failed. This analysis yielded four main categories. They are summarized in Table 4 and described below.

5.4.1 *diff* has incorrect change alignment

When `git` is asked to compute the “blame” of a file, it uses the built-in command “`diff`” to sequentially compare each two consecutive versions of a file until the origin of each line is found. Hence, `git-blame` relies on “`diff`” to accurately determine what lines have been changed by a given commit (crediting those lines to this commit). As mentioned in Section 2.5.1, `git`’s internal implementation of `diff` uses the Myer’s algorithm [48], which finds the minimum common subsequence between two strings (considering lines as indivisible). It is a greedy algorithm, and its main advantages are that it is fast, space efficient and minimizes the size of the patch (it runs in $O(ND)$, where N is the

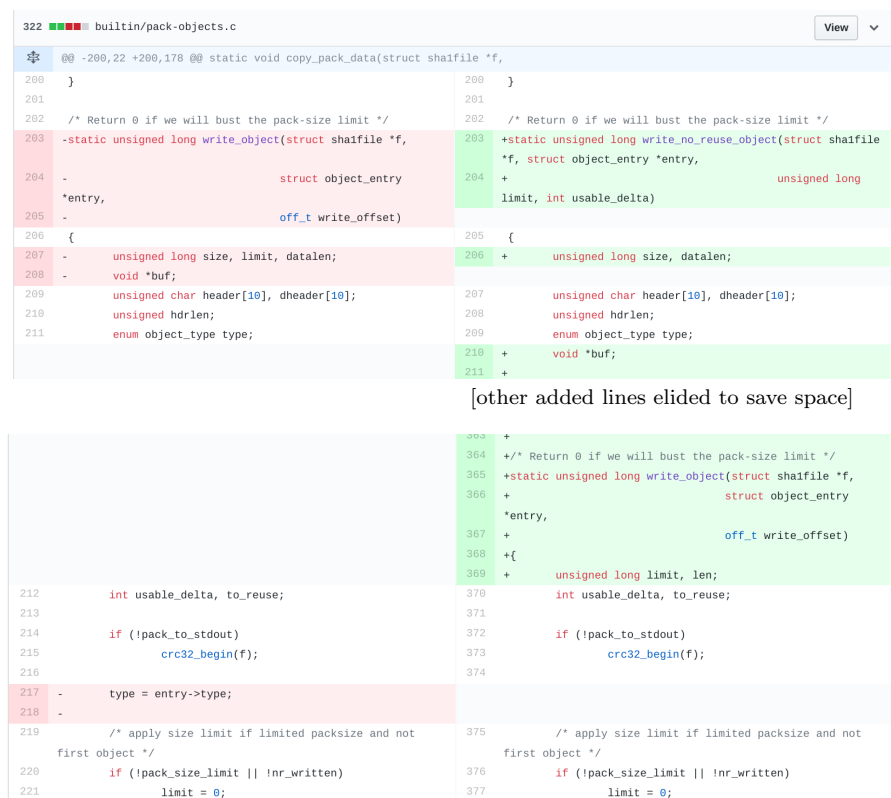


Fig. 14 Example of improper alignment of hunks in commit c9018b0305a56436c85b292edbeacff04b0ebb5d of the `git` project.

total length of the two input strings and D is the length of the diff). Its main disadvantage is that, in many cases, its output does not “intuitively” reflect what the developer has changed³.

Sometimes, the output of Myer’s “diff” does not truly reflect the changes made by the developer. During our analysis we discovered that in many instances “diff” incorrectly aligned changes, implying that the developer has changed different lines than the ones that were most likely changed. This happened when new code that is added is relatively similar to the existing code in the location where the new code is inserted. An example is shown in Figure 14. This figure shows the “diff” of a commit in the `git` project⁴ that added a function `write_no_reuse_object` (line 203) and moved the function `write_object` from line 203 to line 364. However, diff outputs that `write_object` is being

³ For this reason, other “diff” algorithms have been proposed, such as “patient diff” (originally implemented in the version control system Bazaar, and also implemented in `git`). Patient-diff tries to maximize the number of unique unchanged lines by repeatedly running Myers’ diff on sections of the input). For a discussion of its benefits, we refer elsewhere [55].

⁴ <https://github.com/git/git/commit/c9018b0305a56436c85b292edbeacff04b0ebb5d>

replaced by `write_no_reuse_object`) and a new function `write_object` is added (at line 364). The “diff” is incorrect because the lines 203 to 208 look very similar to the lines 203 to 206 in the new file, and diff considers that lines 203 to 208 in the old file have been replaced with the new lines rather than pushed down as a result of the new added code. Thus, blame improperly identifies this commit as the one that adds `write_object`.

We found that this type of error was more common in blame-by-token because sequences of inserted tokens are more likely to be similar to sequences of tokens already in the code, in comparison to sequences of inserted lines being similar to lines already in the code. This is because each original line of code is expanded to several lines in the token repository. All the cases where token-blame was incorrect were attributed to this problem. One of the main issues with this problem is that once a change is misaligned, the whole history of the section affected by this misalignment will have an incorrect blame history, for either “blame” method. A recent improvement to git-diff called “indent-heuristic” tries to address this problem. Unfortunately it is not available to git-blame, see <https://hackernoon.com/whats-new-in-git-2-11-64860aea6c4f> for an example of the use of this feature). Note that these features were written primarily to “make patches easier to read.”

There was one special case of this error that occurred only in blame-by-token: a new line inserted above an almost identical line results in an incorrect change alignment. This case is further discussed below under subsection 5.5.2.

