

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

Daniel M. German · Bram Adams ·  
Kate Stewart

Draft: July 5, 2018

**Abstract** The blame feature of version control systems is widely used—both by practitioners and researchers—to determine who has modified a given line of code, and the commit where the contribution was made. The main disadvantage of blame is that, when a line is modified several times, it only shows the last person who 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-line is capable of properly identifying the commit that introduced with an accuracy between 94.5% and 99.2%, while blame-per-line can only achieve an accuracy between 75% and 91%. This 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 each token in the Linux kernel.

## 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 known as colloquially

---

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

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 [24, 26, 58, 56] (specifically, to identify the commit that inserted the bug, using—for example—the SZZ algorithm [33]), to recommend experts on a given area of the source code [41, 43, 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. [48, 49, 59, 67, 61, 60, 68, 30, 66, 5, 36, 41, 18, 52, 38, 33]).

One of the major disadvantages of blame is that it operates at the line level [39, 20, 54]. 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). This problem is exacerbated as the history of a project grows, and the code might be modified over and over again.

Another aspect that is gaining relevance is the need to identify who are the authors of the source code for legal purposes. 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 [37, 64]. 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—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 three quarters of a million changes to it.

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). Specifically, our contributions are:

- A method that, given a git repository, creates a new synthetic git repository that has the same commits and metadata than 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 ([github/cregit/](https://github.com/cregit/)) that is already being used to improve blame information at the token level in the Linux kernel.
- 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 as 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, blame-by-line is accurate between 75 and 91%, while blame-by-token is accurate between 95 and 99% of the time.
- A categorization of the reason why blame-by-line and blame-by-token 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

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 [20]).

### 2.1 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

|  |  |   |
|--|--|---|
| <pre> #include &lt;stdio.h&gt;  int sum(int i) {     int total = 0;     while (i--&gt;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 &lt;stdio.h&gt;  int sum(int i){     int total = 0;     while (i--&gt;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 &lt;stdio.h&gt;  long sum(long i) {     long total = 0;     while (i--&gt;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).

```

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).

white space-only changes to a line). She would then contact the resulting 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. Most recently, Meng et al. developed a git extension (`git-author`) [39] 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 incorreced attributed to *Dev C* and line 6 to all three developers.

Hence, both problems seem to have the same root cause: 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.

## 2.2 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 sueing distributors of products that contain the Linux kernel [55, 4, 64]. Meeker provides a detailed description of McHardy's actions [37]. 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.

```

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)`.

A major challenge in this case is to determine—using currently available tools—McHardy’s contributions to the kernel and 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.

`git-blame` can be used to point to the commit that contributed specific code to a project. It would still be necessary to analyze this commit to deter-

mine if who the true author of the code is. For example, the new code might be refactoring of old code, or might have been copied from an external source.

### 2.3 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 the function `fact` was added. `git-blame` (as shown in Figure 2) will incorrectly indicate that the second commit was the one that added this function.

### 2.4 Related Work

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

#### 2.4.1 Origin Analysis

Origin analysis focuses on tracking code functionality across multiple revisions of a code base [22,13]. 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 [9]. 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. [40,44] and Ukkonen et al. [62]. 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 [40]. 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 [50], 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 [10]. While a wide variety of tree-based approaches exist [3, 6, 14, 16, 23, 29, 42, 45, 47], even for other documents than source code [35, 65], the most commonly used approaches are ChangeDistiller [17] and GumTree [15].

ChangeDistiller [17] uses bigram string similarity to match AST nodes between file revisions, followed by similarity measures at the level of subtrees [10]. GumTree [15] 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. [27] 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 [25] and Kim et al.’s LSdiff [32] 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 [25] or logic rules [32] 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 [57] 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.

#### 2.4.2 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 Tsikrerdakis [61], 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 [46], 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. [53] (history slicing) and Meng et al. [39] 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. [61] 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` [39] 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 [36] recommends the developer who most recently committed to a file as expert for that file. This approach was improved by Mockus et al. [43], who consider the number of changes made by a developer, and by Girba et al. [21], 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. [18] 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 [52]. In later work [34], these authors empirically found that both usage and implementation expertise recommenders perform similarly. Fritz et al. [19] used this insight to create a degree of knowledge model to recommend experts by combining both usage and implementation metrics.

