Are Developers Fixing Their Own Bugs?
Tracing Bug-fixing and Bug-seeding Committers

Daniel Izquierdo-Cortazar,
Andrea Capiluppi, Jesus M. Gonzalez-Barahona
Universidad Rey Juan Carlos, University of East London
{dizquierdo, jgb}@gsyc.es, a.capiluppi@uel.ac.uk

Abstract
The process of fixing software bugs plays a key role in the maintenance activities of a software project. Ideally, code ownership and responsibility should be enforced among developers working on the same artifacts, so that those introducing buggy code could also contribute to its fix. However, especially in FLOSS projects, this mechanism is not clearly understood; in particular, it is not known whether those contributors fixing a bug are the same seeding it in the first place.

This paper aims to study this issue, by analysing the comm-central FLOSS project, which hosts part of the Thunderbird, SeaMonkey, Lightning extensions and Sunbird projects from the Mozilla community. The analysis is focused at the level of lines of code and it uses the information stored in the source code management system, parsed using the diff tool.

The results of this study show, at first, that in 80% of the cases, the bug-fixing activity involves source code modified by at most two developers. Second, it emerges that the developers fixing the bug are only responsible for 3.5% of the previous modifications to the lines affected; this implies that the other developers making changes to those lines could have made that fix. We conclude by stating that, in most of the cases the bug fixing process in comm-central is not carried out by the same developers than those who seeded the buggy code.

1 Introduction
One of the most recognised advantages of the Free/Libre/Open Source Software (FLOSS) development model is its reliance on an open process: anyone is welcome to contribute; the majority of developers can focus on modularised, limited sections within a very large and complex system; and few core developers are generally experts in several areas of the source code, in a well accepted layered model (the “onion
model” [Error: Reference source not found]). These layers have been connected to actual responsibilities; core developers should focus on the main, more important features, while experimental versions should be implemented and tested by contributors on the development fringes [Error: Reference source not found]. Also, the layers of such model have been related to a shift in productivity: a recurring finding within FLOSS empirical research has shown that most of the development work is achieved by a small amount of developers, in a typical Pareto distribution [Error: Reference source not found].

The combinations of all the findings above have various, and not completely understood, effects. In some cases, a strong territoriality will emerge among developers “owning” certain parts of the code, and becoming more and more proficient in those [Error: Reference source not found, Error: Reference source not found]. In other cases, the very nature of the FLOSS development implies that contributors join and then leave without necessarily halting the project [Error: Reference source not found], but resulting in abandoned code and orphaned lines [Error: Reference source not found]. Finally, certain developers will need to be active in maintenance activities: corrective maintenance fixing bugs in various parts of the code, for instance when source code is first introduced by developers with a low knowledge of the project (junior developers), perfective maintenance when developers leave the project and abandon their contributions [Error: Reference source not found], or adaptive maintenance when source code is contributed in a programming language different from the main one supported by the project. Although in specific FLOSS communities there is the shared expectation that the original contributor will support his/her modules [Error: Reference source not found] (especially in highly modular FLOSS projects, as Moodle or Drupal), the volatility of contributors and the process of bug fixing need to be clarified with respect of who introduced a certain bug, and who contributed the code to fix it. Examining and determining the proportion of errors that are fixed by different developers than those who introduced the error could provide a first approach to better understand the bug-fixing process in the specific FLOSS communities being studied.

In order to tackle this problem, the present study analyses the code base contained within the comm-central project, a Mercurial SCM repository of Mozilla components (Thunderbird, SeaMonkey, the Lightning extension and Sunbird). Given the number and ID of each fixed bug, this research evaluates which changes have been performed, and by who, in the process of fixing the specific bug. The objective of this research is to evaluate patterns of bug-fixing activities within this FLOSS community, in order to detect, if any, the most recurrent and relevant scenarios among developers fixing bugs and those seeding the problem in the first place.

This paper makes two main contributions:

• Identifying bug-fixing and bug-seeding committers: the detection of those commits that have fixed a bug is crucial to determine the previous changes that took place to seed that bug. Using the source code lines that were handled by committers and tracing their history back make possible to know who previously handled those lines. Thus, it is possible to trace the changes in the SCM that made possible the birth of a potential bug.

http://hg.mozilla.org/comm-central
Characterization of bug-seeding activity: once the bug-seeding commits have been detected, it is also interesting to know how many developers have been involved in those commits that later has been raised as a bug. With this approach, we are able to know the number of people that added or modified a piece of source code before it was detected as an issue by the community.