5.4.2 Line was modified after the token is inserted

✱ @@ -398,9 +388,9 @@ static int brcmstb_gpio_irq_setup(struct platform_device *pdev,	
398 if (priv->can_wake)	388 if (priv->can_wake)
399 bank->irq_chip.irq_set_wake =	389 bank->irq_chip.irq_set_wake =
brcmstb_gpio_irq_set_wake;	brcmstb_gpio_irq_set_wake;
400	390
401 - gpiochip_irqchip_add(&bank-> bgc ,gc, &bank->irq_chip,	391 + gpiochip_irqchip_add(&bank->gc, &bank->irq_chip, 0,
0,	
402 handle_simple_irq, IRQ_TYPE_NONE);	392 handle_simple_irq, IRQ_TYPE_NONE);
403 - gpiochip_set_chained_irqchip(&bank-> bgc ,gc, &bank->	393 + gpiochip_set_chained_irqchip(&bank->gc, &bank->
irq_chip,	>irq_chip,
404 priv->parent_irq,	394 priv->parent_irq,
brcmstb_gpio_irq_handler);	brcmstb_gpio_irq_handler);
405	395
406 return 0;	396 return 0;

Fig. 15 Example of a change to the same line as the token of interest, but to different tokens. In this example (from Linux commit 0f4630f3720e7e6e921bf525c8357fea7ef3dbab), the token `bgc` is removed from two lines, yet the line is reported within the “blame” output for the `gc`. After this change, all other tokens remaining in these two lines will be incorrectly attributed to this commit by `git-blame`.

This is the most common reason why blame-by-line failed (almost 82% of failures). This situation arises when the line containing the token of interest

is modified, yet the token of interest remains untouched. Blame-by-line will incorrectly attribute the token of interest to the latter commit, not to the one that originally added the token. This scenario is exemplified in Figure 15 (an excerpt of commit 0f4630f3720e7e6e921bf525c8357fea7ef3dbab in Linux). Note that this commit removes one token from each line (`bgc`) and all the other tokens in these two lines remain unchanged. Blame-by-line will incorrectly attribute any of these remaining tokens in these two lines to this commit.

5.4.3 Whitespace was modified after token is inserted

<pre> int fact(int i) { - if (i == 0) - return 1; - else - return i*fact(i-1); } int main(void) { - printf("Fact: %d\n", - fact(10)); - printf("Sum: %d\n", - sum(10)); - return 0; } </pre>	<pre> 8 int fact(int i) 9 { 10 + if (i == 0) return 1; 11 + else return i*fact(i-1); 12 } 13 int main(void) 14 { 15 + printf("Fact: %d\n", fact(10)); 16 + printf("Sum: %d\n", sum(10)); 17 return 0; 18 } </pre>
--	--

Fig. 16 Example of changes in whitespace that are not detectable by `git-blame -w` (ignore whitespace) because the change splits the line into two. In this example (from Linux commit c151aed6aa146e9587590051aba9da68b9370f9b), the changed line has been split into two, but the code (all the tokens) remains the same. `git-blame` does not handle whitespace changes that involve merging two or more lines into one line either. After this change, all tokens in the original line will be incorrectly attributed to this commit by `git-blame`, even when asked to ignore whitespace.

Whitespace changes are widely known as a major reason why `git-blame` might be incorrect. For this reason, `git-blame` is capable of ignoring whitespace, but only if the change of whitespace is within the line itself. It is not capable of detecting changes in whitespace when a line is split or two or more lines are merged. We decided to run `git-blame` without this option to evaluate the impact of whitespace in blame-by-line, and because Github’s blame view (which we used to manually analyze changes) does not support it. 11% of the failures of blame-by-line are attributed to this problem.

We also evaluated how many of these failures would exist had we run `git-blame -w`. The results are shown in Table 5. In total, 8 “blame” failures still would not have been avoided (20%). This means that the 32 failures that would have been avoided using `git-blame -w` would have improved the accuracy of blame-by-line from 80.6% to 82.2%. This low increase is due to the fact that none of these projects had massive reformatting of the source

Type	Linux	Lucene-solr	Guava	Git	Elastic-search	Total
Line split	1	0	4	1	1	7
Lines merged	0	0	1	0	0	1
Total	1	0	5	1	1	8

Table 5 Whitespace modifications that would not have been detected by `git-blame -w`. They represent 20% of all “Whitespace is modified” failures of Blame-by-Line.

code. Still, in practice we strongly recommend to always run `git-blame` with the “ignore whitespace” option.

5.4.4 File rename is not detected

By default, `git-blame` is capable of identifying if a file is renamed during a commit. There are two scenarios for this: first, the file was explicitly renamed with `git mv`; otherwise, `git-blame` uses a simple heuristic: if a file is removed and another is added, and the proportion of common lines between both files is above 50% (the default in `git`) then it is considered that the file was renamed.

There are only a handful of failures (1% of all) that were attributed to `git-blame` not being capable of identifying a rename (3 instances, all in Elastic-search). For example, commit `add18a5c99d` in Elastic-search removed the file `index/query/BaseQueryBuilder.java` and added the file `action/support-ToXContentToBytes.java`. The similarity of these two files was 49% percent (in the blame-by-line repository), just below the default threshold needed to detect the rename; in the token version of the file, the similarity was 67%, thus blame-by-token correctly identified the rename.

We verified the contents of the files and determined that this was indeed a file rename; in this case, the file was small, many lines with comments were added to the new file, and some identifiers and logic were changed. Blame per token was more resilient because each line, on average, had 7 tokens. Hence, a change in one comment or in one identifier had a smaller impact on the similarity of a tokenized file than its original source code.