Finally, Hattori et al. [28] 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 are divisible and mergeable. This limitation becomes amplified when a project is composed of many different files modified a large number of times (e.g., 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 the largest a non-divisible unit of source code. Under this assumption, a token cannot be modified, it can only be removed or replaced

```

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.

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 accross 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 `git` repository that tracks the evolution of the source code with the same commits, but in which each file in a commit has been reformatted such that there is only one token per line. One can 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* with our running example. For this example, we have used a tokenizer filter to create “views”. This tokenizer converts

|                                  |                                  |                                  |
|----------------------------------|----------------------------------|----------------------------------|
| preprocessor #                   | preprocessor #                   | preprocessor #                   |
| directive include                | directive include                | directive include                |
| 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 <sub>1</sub> , token) | ... view(c <sub>2</sub> , token) | ... view(c <sub>3</sub> , 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 original source code file into a version in which each token occupies one line (we have used `srcml` [12,11] for this purpose). Note that each line corresponds to 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>`, 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 the 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 Formalization

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

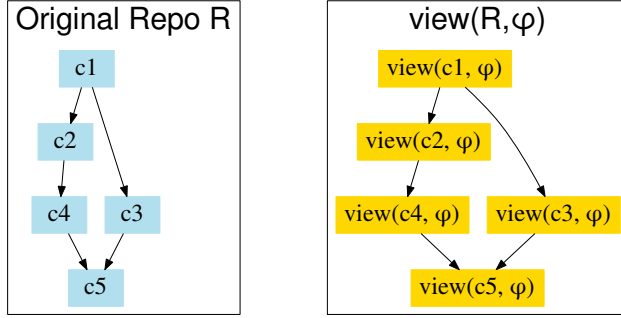
1. for any text file  $f$ ,  $\phi(f)$  is also a text file;

```

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.



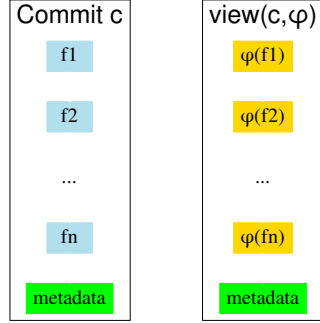
**Fig. 7** DAG of the original repository  $R$  and of its view according to mapping  $\phi$ . The two DAGs show the bijective nature of views.

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

We can easily extend the definition of view from one file to a set of files: the view of a set of files  $F$  using mapping  $\phi$ , denoted as  $\phi(F)$ , is the set of the views of each of the files in  $F$  using  $\phi$ .

Second, a version control repository  $R$  is composed of two parts: a set of commits  $C$  and a set of labels  $L$ . Each label is a pair  $\langle n, c \rangle$  where  $n$  is the name



**Fig. 8** Anatomy of a commit in terms of its file revisions and metadata, and its corresponding view according to the  $\phi$  mapping. The metadata of the commit remains unchanged through the mapping.

of the label and  $c$  its corresponding commit. A subset of these labels represents the current branches of the repository (for example, the commit corresponding to the label “master” will represent the current head of this branch).

Each commit  $c \in C$  is then a triplet  $\langle F, P, M \rangle$  consisting of the set of file revisions  $F$  in the repository after the commit  $c^1$ , the ordered list  $P$  of the commits’ parents (potentially empty) and the associated commit metadata  $M$  (such as its author, time of the commit, description, etc.). Hence, a repository is a collection of commits where each commit has one or more ancestors. This ancestor relationship is modelled as a directed acyclic graph (DAG), where commits are nodes and the ancestor relationship are edges (see left hand side of Figure 7).

