2013 IEEE International Conference on Software Maintenance

Mining Software Repositories for Accurate Authorship

Xiaozhu Meng, Barton P. Miller, William R. Williams, and Andrew R. Bernat
Computer Sciences Department
University of Wisconsin
Madison, WI 53706 USA
{xmeng, bart, bill, bernat}@cs.wisc.edu

***Abstract*—Code authorship information is important for analyzing software quality, performing software forensics, and improving software maintenance. However, current tools assume that the last developer to change a line of code is its author regardless of all earlier changes. This approximation loses important information. We present two new line-level authorship models to overcome this limitation. We first define the *repository graph* as a graph abstraction for a code repository, in which nodes are the commits and edges represent the development dependencies. Then for each line of code, *structural authorship* is defined as a subgraph of the repository graph recording all commits that changed the line and the development dependencies between the commits; *weighted authorship* is defined as a vector of author contribution weights derived from the structural authorship of the line and based on a code change measure between commits, for example, best edit distance. We have implemented our two authorship models as a new git built-in tool *git-author*. We evaluated *git-author* in an empirical study and a comparison study. In the empirical study, we ran *git-author* on five open source projects and found that *git-author* can recover more information than a current tool (*git-blame*) for about 10% of lines. In the comparison study, we used *git-author* to build a line-level model for bug prediction. We compared our line-level model with a representative file-level model. The results show that our line-level model performs consistently better than the file-level model when evaluated on our data sets produced from the Apache HTTP server project.**

I. Introduction

Information as to who wrote a given piece of code, authorship, is used to analyze software quality [5, 11, 32, 35, 38], perform software forensics [33], and improve software maintenance [13, 14]. Current tools approximate line level authorship by assuming that the last person to change a line is its author, while ignoring all earlier changes. In this paper, we show how to mine a code repository for the development history of a line of code to assign contribution weights to multiple authors. Using these contribution weights, we can attribute a line to the most responsible author in binary code forensics, directly apply the weights to model source code familiarity, and trace back to earlier commits to determine when bugs were introduced in software quality analysis. Our new method abstracts code repositories as a graph representing the development dependencies between commits. We perform a backward flow analysis based on the results of an enhanced line differencing tool [8] between adjacent commits to extract the development history of a line of code. We then use the history to attribute each character of the line to the responsible author and assign contribution weights. We have implemented this new functionality as an extension to git.

The methods used by current tools (git-blame [11, 32], svn-annotate [38], and CVS-annotate [35]) for obtaining line level authorship loses information. A line of code may be changed multiple times by different developers to fix bugs, to conform to interface changes, or to tune parameters. These changes compose the history of a line of code. For each line of code, current tools report the last commit that changed the line and the author of that last commit. These tools take the last snapshot, while missing the earlier stages of the development history. Therefore, even when the last commit changes only a small fraction of a line of code, the author of the last commit still is credited for the entire line.

In this paper, we define the *repository graph*, *structural authorship*, and *weighted authorship* to help overcome these limitations. The repository graph is a directed graph representing our abstraction for a code repository. In the graph, nodes are the commits and edges represent the development dependencies. For each line of code, we define structural authorship and weighted authorship. Structural authorship is a subgraph of the repository graph. The nodes consist of the commits that changed that line. Development dependencies between the subset commits form the edges. Weighted authorship is a vector of author contribution weights derived from the structural authorship of the line. The weight of an author is defined by a code change measure between commits, for example, best edit distance [36]. We use these two models to extract the development history of a line of code and derive precise line level authorship.

To evaluate our new models, we implemented structural authorship and weighted authorship as a new git built-in tool: *git-author*. We conducted two experiments to show how often the new models will produce more information and whether this information is useful for analysis tools that are based on code authorship information. In the first experiment, we ran git-author over the repositories of five open source projects and found that about 10% of the lines were changed by multiple commits and about 8% of the lines were changed by multiple authors. Analysis tools lose information on these lines when they use the current methods for line level authorship. In the second experiment, we used *git-author* to build a new

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.36

line-level bug prediction model. We compared our line-level model with a representative file-level model [22] on our data sets derived from the Apache HTTP sever project [1]. The results show that the line-level model performs consistently better than the file-level model when evaluated on effort-aware metrics [22, 25].

This work makes the following contributions:

1) The structural authorship model that extracts the development history of a line of code and overcomes the fundamental weakness of current tools.
2) The weighted authorship model that assigns contribution weights to each change of the line and produces precise line-level authorship attribution.
3) The tool *git-author* that is a new built-in tool in git and implements the structural authorship and the weighted authorship model.
4) A study of five open source projects that characterizes the number of lines changed by multiple commits and multiple authors.
5) A line-level bug prediction model that performs consistently better than the file-level model [22].

We provide an overview of version control systems and define our graph abstraction for code repositories in Section 2. We present the structural authorship model in Section 3 and the weighted authorship model in Section 4. We evaluate our new models in Section 5. We discuss related work in Section 6 and then conclude in Section 7.

II. Repository Abstraction

We define the repository graph to capture the fundamental capability of a version control system (VCS). With the repository graph, we can focus on the contents of development history without considering which specific VCS is used. A VCS records the development history of a project by storing all the revisions of source code and the dependent relationship between these revisions. Our graph abstraction models revisions as nodes and the relationship between revisions as edges. We are able to implement the graph structure based on any current mainstream VCS.

A VCS allows programmers to checkpoint their changes. A new revision is created when a programmer commits their modifications to the VCS. The dependencies between revisions also is recorded to maintain the relative order of commits. Current VCS's support concurrent development. Programmers can work on different branches without affecting other people's work and later combine their work by merging branches. Therefore, it is also necessary to record on which existing revision the new revision is based. In addition to these basic capabilities, a VCS often supports reverting previous changes, browsing development history, and other complementary capabilities to facilitate daily development work.