5.4.5 Summary

The main reasons why blame-by-line failed in this study are well known: the line is modified after the token of interest is added, or the line suffered changes in whitespace. These two reasons accounted for 92.8% of the failures of blame-by-line in our study. If `git-blame` is run with the option to ignore whitespace, the number of failures would have been reduced by 8% on average). As it can be seen in Table 4, the distribution of each reason is fairly uniform across all projects.

On the other hand, the major reason for the failure of blame-by-token was the inability of “diff” to correctly align the changes where the developer most likely modified the file. Blame-by-line was also affected by this problem, but only one-third of the times compared to blame-by-token. This is mainly due to the much larger number of “lines” in the *repoView*.

Reason	Count	Ratio
Adding code in-between or in a new line	40	51.2%
Token reuse	38	48.7%

Table 6 Reasons why identifying the origin of a token yielded ambiguous results. Percentages are relative to the number of ambiguous cases. In this case, either method identified a commit that can be argued to be the origin of the specific token.

In our study, further changes to other tokens in the same line are the main reason why blame-by-line failed; whitespace changes contributed only 10% of its failures (80% of those failures—8% overall—would have been avoided running “blame” with the “ignore whitespace” option). The misalignment of changes in the output of “diff” was the only reason why blame-by-token failed; this reason was also present in blame-by-line, but to a lesser extent.

5.5 RQ3. Why can the origin of a token be ambiguous?

In a median of 2.6% of the tokens (see Table 2 for the results per project), we discovered that each method identified a different commit, but both commits could be considered to be the source of the token. We classified the observed cases into two further categories, which we explain below. Their frequency is shown in Table 6. The first two categories, *Token reuse* and *Adding code in-between or after*, contributed each approximately 50% of the instances.

5.5.1 Token reuse

Frequently, a token was reused from a previous statement that was removed. For example, let us assume that the line:

```
int a = size(PI);
```

... is replaced with:

```
y = distance(PI * x);
```

In our analysis, we considered each token to be independent from each other. Thus, in this example we will consider that the tokens “=”, “(”, *PI*, “)” and “;” were reused when the new statement replaced the old one. There are two potential reasons for this change: one is that the line was completely replaced, and some tokens are common by accident. Another reason is that the line of code was improved (it is still an assignment that involves a call to a function that uses *PI* as a parameter). This scenario brings interesting questions: when is a new line of code: a) an unrelated replacement for another line of code?; or b) derived from the line of code it replaces? We discuss this issue further in Section 6.

<code>#include <string></code>	<code>#include <iostream></code> <code>#include <string></code>
--------------------------------------	--

Fig. 17 A commit that changed the code from left to right could be performed in many different ways. The most natural is to insert the first line, but it is also possible to split the original line in the middle, leaving part of the original line in the first line, and part in the second line.

5.5.2 Adding code in-between-lines or in a new line

This scenario is a special case of `diff` has incorrect change alignment and occurs when the start of code that is inserted is identical to the start of the code right before the location of insertion (and when the end of the code being inserted is identical to the end of the code after the location of insertion). Figure 17 illustrates this scenario in a case where we want to add a new include statement (`#include <iostream>`). The code on the left hand side represents the code before the commit, and the code on the right hand side the code after the commit. In this example, there are at least two potential ways this change can be performed:

1. adding `#include <iostream>` *newline* before `#include <string>`, or
2. adding `<iostream>` *newline* `#include` after `#include`

The first is the most likely (and natural) scenario and the way blame-by-line interprets this change. However, it can also be argued that the commit could have followed the second scenario, splitting the original line into two; this is the interpretation of blame-by-token. In every instance of this issue, blame-by-line provided a more natural interpretation for these changes than blame-by-token.

5.5.3 Summary

We found that between 2.6% and 7% of tokens in the projects in our study had an ambiguous commit that inserted them. The reasons were two-fold (with almost the same frequency): adding code in-between a line or in a new line; and reusing tokens from a previous statement.

5.6 Threats to Validity

The main purpose of this study is two-fold: first, to demonstrate that the concept of *repoView* as prototyped by the `cregit` tool for the purpose of blame-by-token is effective. Second, to quantify the effect of blame-by-token compared to blame-by-line.

With respect to construct validity, we have released `cregit` as an open source tool⁵; anybody can inspect its implementation. With respect to internal

⁵ <http://turingmachine.org/2018/cregit>

validity, the main issue of this study is that the classification was made by one author. To minimize this threat, we are making available the dataset used in this study, including the classification made, so others have the opportunity to replicate these results⁶.

Most lines of code in a software system are not modified, and the majority of files in a system have very few modifications. In Linux 4.11, out of 42,934 C source files, 7,152 (17%) have never been modified, and the median number of commits per files is 7, while the average is 25.2; some files become hotspots of modifications. For example, in Linux 4.11 the file `drivers/.../intel_display.c` has 12,631 SLOCs (plus lines of comments) and has had 3,309 different commits (for an average of one commit every 3.8 SLOCs). According to blame-by-line, it contains source code from 1,555 different commits, while blame-by-token identifies 1,774 commits contributing to it. Providing an accurate blame of heavily modified files (whether per line or per token) is much more difficult than of a file that has been rarely modified. The sampling in our empirical study does not consider this imbalance, hence it is possible that the results would be different if we concentrated only in areas of the source code that have been more frequently modified.

With respect to external validity, we only studied two projects in C and three in Java in order to make the manual analysis manageable. We do not make any assertion regarding the accuracy of blame-by-token in general. Our main assertion is that blame-by-line can be improved with blame-by-token. The accuracy of blame-by-line is affected by some practices of the developers (e.g., reformatting code frequently) and stage of the project (code that is heavily maintained versus code that is relatively new); thus, we expect the accuracy of blame-by-line to vary widely between different projects, and the benefits that blame-by-token can provide to a given project to vary accordingly.