Now, given a commit  $c = \langle F, \langle c_1, c_2, \dots, c_n \rangle, M \rangle$ , we define its view using the  $\phi$  mapping (denoted as  $view(c, \phi)$ ) as:

$$view(c, \phi) = \langle \phi(F), P', M \rangle$$

where

$$P' = \langle view(c_1, \phi), view(c_2, \phi), \dots, view(c_n, \phi) \rangle$$

In other words, the view of a commit  $c$  using mapping  $\phi$  is created by: 1) replacing each file  $f$  in the commit with  $\phi(f)$ ; and 2) by replacing each parent of the commit with its corresponding view. The metadata of  $c$  in the view is the same as in the original commit. This is exemplified on the right hand side of Figure 8. By extension, we can also define the view of a set of commits  $S$

<sup>1</sup> Some version control systems such as CVS and SVN track changes (deltas) to files, however, they can be modelled as if these systems track the new revisions instead (as `git` does).

using mapping  $\phi$  (denoted  $view(S, \phi)$ ) as the set of the views of each commit in  $S$  using  $\phi$ .

At the repository level, a *repoView* of a repository  $R = \langle C, L \rangle$  (composed of set of commits  $C$  and labels  $L$ ) using mapping  $\phi$ , denoted  $V(\langle C, L \rangle, \phi)$ , is:

$$V(\langle C, T \rangle, \phi) = \langle view(C, \phi), L' \rangle$$

Where  $L'$  corresponds to the set of labels in  $L$  where the commit  $c$  of each label has been replaced with  $view(c, \phi)$ . Hence, informally, the *repoView* of a repository is formed by replacing each of the commits in the original repository with their views (using the mapping  $\phi$ ). Effectively,  $V(\langle C, L \rangle, \phi)$  is a version control repository that tracks the evolution of the views of the files of the original repository, in a similar way that the original repository tracks the evolution of the original files.

### 3.3 Token view of a repository

Using the concept of *repoView*, a token view of a file is defined as a view of a source code repository where each token of each revision occupies one line. In other words, the token view of a source code repository is the view of  $R$  using *token* as  $\phi$  mapping:  $V(R, token)$ . By leveraging regular line-level “blame” on token-level views, a token-level “blame” can be easily obtained.

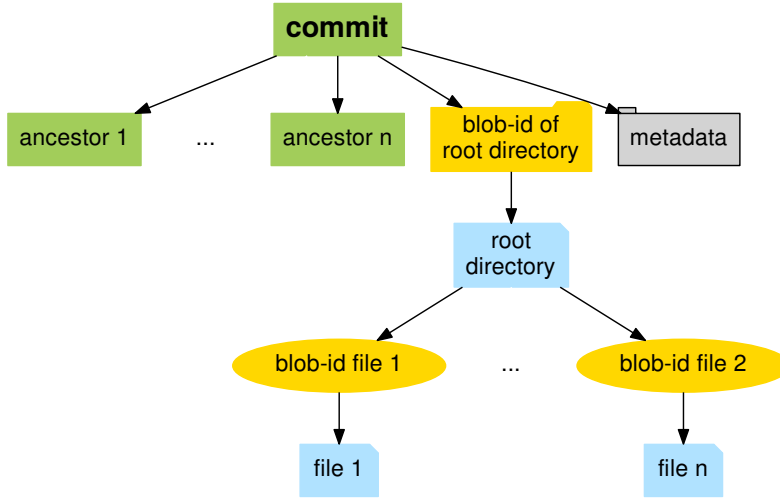
## 4 Implementation

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  $\phi$  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 a string to a given commit identifier (the tags). These labels correspond to a)



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

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  $\phi : \text{file} \rightarrow \text{file}$

**Result:**  
the *repoView*  $V$  of  $R$

**# Phase 1: create views of blobs**  
**for** each blob  $b \in R$  **do**  
| create a new blob  $b' = \text{view}(b, \phi)$ ;  
| 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  $c' = \text{view}(c, \phi)$  such that  
| metadata( $c'$ ) = metadata( $c$ ) + original commit-id;  
| **for** every blob  $b \in c$  **do**  
| |  $b' \leftarrow \text{view}(b, \phi)$ ;  
| | add  $b'$  to  $c'$ ;  
| **end**  
| **for** every  $i$ -th ancestor commit  $a_i$  of  $c$  **do**  
| | make  $\text{view}(a_i, \phi)$  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  $\text{view}(c_l, \phi)$   
**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*.