The paper is organized into the following sections. Section Error: Reference source not found analyzes the related work and the background for the study. Section Error: Reference source not found and Error: Reference source not found introduce the technique used to extract data from the Mercurial SCM based on the hg diff tool. Section Error: Reference source not found presents the main results found after using the proposed method comm-central, while section Error: Reference source not found raises a set of threats to validity. Finally section Error: Reference source not found concludes the paper with pointers towards some further work yet to be implemented.

2 Background and Related Work

This section reports on the related work and the existing tool-sets, and it shows how this research builds on, compares to or complements existing approaches and results.

This paper uses the diff tool to identify changes between revisions: diff is provided by several source code management systems, and its basic algorithm has been theoretically and extensively explained ([Error: Reference source not found], [Error: Reference source not found], [Error: Reference source not found]). This tool basically collects two revisions of a file (or revisions of the same directory) and it returns the differences found between them. Its main goal is to look for “plain” differences between two files: however, its implementation contains both a way to identify the “actual” differences between two files, and a facility to ignore “apparent” differences (e.g., spaces, indentations, newlines, etc). The GNU implementation of this algorithm is explained in [Error: Reference source not found]: this paper uses the “unified” format of the diff algorithm to retrieve all the differences between each two revisions of the source code found at the Mercurial repository of the comm-central project. Other researchers used the diff tool in their approach when retrieving data from FLOSS repositories (specifically CVS and Subversion [Error: Reference source not found], [Error: Reference source not found]).

Previous studies have made use of both SCM repositories and log messages left by developers, as a way to determine whether an observed activity is a bug-fixing process. Focus has been given to how developers should know precisely how this is being carried out (i.e., the process) and by whom (i.e., the responsibilities [Error: Reference source not found]). Some authors [Error: Reference source not found], [Error: Reference source not found] have worked at this level; however, it has to become clear that some FLOSS communities are more effective than others in documenting whether a commit is fixing an existing bug, or if it is a more generic maintenance activity. The present study is only based on the Mozilla Community, since within this community, it is relatively simpler (compared to other communities) to determine if one of its commits is related to a bug in the Bug Tracking Systems. In this community, and within the SCM recorded activities, most of the commits dealing with bug fixes (or related to an open bug report) are tagged with an initial word "bug"
or "Bug". In some rare occasions, these have been detected to be generic features and not real bugs. A cross-validation is performed below, in order to visualise the precision and recall of this approximation, and it is shown that the above mismatch represents a minor number of occurrences.

Similarly to previous studies [Error: Reference source not found, Error: Reference source not found], this research is performed at the granularity level of source lines, which provides a way of handling the ambiguity of working with commits. When considering the committer A who fixes a certain bug, and the lines she modifies, some of these lines could have been introduced fully or partially by the same committer, or introduced by different committers without the participation of A (pictured in Figure Error: Reference source not found). Extending these two basic scenarios, we could find further scenarios:

- the same set of lines was modified in a previous commit by the same developer A (only);
- the same set of lines was modified in a previous commit by a different developer B (only);
- the same set of lines was modified by more than one developer (A+B+C+...), including the same developer A fixing the bug;
- the same set of lines was modified by more than one developer (B+C+D+...), but excluding the developer fixing the bug;

Figure 1: Scenarios of committers and lines changed. Lines that are introduced in a given fix time (t-1) are later (t) detected as being part of a bug-fixing commit. Thus, the set of lines that are being handled in (t) could have been previously introduced by the same developer (A), partially introduced by the same developer or introduced by a different developer.

In terms of relating the bug-fixing process and its responsibilities, some authors have dealt with the idea of who should be fixing a certain bug [Error: Reference source not found, Error: Reference source not found] based on previous changes of the same file, or at least slices of the changes introduced in a file. Another approach used to deal with the
same problem has been adopted at the level of the bug tracking system. In a study based on the development of Microsoft Windows Vista and Windows 7, it has been found that the number of reports “opened” by one developer and initially “assigned” to her development team tend to be fixed more quickly than bugs that are assigned to another development team [Error: Reference source not found]. Finally, it has also been reported that specific FLOSS communities try and reinforce a per-contributor sense of responsibility: in highly modular projects (as for instance Moodle or Drupal), for example, it is a shared expectation within the community that the original contributor will support his/her modules [Error: Reference source not found] and keep them in sync with the evolution of the core system [Error: Reference source not found]. Finally, other authors, have deal with the idea of looking for bug fixing patterns in the source code [Error: Reference source not found] analyzing the different revisions provided by a given SCM system, but focusing on the semantics of the source code. In other words, they are aware of several common fix patterns such as “addition of precondition check” or ”different method call to a class instance”. However, at the level of the source code, and to the best of our knowledge, no studies aiming to determine if developers that fixed the bug are the same than those who introduced the bug have been undertaken.