6 Discussion

6.1 Combining both methods

Blame-by-token is not perfect. In our study, its accuracy varied between 94.5% and 99.2% ($\pm 5\%$ with 95% confidence). However, using both methods could be a significant improvement over using blame-by-line only, because most of the times when blame-by-token was incorrect (or ambiguous), blame-by-line was able to properly identify the commit that inserted a given token.

Hence, further research is needed to combine both methods. First, one should automatically identify when blame-by-token is incorrect, then use blame-by-line to try to find the correct commit. This seems feasible especially because blame-by-token typically is incorrect due to wrong alignment of its “diff”, which might be a detectable situation. The data in our replication package might be useful for this purpose.

⁶ <http://github.com/cregit/evaluation>

```

4 config.c
View file

@@ -1096,7 +1096,7 @@ int git_config_from_file(config_fn_t fn, const char *filename, void *data)
1096     return ret;
1097 }
1098
1099 -int git_config_from_buf(config_fn_t fn, const char *name, const char *buf,
1100     size_t len, void *data)
1101 {
1102     struct config_source top;
1103     @@ -1132,7 +1132,7 @@ static int git_config_from_blob_shai(config_fn_t fn,
1132     return error("reference '%s' does not point to a blob",
1133     name);
1134 }
1135 -ret = git_config_from_buf(fn, name, buf, size, data);
1136 free(buf);
1137 return ret;
1138
1139 +int git_config_from_mem(config_fn_t fn, const char *name, const char *buf,
1140     size_t len, void *data)
1141 {
1142     struct config_source top;
1143     return error("reference '%s' does not point to a blob",
1144     name);
1145 }
1146 +ret = git_config_from_mem(fn, name, buf, size, data);
1147 free(buf);
1148 return ret;

```

Fig. 18 Example of the diff of a rename of a function in the line-based repository.

6.2 When code is refactored, copied and or moved

Identifying whether and how code has been refactored in a commit is not trivial [68], especially since such refactorings require to identify a mixture of code removal, alteration, and reinsertion. We believe that diffs at the token level, which include both token value and its type, might provide a new perspective on diffs (enhancing other methods such as structural diffs), helping to improve methods to detect refactoring, code movement and renaming of entities. A commit diff in a token-view repo created with our method can complement a tree-based diff created using a technique like gumtree⁷.

Figure 18 shows the diff of a commit that renamed one identifier in the original repo. As can be seen, the diff correctly shows that the two lines changed. Figure 19 shows the same change in the token-view repo: it is easy to spot that a function renaming happened (the declaration of the old function—the DECL marker—is replaced with a new declaration). To some extent, this information can be inferred from the line diff, but would require further processing.

The declaration repository view (which tracks only global declarations in files) is also useful to easily identify changes in the structure of the code. Figure 20 shows an example of a commit (5088d3b38775f8ac1... in the git project) that has incorrect alignment (both, token and line blame get this change wrong due to incorrect diff alignment); this commit looks like it removed the function `transport_check_allowed` and replaced it with `protocol_whitelist` (among many other changes). However, when inspected in the declaration view (shown in Figure 21), it is easy to observe that the commit added 3 functions and did not remove `transport_check_allowed`. This information is available in the diff in the original repository (line-based) but it would need to be parsed; the declaration provides this information without the need of any further processing.

Another advantage of the declaration view is the ability to use blame (a “declaration-based blame”) that can tell us the commit that was responsible for adding any declaration. Figure 22 shows an excerpt of the declaration-

⁷ <https://github.com/GumTreeDiff/gumtree>

6 ■■■ config.c			
@@ -5516,9 +5516,9 @@ block }			
5516	end_function	5516	end_function
5517		5517	
5518	begin_function	5518	begin_function
5519	-DECL function git_config_from_buf	5519	+DECL function git_config_from_mem
5520	name int	5520	name int
5521	-name git_config_from_buf	5521	+name git_config_from_mem
5522	parameter_list	5522	parameter_list
5523	name config_fn_t	5523	name config_fn_t
5524	name fn	5524	name fn
@@ -5714,7 +5714,7 @@ return ;			
5714	block }	5714	block }
5715	name ret	5715	name ret
5716	operator =	5716	operator =
5717	-name git_config_from_buf	5717	+name git_config_from_mem
5718	argument_list	5718	argument_list
5719	name fn	5719	name fn
5720	argument_list ,	5720	argument_list ,

Fig. 19 Example of the diff of the same rename in Figure 18 but in the token-view repository. It shows which specific change corresponds to the declaration of the function; the other is its use.

based blame of the file `branch.c` of the git project (we revisit this use below, in subsection 6.5). This view, in combination with the original line-based and the token-based repos could be used to identify refactorings: a significant removal and re-addition of similar lines/tokens results in the addition (and potential removal) of several declarations.

`cregit` could also be used for the detection of type-2 clones. For example, it is easy to create a view that only tracks the type of token (dropping its actual value—we will refer to it as the *token-type-only* view). This way we can simply compare sequences of strings in the code as if they were type-1 clones (identical substrings). This can be done both at the view level (the contents of the files in the view repository) or in the diffs of the commits. `git` is capable of identifying copying (if explicitly asked, and allows the user to specify the minimum length of the copy), which can be used on the *type-of-token-only* view to locate type-2 clones. However, this method has the potential to add a lot of noise (false positives), such as common idioms. Hence, it is important to study and evaluate what should be the minimal size of a match in order to maximize useful information.

Overall, a token-view repository of a project can be used to study code cloning and refactoring in an easier way than the original repository. We have made available the token view repositories in this study as part of our replication package to encourage others to look into these problems.