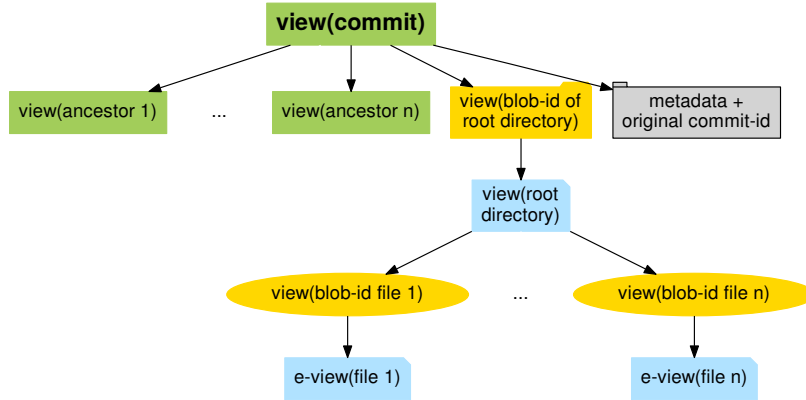
Our implementation of algorithm 1 is based on BFG Repo-Cleaner<sup>2</sup>. BFG Repo Cleaner is a tool used to replace strings in every blob in a repository and/or remove blobs in a git repository that contain certain strings. We have extended BFG with the ability to replace the blobs of each commit 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 mapping responsible for generating a view out of the contents of a file is a dispatcher program used by BFG to take the contents of the input file from standard input and output the view to standard output. So far, we have implemented two different views: *token* (which we have presented in section 3) and *declaration*. *declaration* is a filter that outputs the list of global identifiers (variables, constants, methods, classes, functions, etc) defined in the source

---

<sup>2</sup> <https://rtyley.github.io/bfg-repo-cleaner/>



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

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.

|  |  |  |
|--|--|--|
| <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, c<sub>1</sub>)</code> | <code>view(declarations, c<sub>2</sub>)</code> | <code>view(declarations, c<sub>3</sub>)</code> |

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

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 inspecting the *repoView* easier. They are markers inserted into the 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.