3 Assumptions and Definitions

3.1 Assumptions – SZZ algorithm

The main goal of the SZZ algorithm [Error: Reference source not found] is to determine (after identifying the bug-fixing commits, and by using a diff tool) the origins of a bug. The SZZ’s authors assume that those lines that have been removed or modified are the ones where the bug is located. Thus, tracing back the origins of those lines (by means of the `annotate` command in the SCM), the authors could reach the origins of those lines, and admittedly, the origins of the bug. Generally speaking, the first modification or addition of those `modified` or `removed` lines are the origins of that bug. The operationalization of the algorithm used in this paper is a bit different, but based on the same assumption that the previous modification of the removed and modified lines that takes place in the process of fixing a bug in a given commit are the ones which originated or `seeded` that bug.

3.2 Definitions

This study is based on (and could be extended to) projects which use a distributed SCM system called Mercurial. For each of the projects analyzed, we have studied the log provided by each of the named SCMs. For this purpose, these are the definitions that were used in the following empirical study:

- Commit (or revision): change on the source code submitted to the source code management system. This updates the current version of the tree directory
with a new set of changes. Those changes are generally summarized in a *patch* which is a set of lines with specific information about the affected files, but also about the affected lines.

- **Committer**: person who has rights to commit to a specific SCM repository, hence allowed to make changes. The Mercurial case presents some peculiarities: those developers working as a maintainers and uploading changes to the main branch of the repository are not registered by the Mercurial SCM. Thus, all of the changes are initially considered as being the original author. Thus, through this paper, the concept of developer, committer or author will be considered as synonyms. However, out of this paper, and depending on the SCM, those concepts are slightly different.

- **Bug-fixing commit**: this is a special type of commit where some issues as reported by other developers have been fixed. In the comm-central repository this is generally reported in the title by referencing a “bug” or a “Bug”.

- **Line**: this is the basic piece of information of this study and they are generally speaking handled by committers. These lines could be *added* - new line, *modified* - modification of some part of that line and *removed* - there is a deletion of that line.

- **Bug fixing-commit and bug seeding-commit**: given a commit, and the output of the `hg diff` command, it is possible to obtain a complete picture of the lines that were added, modified or removed, but also about the committer, the date and which files were handled. This is necessary first to track which lines have been changed for fixing a bug, and second to track which committers have provided changes to the same set of lines in previous commits. Figure [Error: Reference source not found](#) shows how the latter identification has been achieved.

In the example file (right), three sets of lines can be recognised (“set of lines 1”, “set of lines 2” and “set of lines 3”): the first two sets are affected by changes, the third has been unchanged throughout.

---

2For more information regarding to this issue, the Mercurial website offers a set of third part extensions where this issue could be solved: [http://mercurial.selenic.com/wiki/UsingExtensions](http://mercurial.selenic.com/wiki/UsingExtensions)
Tracking back the history of each set in the database, we are able to know that “set of lines 1” was added in commit number 1 and then modified in commit number 5. With respect to “set of lines 2”, they were added in commit number 5 and modified in commit number 7. With respect to the authorship, we know that the “set of lines 1” was added by a developer named A. The modification of “set of lines 1” and the addition of the “set of lines 2” was done by the same committer, named C and finally, in commit 7 changes were made on the ”set of lines 2” by developer C. In this specific figure, other commits might have happened, but they have not modified or removed the set of lines we are interested in. Specifically, commit number 2, 3, 4 and 6 took place, but none of them modified the studied sets of lines.

4 Empirical Approach and Operationalization

As the main goal of this research, this paper aims to identify and characterise the bug-fixing and bug-seeding activities in FLOSS communities. From a managerial perspective, the bug-seeding activity could be useful to clarify how and when the buggy source code has been introduced into the repository, how developers deal with this, and which effort needs to be applied and by who. In addition, specific sub-questions were formulated to achieve the main goal of the paper:

1. How are the bugs in commit-central recorded and referred to by developers? What is the accuracy and consistency of recording such bug-fixing information?

Rationale: from the maintenance point of view, it is necessary to study how the community records which issues have being fixed. The empirical approach used in this paper is based on the information provided by the log message left
by the developers when fixing a bug. This information depends on the
analysed community (i.e., Mozilla), and it could be recorded differently in
another community.

2. How can one define the activities of bug-fixing and bug-seeding when
tracking the same set of lines?
   **Rationale:** this question is related to the detection of bug-seeding commits
that later were classified as “buggy” by the community. And more specifically,
how they are detected by means of the differences found between each pair of
revisions in the source code.