<pre> 912 -void transport_check_allowed(const char *type) 913 { 914 - struct string_list allowed = STRING_LIST_INIT_DUP; 915 - const char *v = getenv("GIT_ALLOW_PROTOCOL"); 916 917 - if (!v) 918 - return; 919 920 - string_list_split(&allowed, v, ':', -1); 921 - if (!string_list_has_string(&allowed, 922 type)) 923 924 die("transport '%s' not allowed", type); </pre>	<pre> 912 +static const struct string_list *protocol_whitelist(void) 913 { 914 + static int enabled = -1; 915 + static struct string_list allowed = 916 + STRING_LIST_INIT_DUP; 917 + if (enabled < 0) { 918 + const char *v = 919 + getenv("GIT_ALLOW_PROTOCOL"); 920 + if (v) { 921 + string_list_split(&allowed, v, 922 + ':', -1); 923 + string_list_sort(&allowed); 924 + enabled = 1; 925 + } else { 926 + enabled = 0; 927 + } 928 + } 929 + return enabled ? &allowed : NULL; 930 +} 931 +int is_transport_allowed(const char *type) 932 +{ 933 + const struct string_list *allowed = 934 + protocol_whitelist(); 935 + return !allowed string_list_has_string(allowed, 936 + type); 937 +} 938 +void transport_check_allowed(const char *type) 939 +{ 940 + if (!is_transport_allowed(type)) 941 + die("transport '%s' not allowed", type); </pre>
--	---

Fig. 20 Excerpt of a diff a that adds three functions. The diff has incorrect alignment, making it difficult to read the change. It appears as if the function `transport_check_allowed` was replaced.

6.3 Blame, authorship and copyright

Blame (whether by line or by token) simply points to the commit that inserted a given line or token. It does not determine who authored this code nor who its copyright owner is. We believe `cregit` is an improvement over blame-by-line, as it is more accurate at identifying the commit that added specific tokens. That said, this commit might have simply copied the code from another repository, or another part of the code, or refactored it (see above). Thus, blame-by-token is a tool that can help those assessing authorship and copyright, not a full solution to the problem.

Once a commit is identified as responsible for a given token sequence, it is necessary to study the actual change, its context, and other information not available in the repository. Perhaps an email explains better what this commit did, or perhaps the code is copied from another source, such as StackOverflow. Finding the true origin of the source code, whether internal to the project (i.e., refactorings, code movements and copies) or external (copied or imported code from an external source) is a major research problem that needs further study.

3 transport.c		View file
@@ -14,9 +14,11 @@ DECL function get_refs_via_rsync		
14	DECL function git_transport_push	14 DECL function git_transport_push
15	DECL function insert_packed_refs	15 DECL function insert_packed_refs
16	DECL function is_file	16 DECL function is_file
		17 +DECL function is_transport_allowed
17	DECL function print_ok_ref_status	18 DECL function print_ok_ref_status
18	DECL function print_one_push_status	19 DECL function print_one_push_status
19	DECL function print_ref_status	20 DECL function print_ref_status
		21 +DECL function protocol_whitelist
20	DECL function push_had_errors	22 DECL function push_had_errors
21	DECL function read_loose_refs	23 DECL function read_loose_refs
22	DECL function refs_from_alternate_cb	24 DECL function refs_from_alternate_cb
@@ -37,6 +39,7 @@ DECL function transport_get_remote_refs		
37	DECL function transport_print_push_status	39 DECL function transport_print_push_status
38	DECL function transport_push	40 DECL function transport_push
39	DECL function transport_refs_pushed	41 DECL function transport_refs_pushed
		42 +DECL function transport_restrict_protocols
40	DECL function transport_set_option	43 DECL function transport_set_option
41	DECL function transport_set_verbosity	44 DECL function transport_set_verbosity
42	DECL function transport_take_over	45 DECL function transport_take_over

Fig. 21 The diff of the same commit in Figure 20 but using the declaration view. It shows that only 3 functions were added and none were removed.

git-decl / branch.c		Newer
100644 20 lines (19 sloc) 567 Bytes		Raw Normal view
branch.c: Validate tracking branches with refsspecs instead ...	6 years ago	1 DECL function check_tracking_branch
Move create_branch into a library file	11 years ago	2 DECL function create_branch
branch: publish die_if_checked_out()	4 years ago	3 DECL function die_if_checked_out
Move create_branch into a library file	11 years ago	4 DECL function find_tracked_branch
Make git-clone respect branch.autosetuprebase	10 years ago	5 DECL function install_branch_config
branch: add read_branch_desc() helper function	7 years ago	6 DECL function read_branch_desc
Move code to clean up after a branch change to branch.c	11 years ago	7 DECL function remove_branch_state
Move create_branch into a library file	11 years ago	8 DECL function setup_tracking
Allow tracking branches to set up rebase by default.	11 years ago	9 DECL function should_setup_rebase
Prevent force-updating of the current branch	8 years ago	10 DECL function validate_new_branchname
branch.c: Validate tracking branches with refsspecs instead ...	6 years ago	11 DECL function validate_remote_tracking_branch
Move create_branch into a library file	11 years ago	12 DECL member matches
		13 DECL member remote
		14 DECL member spec
		15 DECL member src
		16 DECL struct tracking
branch: give advice when tracking start-point is missing	6 years ago	17 DECL variable upstream_advice
branch: improve error message for missing --set-upstream-...	6 years ago	18 DECL variable upstream_missing

Fig. 22 Excerpt of “Declaration-based” blame of the file `branch.c` of the git project.