The repository graph is a directed graph $G = (V, E, \Delta)$ used to describe the basic capability of a VCS. A node in V represents a revision or a snapshot of the project and is annotated with information about the snapshot including the author of the snapshot. The snapshot of node i is denoted as s_i, $s_i \in V$, and the author is denoted as a_i. Node s_0 is a virtual node representing the empty repository before any changes are committed. E is the set of edges, representing development dependencies between revisions. Δ is a labeling of E that represents code changes and there is a one-to-one mapping between the elements in E and Δ. We adapt our definition of code changes from Zeller and Hildebrandt [39], where a change δ is a mapping from old code to new code. An edge $e_{i,j}(\delta_{i,j})$, $e_{i,j} \in E$ and $\delta_{i,j} \in \Delta$, means that by applying the change $\delta_{i,j}$ to s_i, we can get code snapshot s_j; so $\delta_{i,j}(s_i) = s_j$. We define $\delta_{i,j}$ to be a tuple of $(\mathcal{D}_{i,j}, \mathcal{A}_{i,j}, \mathcal{C}_{i,j})$ where $\mathcal{D}_{i,j}$ is the set of lines deleted from s_i, $\mathcal{A}_{i,j}$ is the set of lines added to s_i, and $\mathcal{C}_{i,j}$ is the set of pairs of lines changed from s_i to s_j. For node s_i, s_j, and s_k such that $e_{i,j} \in E$ and $e_{j,k} \in E$, we define the composition of change sets as $\delta_{i,k} = \delta_{j,k} \circ \delta_{i,j}$ meaning applying $\delta_{i,j}$ first, and then $\delta_{j,k}$. Our definition implies that the operator $\circ$ is right associative. One key property of the composition operation is that the result of composition of change sets is path independent. The result only depends on the two end nodes.

We illustrate our definition in Figure 1. The repository consists of ten revisions (ten nodes) and three developers: Alice, Bob, and Jim. The author information for a node is represented by its color. The virtual node s_0 has no author information, so we leave it blank. Alice created a branch for her work and committed s_3 and s_4. Later Bob merged Alice's work back to the master branch and created s_7. For the path independent property, we have $(\delta_{4,7} \circ \delta_{3,4} \circ \delta_{2,3})(s_2) = (\delta_{6,7} \circ \delta_{5,6} \circ \delta_{2,5})(s_2) = s_7$. The first part in the equation is the composition along Alice's branch. The second one is the composition along the master branch. The two paths yield the same overall effects, which is the third part in the equation.

Our definition of the repository graph is applicable on any current mainstream VCS. To demonstrate that, we consider how to derive the nodes, edges, and the change sets on edges in three popular version control systems: git, svn, and CVS. Git and svn store each commit as a snapshot of the repository, so the commits correspond to the nodes in the repository graph. CVS on the other hand stores commits as the change set containing added lines and deleted lines. We can derive the contents of nodes by composing consecutive change sets. All the three version control systems record branching and merging, so edges are easy to find. Git and svn provide built-in differencing tools to calculate change sets, but they are not sufficient for our definition of δ because they report changed lines separately as added lines and deleted lines. *ldiff* [8] calculates source code similarity metrics (such as best edit distance and cosine similarity) to match added lines and deleted lines and derive pairs of changed lines. We use *ldiff* to implement our definition of δ. Since we can implement the repository graph on any mainstream VCS, we assume a code repository is represented as a repository graph in the following sections.

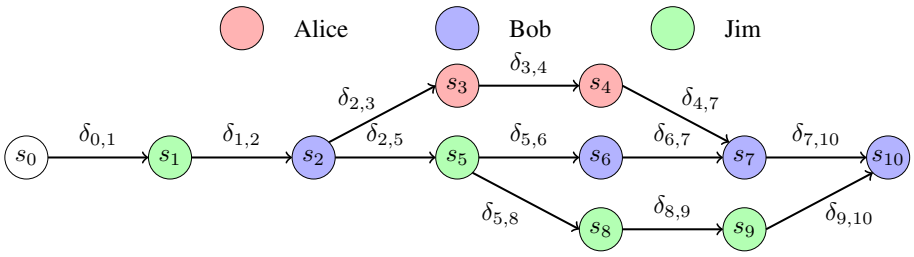

Fig. 1. An example of the repository graph. Nodes are source code revisions, denoted from s_0 to s_{10}. The color of a node shows the author creating the revision. The virtual node s_0 has no author information. Edges represent development dependencies between revisions. $\delta_{i,j}$ on edge $e_{i,j}$ is the code change from s_i to s_j.

III. Structural Authorship

Structural authorship represents the development history of a line of code. We define structural authorship as a subgraph G_l of the repository graph G that includes only the revisions that change line l of code, and the development dependences between these revisions. We present a backward flow analysis algorithm on the repository graph G that extracts the structural authorship. Our analysis processes all lines in a file to provide sufficient context for programmers to view code history. After extracting structural authorship, analysis tools have access to all historical information of a line so that they are not limited to the last change of that line.

Our structural authorship model can be seen as a generalization of the current method that only reports the last change. Both our model and the current method stop searching the history of a line when the line is found to be added. The distinction is that our model can make use of the information in the set of changed lines $\mathcal{C}$, while the current method cannot.

A. Model definition

For a given line of code l appearing in a revision s_v (often the head revision), the *structural authorship* of the pair of (s_v, l) is defined to be a directed graph $G_l = (V_l, E_l, \Delta_l)$. V_l is the set of nodes that changed or added line l. E_l is the set of edges that represent development dependences between nodes in V_l. Δ_l is a labeling of E_l that represents code changes. Before giving the formal definitions of V_l, E_l, and Δ_l, we first introduce notation to describe the relationships between nodes and then extend our definition of $\delta_{i,j}$.