3. Are there specific events in the activity log that could impede the correct
tracking of such set of lines? How to avoid that such events interfere with the
tracking of a given set of lines?
   **Rationale:** Some events in the community could force to move huge quantities
of source code to another repository (e.g., in case of migrations), refactoring
(e.g., when changing loads of methods names), license requirements (e.g.,
when migrating to another license) or others. These factors can cause large
peaks to be visualised in the evolutionary trends, that could artificially skew
the results.

4. Are there recurring patterns of bug-fixing among the developers of the comm-
central community?
   **Rationale:** this questions aims to study the behaviour of developers when
fixing bugs and try to look for specific patterns of bug-seeding activity. It is
still not well understood how bugs are being introduced in the source code and
if those developers that usually introduce issues are the same ones in charge of
fixing them. Another interesting question is the one related to how many
people are usually introducing changes to the same piece of source code that
later is found to be “buggy”.

### 4.1 Understanding the diff output

Past research studies have focused on source code lines in two ways, either by using
the source code management system (SCM) hosting the project, or by first
downloading the source code from the repository, and then using the diff tool
provided by the operating systems. In the first case, it is necessary to download the
SCM and later use the diff tool provided within, but most researchers avoid that
mostly due to the bottleneck represented by the network. In the latter, one has to
download the source code for all the revisions of all the files contained in a software
system. Using a distributed SCM such as Git or Mercurial (instead of a traditional
SCM, as CVS or SVN), the bottleneck of the network is removed and the
 corresponding analysis becomes much faster. As documented in section *Error:
Reference source not found*, this approach still holds some limitations, that have to be
addressed in the threats to validity.

A diff is a summary of the changes undertaken between two files, and stored in a
SCM system. The diff command compares the files line by line and summarizes the
differences in a specific format. Below, the partial output of a unified diff format
between two commits (12 and 13) in the comm-central repository is shown. This
example is not specific from the source code since this is a special file to build the project, however it is simple enough to be easily understood.

diff -r f1...ld -r 0b...f7 suite/build.mk
@@ -43,6 +43,10 @@
 TIERS += app
 +ifdef MOZ_COMPOSER
 +tier_app_dirs += editor/ui
 +endif
 +
 ifdef MOZ_CALENDAR
 MOZ_EXTENSIONS += webdav
 endif

The `hg diff` command, by default, shows the diff between two revisions using the unified format: the diff format starts with two-lines header where the original file name is preceded by −−− and the new file is preceded by +++ After this, there are one or more change hunks (usually named as chunks) which contain information related to the differences in the file. Those lines which were added starts with a "+" character, those removed starts with a "−" character and those which were neither added, nor removed starts with a space character " ". Finally, if a line is modified, this is represented as added and removed, so this changes will appear adjacent to one another. Thus, if a set of lines (all adjacent) are modified, the old revision of the lines will appear starting all of them with a minus character and the new revision of the lines will appear with a plus character. In the previous example, four lines have been added in a file called “build.mk”. The values between “@@” represents the position of those lines in that file before and after the change). For more information it is recommended to read the reference [Error: Reference source not found].

4.2 Retrieving Information from `diff` Files

A freely available tool has been used to retrieve information from each pair of revisions. BlameMe3 consists of several steps that are specified in the following bullets:

- *Downloading the SCM*: the BlameMe tool is specifically prepared to work with Git or Mercurial repositories. These are distributed SCM and provide all

3http://git.libresoft.es/
of the history locally. This is an advantage if compared to other centralized
SCMs such as CVS or SVN since there is an actual and huge difference in
terms of time (avoiding the bottleneck represented by the network access).
Thus, in first place, the SCM is downloaded.

- **Collecting Commits:** As seen above, the very `hg` command provides a special
  command to check differences between two revisions: `hg diff`. This has been
  used to interact between the program and their Mercurial repository.

- **Parsing the several revisions:** secondly, the tool is launched using that
  previously downloaded repository and storing all of the differences in a
  MySQL database. For this purpose, each of the lines is stored in memory
together with its reference to its file and the position in that file (specifically,
there is a list per file, and each node is the position in that file for a given
line). If a new set of lines are detected to be added or removed, those are
directly added in the specified position (explained in section Error: Reference
source not found.

### 4.3 Case Study

The proposed method has been applied to describe the bug fixing process at the level
of source lines using the `comm-central` project and its Mercurial repository. As
mentioned, this repository contains the source code of Thunderbird, SeaMonkey,
Lightning extension and Sunbird.4

The use of the Mercurial repository (after the migration from CVS) started on the
22th of July, 2008 and it has been studied till the 20th of July, 2010 (i.e., two years of
source code history). Considering the whole life of the project until the start of this
study, 5,982 commits were studied and the differences between revisions have been
stored in a MySQL database. In this database, we have stored information from
4,973,038 changes to the source code among added, modified and removed lines.