6.4 When is code derived from other code?

In Section 5.5.1, we discussed the situation where code replaces other code and it becomes unclear whether the new code is derived from the previous code or is new. For instance, if a function replaces another one, are the braces from the old function in fact reused in the new function? If the answer is yes, this could imply that the new function shares structural information with the

one removed; if the answer is no, it would imply that the structure of the new code bears no relationship with the previous one. This might have copyright implications: by sharing its structure, the new function might be considered a derivative work of the previous function, even if no other tokens (aside from semicolons) are reused. It is very likely that there is no simple answer to this question, and each case might have to be considered on its own merits.

6.5 Other Applications of *repoViews*

```
static void __unhash_process(struct task_struct *p, bool group_dead)
{
    nr_threads--;
    detach_pid(p, PIDTYPE_PID);
    if (group_dead) {
        detach_pid(p, PIDTYPE_PGID);
        detach_pid(p, PIDTYPE_SID);

        list_del_rcu(&p->tasks);
        list_del_init(&p->sibling);
        __this_cpu_dec(process_counts);
    }
    list_del_rcu(&p->thread_group);
    list_del_rcu(&p->thread_node);
}
```

Andrew Morton:2003-01-14 00:21:23:[PATCH]
 Create a per-cpu proces counter for /proc
 reporting:e5b36baf5307d3d0987080b5304dbc0199
 1324ae;b

Contributors

Person	Tokens	Prop	Commits	CommitProp
Oleg Nesterov	42	51.85%	6	50.00%
Ingo Molnar	17	20.99%	1	8.33%
Al Viro	15	18.52%	1	8.33%
Andrew Morton	5	6.17%	2	16.67%
Christoph Lameter	1	1.23%	1	8.33%
Eric W. Biedermann	1	1.23%	1	8.33%
Total	81	100.00%	12	100.00%

Fig. 23 Example of the HTML view created for the Linux Foundation showing the source code of the Linux Kernel (located at <http://cregit.linuxsources.org>), coloured by the author of each token. This view overlays the metadata of the commit responsible for inserting the token when the mouse is placed over a token, and it hyperlinks to the commit that added it to the kernel.

The original motivation of **cregit** was to help improve the provenance analysis of source code in the Linux kernel. In April 2017, we deployed **cregit** on servers of the Linux Foundation (<http://cregit.linuxsources.org>). Specifically, we created an HTML output where developers can easily inspect the source code, and, by clicking on a given token, be directed to the commit that is responsible for this token. Since then we have been generating this view for each release of Linux (from 4.7 to 4.20), with new releases being added as they are completed. In particular, the Linux Foundation was inter-

linux-tags / kernel / exit.c 












	100644	53 lines (52 sloc)	1.48 KB	
	[CVE-2009-0029] System call wrappers part 07	9 years ago		1 DECL function SYSCALL_DEFINE1
				2 DECL function SYSCALL_DEFINE1
	[CVE-2009-0029] System call wrappers part 08	9 years ago		3 DECL function SYSCALL_DEFINE3
	[CVE-2009-0029] System call wrappers part 07	9 years ago		4 DECL function SYSCALL_DEFINE4
	[CVE-2009-0029] System call wrappers part 08	9 years ago		5 DECL function SYSCALL_DEFINE5
	[PATCH] move __exit_signal() to kernel/exit.c	12 years ago		6 DECL function __exit_signal

Fig. 24 Excerpt of the output of `git-blame` of `kernel/exit.c` in the declarations view repository of Linux. This view shows only the declarations in the file and the commits that added them.

ested in knowing who the contributors of the Kernel were. For this reason, the source code is coloured according to the person authoring the commit. An example of this view is shown in Figure 23. We invite the reader to visit <https://cregit.linuxsources.org>.

The *Linux sources* repositories created with `cregit` have been well received by the Linux community. Shuah Khan, a developer at Samsung described how the `cregit`’s HTML view can be used to help in debugging the kernel⁸. Heather Meeker, an intellectual property lawyer, used `cregit` to inspect the source code contributed by a specific developer (Patrick McHardy) [41]. Two open source projects have asked us to create these views for their source code, one of them to help them quantify who contributes to their project, and the other to help identify source code contributed by a specific developer. During the first 5 months of 2018, this website had 3,600 different visitors. This evidence is by no means empirical, but demonstrates that `cregit` is being used today.

Micro-commits Using the tokenized view repository, it is straightforward to see, for every commit, the tokens that this commit adds or removes (using `git log -p` on the tokenized view repository). One aspect that has surprised us is the prevalence of very small commits to the source code. For example, in the Linux kernel, 4% of all commits remove one token and add one token (4% in Git, 3% in ElasticSearch, 6% in Lucene-solr, 7% in guava). Furthermore, 11% of commits remove at most 3 tokens and add at most 3 tokens to Linux (9% in Git, 7% in ElasticSearch, 12% in Lucene-solr and 16% in Guava). These commits might provide important insights on what typical bug fixes are, and could be used to improve methods to self-repair code and to improve defect prediction methods.

Higher-level Analysis of Source Code Evolution Instead of analyzing source code evolution at a finer granularity, one can design views that abstract away coding details. For example, to study the evolution of the APIs in a software

⁸ <https://blogs.s-osg.org/made-change-using-cregit-debugging/>

project, one could use the *declaration* view, which simply outputs, for each file, the name (and type) of all global identifiers, in lexicographical order. The resulting view only contains declarations of global variables and methods, as is illustrated in Figure 11 for our running example. It is easy to see how nothing changed at the API level between revisions 1 and 2 of the *repoView* (and hence of the original repository), while one method declaration was added in the third revision. Figure 24 shows an excerpt of the output of `git-blame` of a file in the declarations view of the Linux repository. As it can be seen, this view only shows global declarations and the commits that added them.