We define $s_i \rightarrow s_j$ if and only if there is a directed path in G from s_i to s_j. For the starting revision s_v, its ancestor set contains the potential revisions that could be in V_l. We define the ancestor set of a node s_i as

$$ance(s_i) = \{s_k \in V | s_k \rightarrow s_i\}$$

To determine what lines a node s_i has changed or added, we define the total effect of s_i as:

$$\delta_i = \bigcup_{s_k \in pred(s_i)} \delta_{k,i}$$
$$\mathcal{D}_i = \bigcup_{s_k \in pred(s_i)} \mathcal{D}_{k,i}$$
$$\mathcal{A}_i = \bigcup_{s_k \in pred(s_i)} \mathcal{A}_{k,i}$$
$$\mathcal{C}_i = \bigcup_{s_k \in pred(s_i)} \mathcal{C}_{k,i}$$

Now we can define V_l as the set of revisions of s_v and its ancestors that add or change the line l:

$$V_l = \{s_j \in (ance(s_v) \cup \{s_v\}) | l \in (\mathcal{A}_j \cup \mathcal{C}_j)\}$$

An edge in E_l represents a path that does not go through nodes in V_l. For s_i and s_j such that $s_i \rightarrow s_j$, we define $s_i \xrightarrow{\overline{V_l}} s_j$ if and only if there exists one or more directed paths from s_i to s_j and none of the intermediate nodes on the path are in V_l. This relationship is used to describe the development dependency between two nodes in V_l. We can define E_l as:

$$E_l = \{e_{i,j} | (s_i, s_j \in V_l) \wedge (s_i \xrightarrow{\overline{V_l}} s_j)\}$$

Note that a single $e_{i,j}$ in E_l can result from multiple directed paths in the original G.

We now extend our definition of $\delta_{i,j}$ to the case where $s_i \rightarrow s_j$ so that δ can be used to describe Δ_l. If $<s_i, s_{k_1}, \ldots, s_{k_m}, s_j>$ is a directed path from s_i to s_j, then

$$\delta_{i,j} = \delta_{k_m,j} \circ \delta_{k_{m-1},k_m} \circ \cdots \circ \delta_{i,k_1}$$

Note that the specific choice of the path is not important because the result of composition of change sets is path independent. Δ_l then can be defined as

$$\Delta_l = \{\delta_{i,j} | e_{i,j} \in E_l\}$$

We illustrate our subgraph definition with an example. In the repository graph G shown in Figure 1, suppose we have the following scenario: Line l was first introduced into the project by Bob in revision 2 (s_2). Alice changed l in revisions 3 and 4 in her branch. Jim changed l in revision 9 in his branch. Bob merged Alice's branch in revision 7. Since Alice and Jim made independent changes to l, when Bob finally tried to merge Jim's branch, Bob had to solve the conflict by taking either Alice's change or Jim's change; we assume that Bob took Jim's change. The structural authorship G_l is shown in Figure 2.

B. Backward flow analysis

We calculate the structural authorship graphs in two steps. In the first step, we use a backward flow analysis to calculate

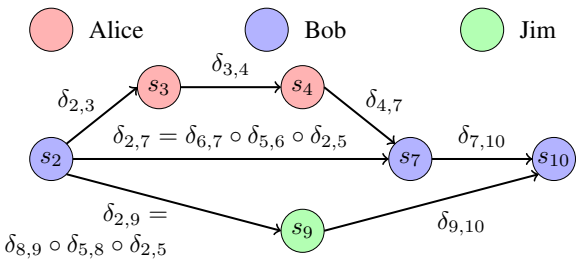

Fig. 2. An example of the structural authorship graph. Nodes in $V_l = \{s_2, s_3, s_4, s_7, s_9, s_{10}\}$ changed or added line l. Edges represent extended development dependencies between revisions. $\delta_{i,j}$ on edge $e_{i,j}$ is the extended code change from s_i to s_j.

input : V, E, F, and s_v
output: $\{V_l | l \in F\}$
// The live lines that can reach s_v
1 *liveLines*$[s_v] \leftarrow F$;
2 **for** $s_i \in (ance(s_v) \cup \{s_v\})$ *in reverse topological order in* G **do**
// Phase 1: calculate δ for s_i
3 **for** $s_k \in pred(s_i)$ **do**
4 $\delta_{k,i} \leftarrow$ *ldiff*(s_k, s_i, F);
5 $\delta_i \leftarrow \delta_i \cup \delta_{k,i}$;
// Phase 2: update V_l
6 **for** $l \in$ *liveLines*$[s_i]$ **do**
7 **if** $l \in \mathcal{A}_i \cup \mathcal{C}_i$ **then**
8 $V_l \leftarrow V_l \cup \{s_i\}$;
// Phase 3: pass live lines to preds
9 **for** $s_k \in pred(s_i)$ **do**
10 **for** $l \in$ *liveLines*$[s_i]$ **do**
11 **if** $l \notin \mathcal{A}_{k,i}$ **then**
12 *liveLines*$[s_k] \leftarrow$ *liveLines*$[s_k] \cup \{l\}$;
13 *liveLines*$[s_i] \leftarrow \emptyset$;

Fig. 3. *S-Author*: An algorithm that extracts V_l for all lines of code in file F starting at revision s_v

V_l. In the second step, a depth first search is used to calculate E_l and Δ_l. In our repository graph abstraction, V and E can be directly accessed through API of the underlying VCS, but we have to use *ldiff* to calculate Δ in our analysis.

In the first step, we use the backward flow analysis shown in Figure 3 to extract V_l from the repository graph G. We perform our analysis on all of the lines in a file F rather than an individual line l for two reasons. First, by processing all lines in F together, we can order the computation so that we neither make redundant calls to *ldiff* nor store the results of *ldiff*. Second, programmers usually want to view code history in a context, so presenting histories of several lines together is more useful.

Our algorithm calculates dataflow information for each node and adds nodes to V_l. For node s_i, its dataflow information records the live lines that can reach the starting node s_v from s_i before being deleted. We use a map *liveLines* that associates a node to a set of live lines to efficiently update the dataflow information. At the beginning, all lines in F are live (line 1).

Because G is acyclic, the traditional work list algorithm for dataflow analysis is not necessary in our case. It is sufficient to visit each node from s_v in the reverse topological order of G (line 2). For each node s_i, there are three major phases: calculating change sets (lines 3-5), updating V_l (lines 6-8) and passing live lines to the predecessors of s_i (lines 9-12).