The case study presented in this paper is based on the study of the data retrieved
from the differences between two revisions of the source code, but filtered down to
those commits detected as a bug fixing process. Thus, the total amount of lines taking
into account just those commits detected as fixing process, are 2,912,866 and 2,969
out of 5,982 commits.

### 5 Results

This section provides the results of the empirical study performed on the comm-
central repository. First, the study of how to properly detect bug-fixing commits is
reported, detailing on the precision and recall in such process. Second, the issue of
dealing with large commits is presented and addressed. Third, the approach of
detecting bug-fixing and bug-seeding committers is clustered in several scenarios, and
finally the results on each scenarios are proposed.

---

4However, as addressed in
https://developer.mozilla.org/en/comm-central,
this only includes a subset of the code required to build those projects.
5.1 Identifying bug-fixing commits

This first part of the research aims to validate the log messages provided by the comm-central community, and to understand the consistency and reliability of their records with regards to bug-fixing activities. To achieve this purpose, we developed an empirical approach and then checked how many false positives and false negatives we obtained from applying it. The approach used is as follows: at first, all the commit messages containing the word “bug” or “Bug” were checked out from the Mercurial SCM; for a random set of 100 of such messages, the log message was inspected, and it was checked manually if these are actually bugs or not, either checking the underlying source code or by parsing the relative Bug Tracking System. The results for evaluating the precision and recall of such approximation, and its constituent parts are as follows:

(TP) True positive: 78
(FP) False positive: 7
(TN) True negatives: 6
(FN) False negatives: 9
Total commits: 100

Therefore we evaluated:
• Positive predictive value: \( \frac{TP}{TP+FP} = \frac{78}{78+7} = 91.7\% \)
• Negative predictive value: \( \frac{TN}{FN+TN} = 40\% \)
• Sensitivity = \( \frac{TP}{TP+FN} = \frac{78}{78+9} = 89.65\% \)
• Specificity = \( \frac{TN}{FP+TN} = \frac{6}{7+6} = 46.15\% \)

Since the Precision actually coincides with the positive predictive, and the Recall coincides with the sensitivity, we conclude that precision=91.7% and recall=89.65%.

5.2 Dealing with very large commits

As reported in previous studies, software systems, and most noticeably FLOSS systems, display at times high (and isolated) peaks of activity. In some specific cases, it has been possible to detect a very large amount of source lines (e.g., more of 80% of the overall system) being moved within FLOSS projects. This means that in some changes, one can detect huge changes reaching million of lines. From a maintenance or evolutionary point of view, this is hardly accountable as a maintenance activity. However, this problem has not been taken into account by [Error: Reference source not found], whose analysis is one of the pillars for this study.

Also in the study of the comm-central repository, it has been found that a small number of commits (no more than 10% of the total set) handles several thousands (in some cases hundreds of thousands) of lines in just one commit. Apart from exceptional cases where developers indeed modified a vast amount of source lines, the peaks could also be caused by automatic bots, changes in the licenses, or by accidental removal and addition of source code. As an example of such distortions, figure shows the number of aggregated number of

11
removed lines. The figure depicts a situation of common removal of lines, but in some specific commits, we can see how suddenly a large set of lines is removed (for example, close to id 723 or 4,200).

In order to deal with such distortions, we have simply removed the commits fully or partially affected by those changes: given an overall number of 2,912,866 lines and 2,969 commits detected in the bug fixing process, the sample was therefore reduced to 731,941 lines and 1,747 commits. In summary, the four largest commits (0; 1,002; 817; 5,213 and 5,383) and those whose lines were previously added, removed or modified in one of those commits, were removed from the overall sample.

Figure 3: Aggregated number of removed lines detected in bug-fixing commits

5.3 Identifying bug-fixing and bug-seeding committers

In order to detect the bug-fixing committers, and the developers dealing in the past with the same section of code (as per the scenarios in Figure Error: Reference source not found), this paper uses the same assumption formulated in [Error: Reference source not found]: in a bug-fixing commit, one has to consider only the “set of lines” removed or modified in that commit (see Figure Error: Reference source not found), instead of the whole file, or set of files, committed in the transaction.

This algorithm is named as the “SZZ algorithm” and fully detailed in [Error: Reference source not found]: considering the set of lines modified in a bug-fixing commit, the algorithm focuses on the previous commit in time (i.e., “one step back”) where each of the lines in the set was modified: in this way, it is possible to obtain the latest commit where each line was previously modified (Figure Error: Reference source not found), and correlating it with their actual committer (Figure Error: Reference source not found).