We have implemented the *token* filters for the following languages: C, C++, Java, Go and Python. For C/C++ and Java, we use `srcml` [12, 11] 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

```

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.

records, and we use exuberant ctags to insert the annotations to the output of the parsers.

Regarding 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).

## 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.

| 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.

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 398 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, our sample of 398 tokens per project provides a confidence interval of  $\pm 5\%$  for a confidence level of 95%. 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.

Specifically, for blame-by-line, we run **git-blame** on the corresponding file in the original git repository of the project and recorded the commit id responsible for modifying last the line where the token was located. For blame-by-token, we mapped the location of the token from the original repository to the location it occupied in the token-view repository (the tokenizer of **cregit** outputs file, line and column information). We then run **git-blame** on the token’s file in the token-view repository and recorded the commit responsible for modifying most recently the line corresponding to this token. Because this is a commit in the *repoView*, we also recorded the commit in the original repository that corresponded to this commit (every commit in the *repoView* has a corresponding commit in the original repository).

After this process was completed, we had, for each token:

- its location (file, column);
- the blame-by-line commit id; and
- the blame-by-token commit id.

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, hence both blame-by-token and blame-by-line are 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.

To determine which of the five outcomes applied for a given “blame” output, we manually inspected the history of the source code using `git` and GitHub in order to identify the actual commit that inserted the token. We also documented a brief explanation of the rationale for the classification, usually the explanation of why a method was wrong. We then grouped these explanations into a set of categories that explain each failure.

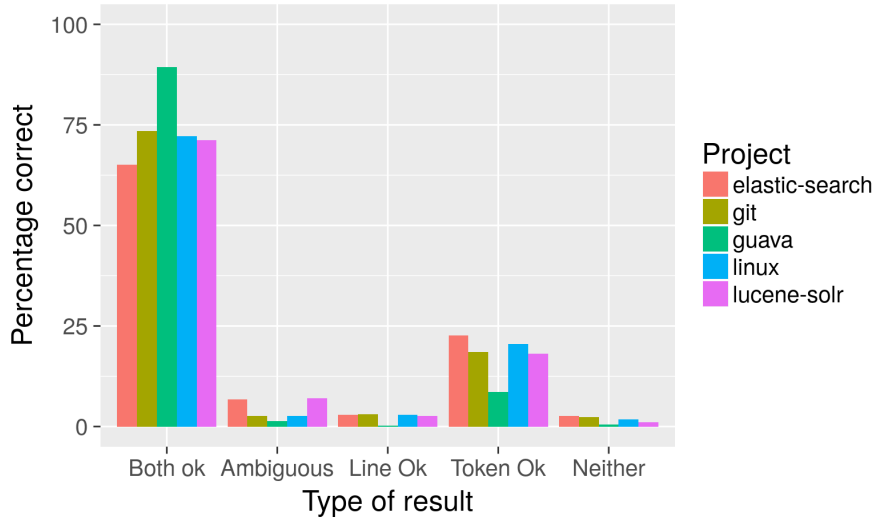
## 5.2 RQ1. Is blame-by-token more accurate than blame-by-line?

**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 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

| 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 398. The five categories are mutually exclusive.



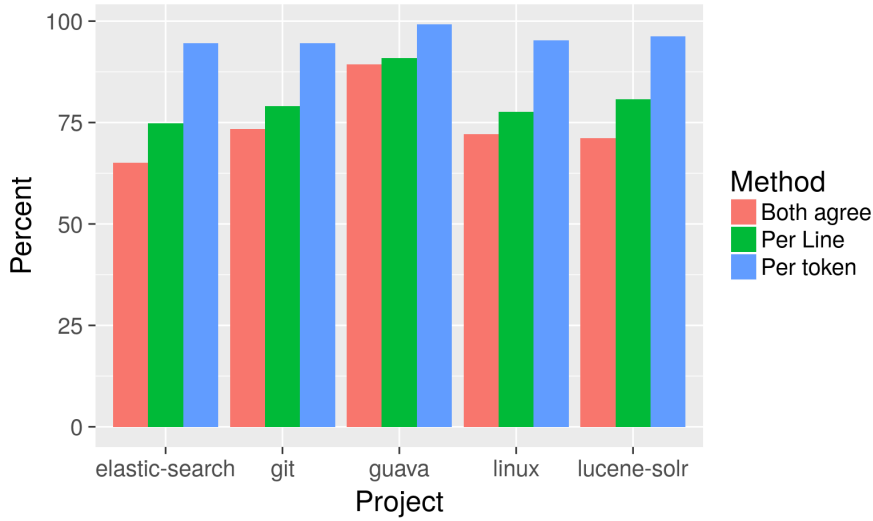
**Fig. 13** Breakdown of the accuracy of each blame method (see Table 2 for details).

| 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 398.

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 develop-



**Fig. 14** Results for the correctness of each method (Both Ok, Token Ok and Line Ok percentages of Table 3), for each project.

ment 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. We also observed that 43% of Guava’s source files have never been modified. In comparison, in `git` only 9% of source files were never modified, 6.5% of Linux, 21.8% of `elasticSearch`, and 25.9% of `Lucene-Solr`. The more files are modified, the more likely blame-by-token will be different than blame-by-line.

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 only were correctly classified 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).

### 5.3 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

| Method         | Reason  | Count | Ratio  |
|----------------|---|-------|--------|
| Blame-by-Token | <code>diff</code> has incorrect change alignment. | 77    | 100.0% |
| Blame-by-Line  | Line was modified after the token is inserted.    | 305   | 82.2%  |
|                | Whitespace was modified after token is inserted.  | 40    | 10.8%  |
|                | <code>diff</code> has incorrect change alignment. | 23    | 6.2%   |
|                | File rename not detected.                         | 3     | 0.8%   |

**Table 4** Reasons why identifying the origin of a token yielded wrong results. The percentages are relative to each blame approach’s incorrect classifications.

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

**Table 5** Breakdown per project regarding the reasons why identifying the origin of a token yielded wrong results.

why each method failed. This analysis resulted into four reasons. They are summarized in Table 4 and described below.

### 5.3.1 *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 2.4.1, `git`’s internal implementation of `diff` uses the Myer’s algorithm [44], which finds the minimum common subsequence between two strings (considering lines as indivisible). It is a greedy algorithm with as main advantages that it is fast, space efficient and minimizes the size of the patch (it runs in  $O(ND)$ , where  $N$  is the 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 changed<sup>3</sup>.

<sup>3</sup> 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 [51].





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. 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.

### 5.3.2 Line was modified after the token is inserted

|  |  |
|--|--|