7 Conclusions

In this paper, we have presented a novel way to increase the granularity of blame information in a version control system. This method relies on the creation of a “view” repository that has the same commits as the original repository but tracks a view of the source code. By breaking the source code such that each token is mapped to a different line, we are able to increase the fidelity of blame information from lines of code to tokens. We have implemented this method in a tool call `cregit`, and released it as open source.

To demonstrate the usefulness of our approach, we conducted an empirical study that evaluates the accuracy of blame when tracking the source code of a system (by line of code) versus tracking each token independently. In this study, conducted in five mature open source projects we found that blame-by-token reports the correct commit that adds a given source code token between 94.5% and 99.2% of the times (± 5 with a confidence of 95%), while the traditional approach of blame-by-line reports the correct commit that adds a given token between 74.8% and 90.9% (± 5 with a confidence of 95%).

Furthermore, we analyzed the reasons each method is incorrect: blame-by-line is usually wrong because a line was modified somewhere else after the token of interest is inserted and in 10% of cases, because the whitespace of the line was modified; in contrast, blame-by-token is wrong because diff incorrectly aligns changes. We believe that blame-by-token can be used to complement the use of blame-by-line, and help in understanding how commits change and evolve a software system.

`cregit` is open source⁹ and is currently being deployed by the Linux Foundation to help developers inspect the origin of its source code¹⁰.

References

1. John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, pages 361–370, New York, NY, USA, 2006. ACM.

⁹ <http://github.com/cregit>

¹⁰ <http://cregit.linuxsources.org>

2. Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta. Lhdiff: A language-independent hybrid approach for tracking source code lines. In *ICSM*, pages 230–239. IEEE Computer Society, 2013.
3. Dimitar Asenov, Balz Guenat, Peter Müller, and Martin Otth. Precise version control of trees with line-based version control systems. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering*, pages 152–169, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
4. Pablo Neira Ayuso. Frequently asked questions regarding gpl compliance and netfilter. <http://www.netfilter.org/licensing.html#faq>, 2017.
5. P. Bhattacharya, I. Neamtiu, and M. Faloutsos. Determining developers’ expertise and role: A graph hierarchy-based approach. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 11–20, Sept 2014.
6. Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, June 2005.
7. Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 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*, ESEC/FSE ’11, pages 4–14, New York, NY, USA, 2011. ACM.
8. G. Canfora, L. Cerulo, and M. Di Penta. Tracking your changes: A language-independent approach. *IEEE Software*, 26(1):50–57, Jan 2009.
9. Scott Chacon and Ben Straub. *Pro Git*. APres, 2nd edition, 2014.
10. Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkely, CA, USA, 2nd edition, 2014.
11. Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’96, pages 493–504, New York, NY, USA, 1996. ACM.
12. W. G. Cochran. *Sampling Techniques*. John Wiley and Sons, Inc., New York, 2nd edition, 1963.
13. M. L. Collard, M. J. Decker, and J. I. Maletic. Lightweight transformation and fact extraction with the srcml toolkit. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, pages 173–184, Sept 2011.
14. M. L. Collard, M. J. Decker, and J. I. Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*, pages 516–519, Sept 2013.
15. Julius Davies, Daniel M. German, Michael W. Godfrey, and Abram Hindle. Software bertillonage: Finding the provenance of an entity. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR ’11, pages 183–192, New York, NY, USA, 2011. ACM.
16. Georg Dotzler and Michael Philippsen. Move-optimized source code tree differencing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 660–671, New York, NY, USA, 2016. ACM.
17. Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.
18. Michael D. Feist, Eddie Antonio Santos, Ian Watts, and Abram Hindle. Visualizing project evolution through abstract syntax tree analysis. In *2016 IEEE Working Conference on Software Visualization, VISSOFT 2016, Raleigh, NC, USA, October 3-4, 2016*, pages 11–20, 2016.
19. Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, November 2007.
20. Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a programmer’s activity indicate knowledge of code? In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE ’07, pages 341–350, New York, NY, USA, 2007. ACM.

21. Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, pages 385–394, New York, NY, USA, 2010. ACM.
22. Daniel M. German. A study of the contributors of postgresql. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06*, pages 163–164, 2006.
23. Daniel M. German, Ahmed E. Hassan, and Gregorio Robles. Change impact graphs: Determining the impact of prior codechanges. *Information & Software Technology*, 51(10):1394–1408, 2009.
24. T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Eighth International Workshop on Principles of Software Evolution (IW-PSE'05)*, pages 113–122, Sept 2005.
25. M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, Feb 2005.
26. Masatomo Hashimoto and Akira Mori. Diff/ts: A tool for fine-grained structural change analysis. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering, WCRE '08*, pages 279–288, Washington, DC, USA, 2008. IEEE Computer Society.
27. A. E. Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 78–88, May 2009.
28. Ahmed E. Hassan and Richard C. Holt. C-REX: An Evolutionary Code Extractor for C - (PDF). Technical report, University of Waterloo, 2004. <http://plg.uwaterloo.ca/aee-hassa/home/pubs/crex.pdf>.
29. H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 200–210, June 2012.
30. Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 200–210, Piscataway, NJ, USA, 2012. IEEE Press.
31. Lile Palma Hattori, Michele Lanza, and Romain Robbes. Refining code ownership with synchronous changes. *Empirical Softw. Engg.*, 17(4-5):467–499, August 2012.
32. Yoshiki Higo, Akio Ohtani, and Shinji Kusumoto. Generating simpler ast edit scripts by considering copy-and-paste. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 532–542, Piscataway, NJ, USA, 2017. IEEE Press.
33. Akinori Ihara, Yasutaka Kamei, Masao Ohira, Ahmed E. Hassan, Naoyasu Ubayashi, and Ken-ichi Matsumoto. Early identification of future committers in open source software projects. In *Proceedings of the 2014 14th International Conference on Quality Software, QSIC '14*, pages 47–56, Washington, DC, USA, 2014. IEEE Computer Society.
34. David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 351–360, New York, NY, USA, 2011. ACM.
35. Shuah Khan. Who made that change and when: Using cregit for debugging. <http://www.gonehiking.org/ShuahLinuxBlogs/blog/2018/10/18/who-made-that-change-and-when-using-cregit-for-debugging/>, Oct 2018.
36. Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.
37. Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.
38. D. Ma, D. Schuler, T. Zimmermann, and J. Sillito. Expert recommendation with usage expertise. In *2009 IEEE International Conference on Software Maintenance*, pages 535–538, Sept 2009.
39. Christian Macho, Shane McIntosh, and Martin Pinzger. Extracting build changes with builddiff. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17*, pages 368–378, Piscataway, NJ, USA, 2017. IEEE Press.