This figure only shows those commits where at least one line was removed.

It should be noticed that the commits listed here are real commits, while the aforementioned, 723 or 4,200 are ids and they do not correspond to real revision numbers in the SCM.
not found). The assumption of the algorithm, also used in this paper, is that the bug was actually introduced in that previous commit.

In order to visualise at first the summary of results, Figure Error: Reference source not found shows, for each bug-fixing commit, how many developers were previously changing the set of lines involved in the fix (i.e., “bug-seeding” these lines). The results show that 1,035 bug-fixing commits (60% of the overall sample) have only one previous developer for the whole set of changes that took place in that commit. Considering two previous developers, we are covering 80% of the total commits of the sample (1,392 out of 1,747 commits overall). In both cases, this does not specify whether the committer of the bug-fixing commit is also among the developers of the previous change. This first sample represents more than a 50% of the total commits, but with respect to the number of lines, it only represents some 7% of the total lines (50k out of 747k lines).

Figure 4: Number of committers involved in changes at (t-1) to lines bug-fixed at t

Thus, based on this initial set of results, commits detected as a bug-fixing changes are divided in several scenarios (S), depending on the number of different people introducing the bug, and who they are. Since the most numerous sets are the one with just one or at most two previous developers, we have specifically focused on them. Hence, three main sets are found: those with one, two or more previous developers. These sets are again divided depending if the developer who fixed the bug also participated in its introduction. This provides a final list of six scenarios:

- **S1** bug-fixing and bug-seeding commits made by committer A;
- **S2** bug-fixing commit made by A, bug-seeding commit made by B;
- **S3** bug-fixing commit made by A, bug-seeding commit made by A and B only;
- **S4** bug-fixing commit made by A, bug-seeding commit made by B and C only;
- **S5** bug-fixing commit made by A, bug-seeding commit made by A, B and others;
- **S6** bug-fixing commit made by A, bug-seeding commit made by B, C and others;
5.4 Analysis of Scenarios

Scenarios S1 and S2

Table Error: Reference source not found focuses the analysis on those commits whose number of previous committers detected is exactly one, accounting for some 60% of all bug-fixing commits (and corresponding to scenarios S1 and S2). The summary in the table distinguishes whether the bug-fixing commit is the same committer (Same Comm. column, e.g., scenario 1) or a different one (Diff. Dev. column, e.g., scenario S2) acting on the previous set of changes. In terms of number of people involved, the bug fixing process is mostly performed by different committers from those introducing the bug: only in 6% of such subset (62 out of 1,035 commits) the bug-fixer is the only one involved in the previous commit of that set of lines (e.g., scenario S1).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Commits</td>
<td>1,035</td>
<td>62</td>
<td>973</td>
</tr>
<tr>
<td>Lines</td>
<td>50,078</td>
<td>973</td>
<td>49,348</td>
</tr>
</tbody>
</table>

Table 1: One previous committer

Scenarios S3 and S4

When considering a maximum of two committers committing changes to the same set of lines affected by bug-fixing commits, it is found that only 357 commits comply with the requirement of Scenarios S3 and S4. Table Error: Reference source not found shows the results differentiated for Scenario S3 and S4: 43 out of 357 commits have been participated fully or partially by the same committer who finally fixed the bug. In terms of lines handled, 6,834 lines were co-changed with another committer and submitted by the same committer A, while 218 were co-changed with A but committed by another committer B. These results provide another point of view of the community: generally speaking, it seems that most of the commits where two people have previously participated were mostly handled by people different from those who fixed the bug. However, in 43 commits, the same committer was found to participate in the changes.

This raises another question related to the quantity of source code handled by those committers, when the same committer has also participated. In that case, we realized that just a 3% of that source code was really handled by someone different (inside those 43 commits). This shows a pretty close relation with the first set of study, where just one committer was found.
The other scenarios presented here comprise the commits with up to 10 previous committers handling the source code. Albeit more committers could be possible, the threshold of 10 committers reaches 98% of the total sample analyzed (1,717 out of 1,747 commits). Figures (top and bottom) show the absolute and relative number of commits. In Figure (top) the y-axis are divided by the number of previous developers involved in the set of lines that in the current commit were modified or removed. The x-axis represents the number of absolute commits detected. We can see how figure (top) shows that most of the commits were previously handled by people totally different from the ones who were later dealing with the bug-fixing commit. Figure (bottom) is useful to check the relative percentages of such values, and to conclude that, in all of the cases, more than a 60% of the total bug-seeding commits had a different committer than the one who made the bug-fixing commit. Using relative numbers, we can see how in eight out of ten occasions the second set of data (those commits where the same person who fixed the bug did not handled those lines at all previously) is the most general.