In phase 1, we call *ldiff* to calculate a subset of Δ that are sufficient and necessary for the next two phases. In phase 2, for each live line l, we determine whether s_i is in V_l or not (line 7). In phase 3, we check whether the current live lines will still be live in each predecessor s_k of s_i (line 11). It is possible that l will be dead along one branch, but still be live along another branch.

The analysis finishes after it visits the virtual node s_0. As a special case, we can add s_0 to V_l to represent the state where l has not yet been introduced into the repository. For any $l \in F$, V_l are the nodes in the structural authorship graph.

The memory used for the results of *ldiff* in phase 1 can be freed after the phase 3 in this iteration. *ldiff* produces the δ between two files and has a relative high time complexity, quadratic in terms of the size of the files [8]. Caching the results of *ldiff* can avoid redundant calls to *ldiff*. But we estimate that caching the results of *ldiff* on a large code repository could take a few gigabytes of memory, which is too much for a built-in tool for a VCS.

In the second step, for each node that we have determined is in V_l, we can do a depth first search in G to calculate E_l and Δ_l according to our definitions.

The running efficiency of our algorithms both depends on the actual sizes of structural authorship graphs. G_l could be as large as G in theory. However, G_l is usually small in practice (Section 5.1) and our algorithms show good performance.

IV. Weighted Authorship

The structural authorship graph G_l represents the complete development history of a line of code l. However, existing analysis tools typically operate on numerical or ordinal features rather than a graph, so we wish to provide summaries of this information in a form such tools can consume. We define the *weighted authorship* of l to be a vector of author contribution weights. For each author, we can then use the weighted authorship to determine their contribution, model their familiarity of the line, or estimate their efforts spent on the line. This type of summary information is often used to analyze software quality [5, 32], help familiarize new developers [13], and estimate software development cost [26].

A. Model description

For a line of code l, we define the *weighted authorship* W_l as a vector $(c_1, c_2, \ldots, c_m)$. Each element c_i is the percentage of contribution made by developer i; elements in W_l sum to 1. m is the total number of developers that changed l. By examining G_l, we can determine the value of m. We define each c_i to be the number of characters attributed to developer i divided by the total number of characters in l. For example, if Alice, Bob and Jim are developers 1, 2 and 3, $W_l = (30\%, 20\%, 50\%)$ means that Alice, Bob, and Jim contribute 30%, 20%, 50% of the line respectively. We use

```
   input : l, V_l, E_l, and Δ_l
   output: attr: maps a character in l to its attributed node
1  Let s_v be the last node that changed l;
   // The live characters that can reach s_v
2  liveC[s_v] ← l;
3  for s_i ∈ V_l in reverse topological order in G_l do
4      if |pred(s_i)| == 1 then
           // s_i is created by a normal commit
5          Let s_k be the element in pred(s_i);
6          chars ← AC-BestEdit(l, δ_{k,i});
7          for c ∈ liveC[s_i] ∩ chars do
8              if (c ∉ attr.keys()) or (tstamp(s_i) < tstamp(attr[c]))
               then
9                  attr[c] ← s_i;
10         liveC[s_k] ← liveC[s_k] ∪ (liveC[s_i] − chars);
11     else
           // s_i is created by a merge commit
12         for s_k ∈ pred(s_i) do
13             chars ← AC-BestEdit(l, δ_{k,i});
14             liveC[s_k] ← liveC[s_k] ∪ (liveC[s_i] − chars);
```

Fig. 4. *W-Author*: An algorithm calculating the attribution map for l

characters as the unit of contribution because it is simple and avoids being dependent on the programming language used. While we do not consider the semantics of the code, we do collapse white space to minimize the effects of simple formatting changes. We do not isolate the affect of each of these choices, however the experiments in the following section show that these choices produce satisfactory results.

B. Algorithm

We calculate W_l based on G_l. We first attribute each character in l to the node that introduced that character and then attribute each node to the appropriate developer. We define the attribution map *attr* to maintain this character-to-node attribution. The node-to-developer attribution can be done by checking the author label of each node.

We use the algorithm shown in Figure 4 to compute the attribution map *attr*. The idea is to attribute a character to the node in which the character is added or changed. The algorithm first finds the last revision s_v that changed l; this s_v is the starting point of our algorithm (line 1). For each node in G_l, we maintain the live characters that can reach s_v before being deleted. All characters in l at s_v are live (line 2). We visit each node in G_l in reverse topological order. For each node s_i, we distinguish whether s_i is created by a normal commit or a merge commit by checking the number of its predecessors (line 4). In both case, we define *AC-BestEdit* to calculate the set of characters added or changed in this node (line 6 and 13). These characters are not passed to the predecessors of s_i. For a normal commit, we update the attribution map and pass the live characters (lines 5-10). For a merge commit, we only pass the live characters (lines 12-14).

AC-BestEdit adapts the Wagner-Fischer algorithm [36] for computing the best edit distance to calculate the set of characters in l added or changed by s_i. In the Wagner-Fischer algorithm, the best distance is defined as the minimum number of steps needed to change a source string to a target string. Each step can be adding, deleting, or substituting a character. The algorithm computes a shortest path and returns the minimal number of steps. For an edge $e_{k,i} \in E_l$, the string in s_k is the source string and the string in s_i is the target string. *AC-BestEdit* calculates the shortest path to change the source string to the target string using Wagner-Fischer, and it returns the set of characters added or changed by s_i.

A normal commit has a single predecessor s_k. A character that is added or changed in this node may be also added or changed independently in other nodes (in other branches). Since characters are the unit of contribution, we do not divide the contribution of a character among the multiple commits. In this case, we attribute the character to the node with the earlier commit timestamp (line 8).

For a merge commit, we assume that the commit is either produced during an automatic merge by the VCS or manual selections from one of the multiple branches; therefore a merge commit does not introduce new characters. Since the added or changed characters in one branch actually come from other branches, we just ignore these characters in this merge commit and attribute them to other branches.

The performance of the algorithm depends on the size of G_l. As we will discuss in the next section, the size of G_l is usually small. In our experience, running this algorithm on all lines in a file finishes in around second.