| <pre> 398 @@ -398,9 +388,9 @@ static int brcmstb_gpio_irq_setup(struct platform_device *pdev, 398     if (priv-&gt;can_wake) 399         bank-&gt;irq_chip.irq_set_wake = 400             brcmstb_gpio_irq_set_wake; 401 -     gpiochip_irqchip_add(&amp;bank-&gt;bgc,gc, &amp;bank-&gt;irq_chip, 402         0, 403         handle_simple_irq, IRQ_TYPE_NONE); 404 -     gpiochip_set_chained_irqchip(&amp;bank-&gt;bgc,gc, &amp;bank-&gt; 405         &gt;irq_chip, 406         priv-&gt;parent_irq, 407         brcmstb_gpio_irq_handler); 408     return 0; </pre> | <pre> 388     if (priv-&gt;can_wake) 389         bank-&gt;irq_chip.irq_set_wake = 390             brcmstb_gpio_irq_set_wake; 391 +     gpiochip_irqchip_add(&amp;bank-&gt;gc, &amp;bank-&gt;irq_chip, 0, 392         handle_simple_irq, IRQ_TYPE_NONE); 393 +     gpiochip_set_chained_irqchip(&amp;bank-&gt;gc, &amp;bank-&gt; 394         &gt;irq_chip, 395         priv-&gt;parent_irq, 396         brcmstb_gpio_irq_handler); 397     return 0; </pre> |
|--|--|

**Fig. 16** 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 16 (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.

### 5.3.3 Whitespace was modified after token is inserted

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

|  |  |
|--|--|
| <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. 17** 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.

| 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     |
| <b>Total</b> | 1     | 0           | 5     | 1   | 1              | 8     |

**Table 6** 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.

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

#### 5.3.4 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

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

**Table 7** 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.

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.

### 5.3.5 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 5, the distribution of each reason is fairly uniform across all the 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.

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 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.

### 5.4 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 7. The first two categories, *Token reuse* and *Adding code in-between or after*, contributed each approximately 50% of the instances.

#### 5.4.1 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.

#### 5.4.2 Adding code in-between-lines or in a new line

|                   |  |                     |
|-------------------|--|---------------------|
| #include <string> |  | #include <iostream> |
|                   |  | #include <string>   |

**Fig. 18** 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.

This scenario 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 18 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, then there are at least two potential ways this change can be performed:

1. adding `#include <iostream>` before `#include <string>`
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.

### 5.4.3 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.

## 5.5 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 tool<sup>5</sup>; anybody can inspect its implementation. With respect to internal 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 results<sup>6</sup>.

With respect to external validity, we only studied two projects in C and three in Java in order to make the manual analysis of 398 sampled tokens per project, 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.

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

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

## 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%. However, it is a significant improvement over blame-by-line, 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 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.

### 6.2 When code is refactored, copied and or moved

Identifying whether and how code has been refactored in a commit is not trivial [63] 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 not only the token or its type) might provide a new perspective on diffs (enhancing other methods such as structural diffs), helping to improve methods to detect refactoring, code cloning and code movement. The syntactical diff (created with our method) can complement a tree-based diff

In fact, `git` is capable of identifying some types of code movement and copying (if explicitly asked). When doing such identification, the complexity of diff grows significantly. To detect code copying, for every hunk in the diff of each commit, `git` tries to match it to all the code already existing (line-by-line comparison). This could be extremely expensive for large software systems such as Linux, where running blame in this manner can literally take days on a single file. Detecting moved code is less expensive: `git` only compares the new code to code that has been removed in the same commit. In both cases, `git` is instructed about the minimum length that the match should have (the default is 20 characters for movements and 40 characters for copies).

We believe that, aside from the extra computational cost of asking blame to detect copies and movements, doing it on the token repository is likely to yield more useful results than in the original repository. However, this also has the potential to add a lot of noise (false positives), where common idioms are detected as copies or movements. Hence, it is important to study and evaluate what should be the minimal size of a match in order to maximize useful information.

Furthermore, the token view of a repository was inspired by the preprocessing stage of `ccfinder` [31] (see Figure 5). In a token-view repository, `git-log`

will output a diff of this token view, simplifying the process of finding code clones in the commit. 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.

### 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.

### 6.4 When is code derived from other code?

In Section 5.4.1, 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*

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