40. David W. McDonald and Mark S. Ackerman. Expertise recommender: A flexible recommendation system and architecture. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work, CSCW '00*, pages 231–240, New York, NY, USA, 2000. ACM.
41. Heather Meeker. Patrick mchardy and copyright profiteering. Open Source, <https://opensource.com/article/17/8/patrick-mchardy-and-copyright-profiteering>, Aug 2017.
42. Xiaozhu Meng, Barton P. Miller, William R. Williams, and Andrew R. Bernat. Mining software repositories for accurate authorship. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13*, pages 250–259, Washington, DC, USA, 2013. IEEE Computer Society.
43. Xiaozhu Meng, Barton P. Miller, William R. Williams, and Andrew R. Bernat. Mining software repositories for accurate authorship. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13*, pages 250–259, Washington, DC, USA, 2013. IEEE Computer Society.
44. Webb Miller and Eugene W. Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.
45. Shawn Minto and Gail C. Murphy. Recommending emergent teams. In *Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR '07*, pages 5–, Washington, DC, USA, 2007. IEEE Computer Society.
46. Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. Type-directed diffing of structured data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe 2017*, pages 2–15, New York, NY, USA, 2017. ACM.
47. Audris Mockus and James D. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering, ICSE '02*, pages 503–512, New York, NY, USA, 2002. ACM.
48. Eugene W. Myers. Ano(nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, Nov 1986.
49. Nicolas Palix, Jean-Rémy Falleri, and Julia Lawall. Improving pattern tracking with a language-aware tree differencing algorithm. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 43–52, 2015.
50. Katherine Panciera, Aaron Halfaker, and Loren Terveen. Wikipedians are born, not made: A study of power editors on wikipedia. In *Proceedings of the ACM 2009 International Conference on Supporting Group Work, GROUP '09*, pages 51–60, New York, NY, USA, 2009. ACM.
51. Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the 20th IEEE International Conference on Software Maintenance, ICSM '04*, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society.
52. Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 491–500, New York, NY, USA, 2011. ACM.
53. Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 491–500, New York, NY, USA, 2011. ACM.
54. Steven P. Reiss. Tracking source locations. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 11–20, New York, NY, USA, 2008. ACM.
55. Johannes Schindelin. [patch 0/3] teach git about the patience diff algorithm. <https://marc.info/?l=git&m=123082787502576&w=2>, Jan 2009.
56. David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08*, pages 121–124, New York, NY, USA, 2008. ACM.
57. Francisco Servant and James A. Jones. History slicing: Assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*, pages 43:1–43:11, New York, NY, USA, 2012. ACM.

58. Francisco Servant and James A. Jones. Fuzzy fine-grained code-history analysis. In *Proceedings of the 39th International Conference on Software Engineering, ICSE '17*, pages 746–757, Piscataway, NJ, USA, 2017. IEEE Press.
59. Simon Sharwood. Linux kernel community tries to castrate GPL copyright troll. The Register, https://www.theregister.co.uk/2017/10/18/linux_kernel_community_enforcement_statement/, Aug 2017.
60. Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: A study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11*, pages 300–310, New York, NY, USA, 2011. ACM.
61. J. Spacco and C. Williams. Lightweight techniques for tracking unique program statements. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 99–108, Sept 2009.
62. C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 812–823, May 2015.
63. The Linux Foundation. Linux foundation and free software foundation europe introduce resources to support open source software license identification and compliance. <https://www.linuxfoundation.org/press-release/2017/04/linux-foundation-and-free-software-foundation-europe-introduce-resources-to-support-open-source-software-license-ident>, Apr 2017.
64. Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Re-visiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 1039–1050, New York, NY, USA, 2016. ACM.
65. Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018*, 2018.
66. Michail Tsikerdekis. Persistent code contribution: a ranking algorithm for code contribution in crowdsourced software. *Empirical Software Engineering*, Nov 2017.
67. Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1):100 – 118, 1985. International Conference on Foundations of Computation Theory.
68. P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 231–240, Sept 2006.
69. Harald Welte. Report from the Geniatech vs. mchardy GPL violation court hearing. <http://laforge.gnumonks.org/blog/20180307-mchardy-gpl/>, March 2018.
70. Zhenchang Xing and Eleni Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05*, pages 54–65, New York, NY, USA, 2005. ACM.
71. Yunwen Ye and Kouichi Kishida. Toward an understanding of the motivation open source software developers. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 419–429, Washington, DC, USA, 2003. IEEE Computer Society.
72. Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. On the scalability of linux kernel maintainers' work. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, pages 27–37, New York, NY, USA, 2017. ACM.
73. Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. On the scalability of linux kernel maintainers' work. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, pages 27–37, New York, NY, USA, 2017. ACM.