V. Evaluation

We implemented our structural authorship model and weighted authorship model in a new git built-in tool: *git-author*. *git-author* uses a syntax similar to that of *git-blame* so has a familiar feel to current users of git. We designed two experiments to compare our new authorship models to the current model that only reports the last change to a line. In the first experiment, we ran *git-author* on five open source code repositories to study the number of lines that were changed in multiple commits and the number of lines that were changed by multiple authors. This experiment shows that *git-author* can recover more information than *git-blame* on about 10% of lines. The results show that most lines are touched only by one author in one commit and the cooperation between developers is restricted to small regions of code. We hypothesized that these small regions of code contain rich information about the software development process and that analysis tools can benefit from this extra information. We conducted our second experiment to verify this hypothesis. Our second experiment evaluated whether the additional information would be useful to build a better analysis tool. We built a new line-level model for source code bug prediction and compared it with the best previously report work on a file-level model [22]. We found that our line-level model consistently performed better than the file-level model. This demonstrates that our new authorship models can help build better analysis tools.

A. Multi-author study

In this experiment, we ran *git-author* on the following five open source projects: Dyninst [31], the Apache HTTP

Repository	Multi. Commits	Multi. Authors	# of lines
Dyninst	53K (12.11%)	40K (9.12%)	434K
Httpd	27K (10.90%)	20K (8.15%)	247K
GCC	279K (8.08%)	217K (6.27%)	3454K
Linux	1440K (9.69%)	1072K (7.22%)	14857K
GIMP	122K (12.82%)	78K (8.12%)	955K

TABLE I
NUMBER OF LINES CHANGED BY MULTIPLE COMMITS AND MULTPLE AUTHORS. THE SECOND COLUMN SHOWS THE NUMBER OF LINES CHANGED IN MULTIPLE COMMITS AND THE PERCENTAGE THEY ACCOUNT FOR IN THE REPOSITORY. THE THIRD COLUMN SHOWS THE SAME INFORMATION FOR LINES THAT CHANGED BY MULTIPLE AUTHORS.

server [1], GCC [15], the Linux Kernel [24], and Gimp [16], extracting the structural authorship for each line of the code. We then counted the number of nodes and the number of authors in each structural authorship graph. Note that we did not run *git-blame* on the five projects because *git-blame* would output only one commit and one author for each line of code.

The results are shown in Table I. About 10% of lines are changed by multiple commits and about 8% of lines are changed by multiple authors. *git-author* produces more information than *git-blame* on these lines.

B. Line-level bug prediction

Our second experiment evaluated whether the information provided by *git-author* would be helpful to build a better bug prediction model. We show that we can build a line-level bug prediction model that is more effective than the best previously reported work on a file-level model by Kamei, Matsumoto at el. [22]. To the best of our knowledge, we are the first project to try to predict bugs at the line level.

We first give an overview of bug prediction and our experiment. We then introduce our new line-level model and the file-level model we compared it to. We discuss our data sets and the metrics used to evaluate the models. Finally, we present our results.

1) Overview: Many research efforts have been dedicated to source code bug prediction to prioritize software testing [18, 20, 22, 29, 30]. Two comprehensive surveys are from Arisholm, Briand, al el. [2] and D'Ambros, Lanza al et. [9].

Three decisions affect the performance of a bug prediction model: the granularity of prediction, a set of bug predictors, and a machine learning technique that trains the model and predicts bugs. Using *git-author* changes the granularity of prediction to the line level and introduces new bug predictors. We do not explore the influence of different machine learning techniques as it is beyond the scope of this paper.

Most of the existing source code bug prediction models predict at the granularity of a source file [18, 28] or even a module [29, 30]. The disadvantage of coarse-grained prediction models is that, even if the prediction results are accurate, developers still have to spend effort to locate the bugs within a module or file. Predicting at a finer granularity, such as at the method level can help to reduce the problem [20, 23]. Our line-level model can locate the suspected lines and help focus testing efforts. It uses the development history of lines of code provided by *git-author* to make prediction. Note that since the development history of a line of code produced by *git-blame* is incomplete, it is impractical to do line-level prediction with *git-blame*. We compared our line-level model to the file-level model because predicting at a file level is well understood.

Level	Predictor name	Definition
Line	WA	Weighted authorship defined in Section 4
	NOA	# of authors
	NOC	# of commits
	LEN	Length of the line
	VAR	Variance of the length of the line across all commits in G_l
	FIX	# of times a line involved in a bug-fix commit
	REF	# of times a line involved in a refactoring commit
	COM	Whether a line is a comment
	AGE	The age of the line
File [22]	Codechurn	Sum of (added lines of code - deleted lines of code)
	LOCAdd	Sum of added lines of code over all revisions
	LOCDel	Sum of deleted lines of code over all revisions
	Revisions	# of revisions
	Age	The age of the file
	BugFixes	# of times a file involved in a bug-fix commit
	Refactor	# of times a file involved in a refactoring commit

TABLE II
BUG PREDICTORS USED IN THE STUDY.

Two types of bug predictors are commonly used: product predictors that summarize code in the predicting snapshot [41] and process predictors that summarize the history of the predicting snapshot [28]. The process predictors have been shown to be more effective than the product predictors [22, 28]. In our experiment, most of our predictors are process predictors.

Many machine learning techniques have been adopted for bug prediction. However, previous studies have shown that the influence of bug predictors on the final prediction results is much larger than the chosen machine learning technique [2, 22]. Therefore, we selected linear learning techniques for both our line-level model and the file-level model. We do not believe this choice will have a noticeable effect on our results.

2) Models: The goal of our line-level model is, given a line of code, to output the probability that the line is buggy. Based on these outputs, a developer could prioritize testing of the software to the lines with higher probabilities of being buggy. We used a linear SVM as the learning technique in our line-level model [12]. The predictors in our new model are shown in Table II. We introduce new predictors including the weighted authorship, the length of the line, the variance of the length of the line across all commits in G_l, and whether the line is a comment. The other predictors were adapted from existing file-level predictors. We compute the values of these line-level predictors from the outputs of *git-author*.