```
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. 19** 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.

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 4.16), with new releases being added as they are completed. In particular, the Linux Foundation was interested 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 19. 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 kernel<sup>7</sup>. Heather Meeker, an intellectual property lawyer, used **cregit** to inspect the source code contributed by a specific developer (Patrick McHardy) [37]. 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 the year this website had 3600 different visitors.

<sup>7</sup> <https://blogs.s-osg.org/made-change-using-cregit-debugging/>

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. 20** 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.

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); and 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 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 20 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, while the traditional approach of blame-by-line reports the correct commit that adds a given token between 74.8% and 90.9%. 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.

`cregit` is currently being deployed by the Linux Foundation to help developers inspect the origin of its source code<sup>8</sup>.

## 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.
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*. Apress, Berkely, CA, USA, 2nd edition, 2014.
10. Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of*

<sup>8</sup> [cregit.linuxsources.org](http://cregit.linuxsources.org)

- the 1996 ACM SIGMOD International Conference on Management of Data, SIGMOD '96, pages 493–504, New York, NY, USA, 1996. ACM.
11. 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.
  12. 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.
  13. 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.
  14. 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.
  15. 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.
  16. 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.
  17. 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.
  18. 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.
  19. 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.
  20. 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.
  21. 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.
  22. 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.
  23. 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.
  24. 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.
  25. 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>.
  26. 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.
  27. 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.
  28. Lile Palma Hattori, Michele Lanza, and Romain Robbes. Refining code ownership with synchronous changes. *Empirical Softw. Engg.*, 17(4-5):467–499, August 2012.

29. 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.
30. 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.
31. Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.
32. 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.
33. 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.
34. 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.
35. 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.
36. 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.
37. Heather Meeker. Patrick mchardy and copyright profiteering. Open Source, <https://opensource.com/article/17/8/patrick-mchardy-and-copyright-profiteering>, Aug 2017.
38. 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.
39. 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.
40. Webb Miller and Eugene W. Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.
41. 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.
42. 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.
43. 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.
44. Eugene W. Myers. Ano(nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, Nov 1986.
45. 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.

46. 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.
47. 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.
48. 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.
49. 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.
50. 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.
51. Johannes Schindelin. [patch 0/3] teach git about the patience diff algorithm. <https://marc.info/?l=git&m=123082787502576&w=2>, Jan 2009.
52. 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.
53. 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.
54. 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.
55. 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/](https://www.theregister.co.uk/2017/10/18/linux_kernel_community_enforcement_statement/), Aug 2017.
56. 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.
57. 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.
58. 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.
59. Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Revisiting 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.
60. 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.
61. Mikhail Tsikerdekis. Persistent code contribution: a ranking algorithm for code contribution in crowdsourced software. *Empirical Software Engineering*, Nov 2017.
62. Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1):100 – 118, 1985. International Conference on Foundations of Computation Theory.
63. 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.
64. Harald Welte. Report from the Geniatech vs. mchardy GPL violation court hearing. <http://laforge.gnumonks.org/blog/20180307-mchardy-gpl/>, March 2018.

- 
65. 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.
  66. 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.
  67. 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.
  68. 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.