**Table 2: Commits: two previous committers**

<table>
<thead>
<tr>
<th></th>
<th>Overall</th>
<th>S3 (A+B)</th>
<th>S4 (B+C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commits</td>
<td>357</td>
<td>43</td>
<td>314</td>
</tr>
<tr>
<td>Lines</td>
<td>15,581</td>
<td>7,052 (6,834 + 218)</td>
<td>8,529</td>
</tr>
</tbody>
</table>

**Scenarios S5 and S6**

Figure 5: Scenarios and their relevance – *Same Committer* refers to S5; *Fully Diff. Committers* refers to S6

15
At an even finer granularity, figure Error: Reference source not found displays all the above Scenarios: depending on the number of bug-seeding committers, this figure shows the number of lines handled by each of the defined group of committers. With the notation “Rest of the commits”, the figure shows scenarios S2, S4 or S6, respectively. With “Same commit, Same Committers”, the figure shows Scenario S1, i.e., how many lines were previously changed by the bug-fixing committer. With “Same commit, Diff Committers”, the figure shows the Scenarios S1, S3 and S5, but discarding the lines previously modified by the bug-fixing committer.

![Figure 6: Scenarios and their relevance – Lines affected](image)

**Discussion**

The analysis of the Mozilla community, and the comm-central project has shown some interesting results: given the specificity of this community, and the process that were put in place at the maintenance level, generalising such results could be problematic. Nonetheless these observations provide an initial set of results to characterise the bug-fixing and bug-seeding activities in the Mozilla community.

The first result has shown that some 50% of the commits have been detected as fixing bugs, which is quite an impressive amount, even considering that the precision in detecting bug-fixing commits proved substantially high. Nonetheless, this value is largely dependent on the policy applied by the Mozilla community when submitting changes to the source code: this policy alone could lead to an overrating or underrating of the results based on this issue.

Regarding the bug-seeding activity, it was found that each piece of source code modified in a bug-fixing commit was previously modified mostly by one developer; furthermore it was also found that a different developer is in charge of such fix, other than the one who seeded the issue. From a managerial perspective, this result shows that with a high probability people are usually fixing bugs that were introduced by other developers. These probably are developers with a specific role in terms of bug-fixing.

Furthermore, focusing on the Scenarios 3 or 4, there is a 80% of probability that a selected bug-fixing commit was introduced by at most two developers. This again shows that in most of the cases, the pieces of source code are by definition a valuable piece of knowledge. Some authors have dealt with the idea of concepts when developing software, and it seems that working at the level of methods or functions is
the best way to understand previous changes made by others. A possibility here is to
match pieces of source code and methods to check this hypothesis.

Finally, at the granularity of commits, Scenarios 5 and 6 have shown that in few
cases the piece of source code was handled by more than two committers. However, it
should be studied why there are commits where their bug-seeding commits where
handled by in some cases dozens of committers. A possible hypothesis could related
to huge movements of lines in a bug-fixing commit: in this context it could be found
that for a given commit, several people previously participated (even dozens of them)
in such a bug-seeding commit.

With respect to lines, results show that, in case of two previous committers, the
same developer is mostly involved (as seen in table). It could be argued that, when the same developer is seeding and fixing a bug, she will
also handle most of the lines of that piece of source code: developers will work on the
known piece of code, and there are some residual lines from others.

6 Threats to Validity

Generally speaking, any empirical study like this is bound to many threats to their
validity. In the authors claim that studying FLOSS projects from an empirical point of view could raise several threats that should be considered. Among
them, we can find those related to the data extraction, the granularity of the study or how mature is the selected projects.

Construct Validity

At first, the heuristic used to obtain bugs from the SCM log messages is far from be
perfect. As seen in the selection of bugs (even for those projects studied in the Mozilla community) are based in a corpus and some other semantic data which improve the data obtained. Also, as addressed by analyzing the SCM logs could be error prone. However,
after manually checking 100 commits with the heuristic used, the percentages of error
were very low. This is due to the selection of a project from the Mozilla community
which generally shows good practices by precisely pointing to the bug tracking system
for almost each change in the source code.

Second, most of the work is based on the analysis by the tool provided by the
Mercurial SCM. Although this is a reliable tool, we have detected some limitations in
the use of diff to retrieve the authorship and other related data. As addressed by the, we could obtain wrong indications in the number of actual changes in the source code after a commit. One of
the main reasons for those changes could be some movements of data from one
directory to another, or some merges from different branches. In order to deal with
them, most of the big spikes, as aforementioned, were removed.