We compared our line-level model to the file-level model from Kamei, Matsumoto et al. [22]. Their model outputs the predicted fault density when given a file. They compared the prediction results of using process predictors and product

predictors with three learning techniques: linear regression [10], regression tree [7], and random forest [6]. Their results showed that using process predictors produced consistently better results than using product predictors and combining them together did not provide further advantages. Therefore, we implemented the file-level process predictors listed in Table II. We chose the logistic regression [12], one type of linear regression, as the learning technique of the file-level model to match the linear SVM used in our line-level model.

Note that when evaluating the effects of *git-author*, it would have been preferable to use the same machine learning technique in the line-level model and the file-level model. However, because the outputs of the line-level model and the file-level model are different, we cannot use the exact same learning technique. Therefore, we can only try to minimize the effects on performance from the factors rather than *git-author*.

3) Data collection: We are unaware of existing bug prediction data sets with line-level predictors; instead we generated new such data sets. Producing a bug prediction data set takes two steps. We first create a *bug map* from a bug record in the bug database to the pair of commits that caused the bug and fixed the bug. We then choose a time point, typically a release, and use the bug map to produce data instances for this snapshot. The second step is repeated at several different release time points so that we could do cross release prediction.

For the first step, we used the SZZ algorithm [35] to find buggy commits and the corresponding *fixing commits* that fixed the bugs in the Apache HTTP server repository. The quality of the results in this step is improved by *Relink* [37], which addresses the problem of missing valid mappings in the original SZZ algorithm [4].

In the second step, we projected the *bug map* onto the chosen snapshot. A bug is relevant to the snapshot if and only if the snapshot is inside the time interval between the buggy commit and the fixing commit. For each relevant bug, we first produced line-level data, and then summarized the data into file-level data. We assume the lines that are deleted or changed in the fixing commit are the buggy lines. Two methods can be used to summarize the line-level data. We can either count all buggy lines as a single bug or count the lines separately. The first method assumes that it takes the same effort to fix every bug, while the second method takes this factor into consideration. We adopted both methods and produced two data sets.

We collected data for seven releases in the Apache HTTP server project and produced two data sets described above. The first data set is denoted as "Bug count" and the other one is denoted as "Line count". Table III summarizes our data sets.

4) Evaluation metrics: Many metrics are used to evaluate bug prediction models. The most commonly used metrics include precision and recall [29, 30], the area under the curve (AUC) of ROC curves [27, 28], and effort-aware metrics [22, 25]. Comparison studies have shown that the choices of metrics can significantly affect the performance of prediction models [2, 9]. The difference of performance on metrics does not mean inconsistent results because different metrics are designed to answer different questions. We use the effort-aware metrics because they are domain specific metrics for bug prediction. They measure not only the accuracy of the predicting results but also the efforts needed to fix the bugs.

Release	# of files	# of bugs	SLOC	# of buggy lines
2.1.1	305	129	177K	670
2.2.0	319	171	202K	746
2.2.6	320	167	205K	708
2.2.10	321	172	207K	664
2.3.0	383	179	207K	680
2.3.10	372	195	218K	747
2.4.0	362	181	223K	555

TABLE III
SUMMARY OF THE DATA SETS. EACH ROW IN THE TABLE SUMMARIZES THE NUMBER OF FILES, BUGS, LINES OF CODE, AND BUGGY LINES IN A RELEASE SNAPSHOT OF APACHE.

In our study, we use two effort-aware metrics: P_{opt}, which measures the closeness of a model to the optimal file level model [25] and cost-effectiveness (CE), which measures the advantages of that model over a random prediction model [2]. The idea of effort-aware metrics is that a developer can first test or inspect the most suspicious lines or the files with largest fault densities and see how many percent of bugs can be found. The assumption is that the effort needed to test a piece of code is roughly proportional to the size of the code [2]. Using the percent of lines tested as the x-axis and the percent of bugs covered as the y-axis, we can draw a curve to visualize the performance of a model. We denote the area under the curve of a model m as $AUC(m)$. P_{opt} and CE can be defined as:

$$P_{opt}(m) = 1 - (AUC(FileOptimal) - AUC(m))$$
$$CE(m) = \frac{AUC(m) - AUC(Random)}{AUC(FileOptimal) - AUC(Random)}$$

In the above formulas, the file optimal model tests files in decreasing order of the fault densities. It represents the upper bound of a file level model. The random model orders the files randomly. We use the average performance of the random model in the CE formula, which is a straight line from $(0, 0)$ to $(1, 1)$. For both P_{opt} and CE, larger values mean better performance. When the values are larger than 1, the model m performs better than the optimal file-level model.

5) Results: We performed cross release prediction on our data set. We chose cross release prediction instead of cross-validation inside a release because the cross-release prediction represents the real practice of how a bug prediction model is used. We used *Liblinear* to do training and prediction on our two data sets [12]. We denote our line-level model as lm, the file-level model as fm, and the optimal file-level model as fm_o.

In the "Bug count" data set, we need to aggregate line-level prediction results into the bug count. We provide three interpretations for our line level models. The first one is that we can identify a bug as long as any line comprising bug is identified. This is the optimistic interpretation and represents the maximal benefits that can be acquired by using our line-

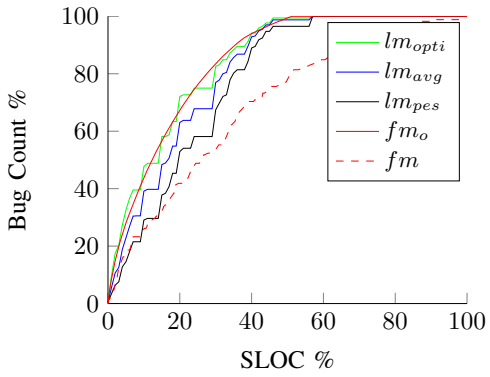

Fig. 5. Cross release prediction from 2.2.10 to 2.3.0 on "Bug count" data set. The x-axis is the percentage of source line of code to test. The y-axis is the percentage of bugs that can be identified.

level model. The second one is that we take partial credit when we identify a buggy line. For example, if we identify one buggy line for a five-line bug, we say we find 20% of a bug. This is the average interpretation and assumes that the more information about a bug is provided, the more likely the bug can be identified. The third one is that only after we identify all buggy lines of a bug, we cover the bug. This is the pessimistic interpretation. We denote the three views as lm_{opti}, lm_{avg}, and lm_{pes}.

The results for the "Bug count" data set are shown in Table IV. Our results of $P_{opt}(fm)$ are consistent with the results shown by Kamei, Matsumoto at el. [22]. The results of $CE(fm)$ are slightly better but still consistent with the results shown by Arisholm, Briand et al. [2]. Therefore, we believe that our implementation of fm is comparable to other implementations and that we can compare our lm to this implementation of fm.

The optimistic interpretation and the average interpretation are consistently much better than the file model in both P_{opt} and CE. The pessimistic interpretation loses to the file model slightly in two rounds of prediction but has a much higher mean value. All the three interpretations have much smaller standard deviation than the file model, so prediction results are more stable on line level. Notice that the value of $P_{opt}(lm_{opti})$ and $CE(lm_{opti})$ in row "2.3.10 $\rightarrow$ 2.4.0" are larger than 1, which shows that the performance of the line level model can even exceed the upper bound of file level models.

Figure 5 shows the prediction results of training on release 2.2.10 and predicting on release 2.3.0. If we only test a small amount of code, the lm_{opti} is actually better than the fm_o, but the lm_{pes} is a little bit worse than the fm. As we test more code, the three interpretations of the line-level model are consistently better than the fm.

The "Bug count" data set assumes that every bug involves the same amount of work to fix. We use the "Line count" data set to measure how many buggy lines can be covered during testing. The overall results are shown in Table V and confirm that the line level model consistently performs better

Train $\rightarrow$ Predict	P_{opt}		CE	
	lm	fm	lm	fm
2.1.1 $\rightarrow$ 2.2.0	0.9148	0.8113	0.7925	0.5404
2.2.0 $\rightarrow$ 2.2.6	0.9425	0.7704	0.8578	0.4321
2.2.6 $\rightarrow$ 2.2.10	0.9470	0.7860	0.8658	0.4579
2.2.10 $\rightarrow$ 2.3.0	0.9153	0.8288	0.7834	0.5624
2.3.0 $\rightarrow$ 2.3.10	0.8660	0.7711	0.6590	0.4173
2.3.10 $\rightarrow$ 2.4.0	0.9343	0.8860	0.8299	0.7050
Mean	0.9200	0.8089	0.7981	0.5192
Standard Deviation	0.0271	0.0404	0.0692	0.0988

TABLE V
RESULTS OF "LINE COUNT" DATA SET.

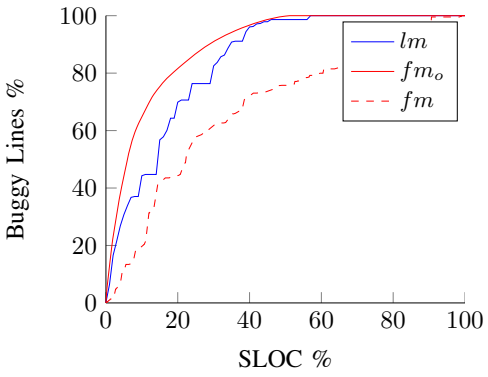

Fig. 6. Cross release prediction from 2.2.10 to 2.3.0 on "Line count" data set. The x-axis is the percentage of source line of code to test. The y-axis is the percentage of buggy lines that can be identified.

in both P_{opt} and CE. Figure 6 shows the results of training on release 2.2.10 and predicting on release 2.3.0 in the "Line count" data set. The line level model performs better than the file level model over all ranges of the curve.

In summary, our two experiments confirm the effectiveness of our new authorship models. The first experiment shows that *git-author* provides more information than *git-blame* by the structured authorship model. The second experiment shows that the information is useful to build better analysis tools.

VI. Related Work

Three types of studies are related to our work: code authorship extraction and visualization [14, 21], software quality and maintenance analysis using code authorship [5, 13, 32], and mining software repositories for histories of source code entities [3, 17, 19, 34, 40]. The first type is similar to our work in terms of the final goal that is to present authorship information to users, but the approaches and the granularity are different. The second type consumes authorship information to analyze software quality or to improve developer familiarization. The third type shares a similar approach with our work. We now discuss each type of studies in more detail.

Syde [21] is a system built on Eclipse that collects every change made by developers. Syde records changes made by developers when they try to compile the code. They then define the owner of a file as the developer making the most number of changes. With Syde's change log, refined ownership can

Train → Predict	P_{opt}				CE			
	lm_{opti}	lm_{avg}	lm_{pes}	fm	lm_{opti}	lm_{avg}	lm_{pes}	fm
2.1.1 → 2.2.0	0.9695	0.9392	0.9023	0.8321	0.9132	0.8243	0.7220	0.5221
2.2.0 → 2.2.6	0.9884	0.9632	0.9297	0.8166	0.9664	0.8935	0.7965	0.4693
2.2.6 → 2.2.10	0.9997	0.9706	0.9339	0.8453	0.9990	0.9148	0.8082	0.5509
2.2.10 → 2.3.0	0.9647	0.9325	0.8965	0.8716	0.8956	0.8007	0.6943	0.6208
2.3.0 → 2.3.10	0.9664	0.9275	0.8848	0.8870	0.8961	0.7756	0.6433	0.6504
2.3.10 → 2.4.0	**1.0013**	0.9665	0.9245	0.9267	**1.0040**	0.8979	0.7700	0.7769
Mean	0.9817	0.9499	0.9120	0.8632	0.9457	0.8511	0.7391	0.5984
Std. Dev.	0.0154	0.0173	0.0184	0.0368	0.0460	0.0532	0.0585	0.0998

TABLE IV
RESULTS OF "BUG COUNT" DATA SET. THE TWO BOLD NUMBERS IN ROW "2.3.10 → 2.4.0" ARE LARGER THAN ONE INDICATING THAT THE PERFORMANCE OF OUR LINE LEVEL MODEL CAN EXCEED THE UPPER BOUND OF ANY FILE LEVEL MODEL.

be extracted on file level. Our work differs from Syde in two ways. First, our authorship models are on line level. Second, our models are applicable to existing repositories and do not require extra compile-time information.