Finally, it is worth to mention that the large additions of lines are an issue which
has not been resolved in this paper. Future revisions following a commit affecting
thousands of lines may lead to the wrong conclusions, by showing that most of the
work was done by just one committer, although this could be just a distortion of few
commits.
Internal Validity

The tools and script used could present some minor bugs that may affect the results. However, they have followed a validation process what makes the results reliable enough. After the initial development and after several tests, a final manual study of several commits was carried out and in all of the cases the comparison between the information in the database and the SCM matched in a 100% of the cases. However, the tools used could raise some errors in the future that could not have been taken into account yet.

External Validity

The selected project is not large enough to represent the overall number of FLOSS projects. However, we present a first initial step to describe the bug-fixing process based on the Mercurial SCM. As further work, the authors want to extend the analysis at least to the whole Mozilla community.

7 Conclusion and Further Work

This paper has presented an empirical analysis of the comm-central FLOSS community, in order to detect whether the bug-fixing activity among developers could be modeled into patterns and recurring scenarios. This community was selected for the consistency and reliability of their messages into the SCM repository, in particular the messages dealing with the bug-fixing activities. With these characteristics, this community and their data can be leveraged to shed important hints on how FLOSS developers proceed to the very needed corrective maintenance, and more importantly, whether the bug-fixing committers are the same who contributed to introduce and seed the bug in the first place.

As a first conclusion, we could confirm the reliability and consistency in referencing the bug-fixing commits within the comm-central community, with a precision larger than 90%: this produces very safe results in terms of tracking the actual bug-fixing committers, and the lines that were modified in the process. It also forms a basis of good practices that will be leveraged in future works when studying the larger Mozilla community (an order of magnitude larger in terms of activity and committers).

Secondly, the proposed method defines and tracks both the bug-fixing and the bug-seeding committers: given the set of lines affected by the bug-fixing commit, the set of previous revisions was studied in order to detect which committers were actually “seeding” such bug without contributing to its removal or fixing.

In third place, the comm-central community produces sporadic peaks of distorted activity logs [Error: Reference source not found, Error: Reference source not found, Error: Reference source not found]. However, this problem has not been raised by [Error: Reference source not found]: this issue should be taken into account when analysis FLOSS projects in order to avoid possible misinterpretation of the results (at times, the whole code base is removed from the repository, just to be added a few hours later, thus distorting the change sets of the vast majority of lines of the project). What was done to take into account this issue was to remove the five largest commits, which alone were responsible of over 2M lines modified, added or removed. Thus, we have
at least removed from the dataset three main cases: 1- Initial import of commits, 2-
huge removal and addition of commits, 3- huge addition and removal of commits. In
those cases, the results could be directly influenced.

In fourth place, using the **diff** provided by the SCM as a way to let us know
authorship at the granularity of a line: other works such as [Error: Reference source not
found, Error: Reference source not found, Error: Reference source not found] have used another
different approach to deal with the idea of following the life of a line. Several
difficulties appear when trying to track the whole life of them, but not at the level of
going a step back in their history. Thus, using this tool could be a faster and more
effective way of determining the authorship of each line.

With respect to the results, it was shown how the corrective maintenance is being
carried out by people on the **comm-central** community. We have detected that only
less than 5% of the bug-fixing commits were handled by who first introduced the
changes or “seeded” the bug. With respect to these results, in most of the cases the
committers involved in the bug-fixing process are not the same as those initially
seeding the bug or the issue. These results are vastly different and unexpected if
compared with corporate software development, where developers “opening” a bug
are most likely to also be responsible for its fixing and closure.

As further work, the authors would like to address some open questions (related
to the GQM approach) that could be easily answered using this same dataset. First of
all, in this paper, the central idea is related to the fixing process and if the committers
are fixing their own bugs. However, we have not studied if those committers are aware
that in some cases they have been introducing errors in the source code, or at least the
seed of a future bug. Checking how many of them have been working in a given
window-time after the detection of a bug could provide another insight of the bug
fixing process.

Another similar idea is related to the seed of the bug. We have seen how given a
commit fixing a bug we could trace when the involved source code was previously
added or modified and, thus, who was the “bug-seeder”. However we do not fully
understand the causes. For instance, we could trace if that developer modified a piece
of source where she usually does not work, if the commit modified a file that was
lately several times modified, if a committer submitted a change in a programming
language not usual to her or some other possibilities.

**Acknowledgment**

The authors would like to thank Prof Cornelia Boldyreff for the extensive comments
on the paper. In addition, this work has been partially funded by the European
Commission, under the ALERT project (ICT-258098).

**References**

Coordination and productivity issues in free software: The role of brooks’ law.
In *ICSM*, pages 319–328. IEEE.

goal/question/metric paradigm. Technical report, College Park, MD, USA.