Rahman and Devanbu [32] analyze the relationship between code authorship and the number of defects in four open source projects. They define a file-level authorship model that computes the percentage of lines owned by each developer using *git-blame*. Fritz, Ou and et al. [13] use code authorship data and developer interaction data to model source code familiarity. Their authorship model is at the source code element level including class, method and field. We believe these studies can benefit from our new authorship models by aggregating accurate line-level authorship information into the corresponding granularities.

Kenyon [3], APFEL [40], Beagle [17] and Historage [19] mine software repositories to produce the history of code entities at granularities finer than files. Their goal is to produce rich semantics for code changes including adding, deleting, modifying, renaming and moving. Our work differs from theirs in two perspectives. First, these tools use heavy-weight semantic analysis for rich semantics of code changes. Therefore, results have to be stored in a relational database for later queries. On the contrary, our tool is light-weighted and can produce results on the fly. Second, our tool is designed to be a built-in tool of git, so it is easy to use for users who are familiar with git.

Servant and Jones [34] define the *history slice* to represent the history of a line of code. Like our structural authorship, it contains the revisions that changed the line. Unlike our model, it ignores branches and assumes that a later revision is based only on a prior revision. Therefore, two independent revisions in different branches can be dependent in history slice.

VII. CONCLUSION

We have presented two line-level authorship models: the structural authorship, which represents the complete development of a line of code, and the weighted authorship, which summarizes the structural authorship to produce author contribution weights. Our two authorship models overcome the limitations of the current methods that only report the last change to a line of code. We define the repository graph as a graph abstraction for a code repository and define a backward flow analysis on the repository graph that derives the structural authorship. Another backward flow analysis is used on the structural authorship to compute the weighted authorship. We have implemented our two authorship models in a new git built-in tool *git-author*. We have evaluated *git-author* in two experiments. In the first experiment, we ran *git-author* on five open source projects and find that *git-author* can recover more information than *git-blame* on about 10% of the lines. In the second experiment, we built a line-level model for bug prediction based on the output of *git-author*. We compared our line-level model with a representative file-level model and found that our line-level model is consistently better than the file-level model on our data sets. These results show that our new authorship models can produce more information than the existing methods and that information is useful to build a better analysis tool.

ACKNOWLEDGEMENTS

We thank Weiyang Chiew for his help in extracting the bug data sets. This work is supported in part by Department of Energy grants DE-SC0003922 and DE-SC0002155, National Science Foundation Cyber Infrastructure grants OCI-1032341, OCI-1032732, and OCI-1127210; and Department of Homeland Security under AFRL Contract FA8750-12-2-0289.

REFERENCES

[1] Apache Software Foundation. Apache http server, http://httpd.apache.org.

[2] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, Jan. 2010.

[3] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE)*, Lisbon, Portugal, Sep. 2005.

[4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, Amsterdam, The Netherlands, Aug. 2009.

[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT*

symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE), Szeged, Hungary, Sep. 2011.

[6] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct. 2001.

[7] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[8] G. Canfora, L. Cerulo, and M. D. Penta. Identifying changed source code lines from version repositories. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR)*, Minneapolis, Minnesota, USA, May 2007.

[9] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, Aug. 2012.

[10] A. L. Edwards. *Introduction to Linear Regression and Correlation*. W.H.Freeman & Co Ltd, 1976.

[11] J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, Waikiki, Honolulu, HI, USA, May 2011.

[12] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, June 2008.

[13] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, Cape Town, South Africa, May 2010.

[14] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE)*, Lisbon, Portugal, Sep. 2005.

[15] GNU Project. Gcc: The gnu compiler collection, http://gcc.gnu.org.

[16] GNU Project. The gnu image manipulation program, http://www.gimp.org.

[17] 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.

[18] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009.

[19] H. Hata, O. Mizuno, and T. Kikuno. Historage: fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution (IWPSE-EVOL)*, Szeged, Hungary, Sep. 2011.

[20] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, May 2012.

[21] L. Hattori and M. Lanza. Mining the history of synchronous changes to refine code ownership. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories (MSR)*, Vancouver, Canada, May 2009.

[22] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2010.

[23] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering (ICSE)*, Minneapolis, Minnesota, USA, May 2007.

[24] Linux Kernel Project. Linux kernel, http://www.kernel.org.

[25] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE)*, Vancouver, British Columbia, Canada, May 2009.

[26] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lawrence, Kansas, USA, Nov. 2011.

[27] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, Jan. 2007.

[28] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering (ICSE)*, Leipzig, Germany, May 2008.

[29] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering (ICSE)*, Leipzig, Germany, May 2008.

[30] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, San Jose, CA, USA, Nov. 2010.

[31] Paradyn Project. Dyninst: Putting the Performance in High Performance Computing, http://www.dyninst.org.

[32] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, Waikiki, Honolulu, HI, USA, May 2011.

[33] N. Rosenblum, X. Zhu, and B. P. Miller. Who wrote this code? identifying the authors of program binaries. In *Proceedings of the 16th European Conference on Research in Computer Security (ESORICS)*, Leuven, Belgium, Sep. 2011.

[34] F. Servant and J. A. Jones. History slicing: assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, Cary, North Carolina, Sep. 2012.

[35] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, USA, May 2005.

[36] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, Jan. 1974.

[37] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, ESEC/FSE '11, Szeged, Hungary, Sep. 2011.

[38] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, Szeged, Hungary, Sep. 2011.

[39] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.

[40] T. Zimmermann. Fine-grained processing of cvs archives with apfel. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange (eclipse)*, Portland, Oregon, Oct. 2006.

[41] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE)*, Minneapolis, Minnesota, USA, May 2007.