Who introduced this bug?
It may not have been caused by the previous commit!

Gema Rodríguez
GSyC/LibreSoft
Universidad Rey Juan Carlos
Madrid, Spain
gerope@libresoft.es

Jesús M.
González-Barahona
GSyC/LibreSoft
Universidad Rey Juan Carlos
Madrid, Spain
jgb@gsyc.es

Gregorio Robles
GSyC/LibreSoft
Universidad Rey Juan Carlos
Madrid, Spain
grex@gsyc.urjc.es

ABSTRACT

It is common practice that developers mark in the versioning system when they are fixing a software bug. At first glance, it could seem reasonable to assume that the fixed bug had been introduced in the previous modification of those same parts of the source code (i.e., in the previous commit). In fact, many studies on bug seeding start with this assumption. However, there is little empirical evidence supporting this assumption, and there are reasons to suppose that in some cases the bug may have been introduced by other actions, such as an older modification, or a change in the API that is being called.

This paper tries to shed some light on this topic, by analyzing the relationship of bug fixes with their previous commits. To this end, we conducted an observational study on bug reports, their fixes, and their corresponding previous commits for the OpenStack project. Our results show that the assumption that bugs have been introduced in the previous commit does not hold for a large fraction of the bugs analyzed. *Gregorio says: maybe we could say something more here. Gema says: podriamos escribir algunos porcentajes de lo que hemos encontrado, algo asi como que un 37% de los commits previos no son responsables de introducir niingun bug*

Keywords

Bug introduction, bug seeding, SZZ algorithm, previous commit

1. INTRODUCTION

When a failure is found in the behavior of a software, the developers try to fix it locating and modifying the source code line(s) that are reponsible for the wrong behaviour in the source code. It seems reasonable to assume that the previous modification of this line or these lines are the cause of the bug; this previous modification is *the previous commit*. But in fact, to find when and where a bug was introduced

in the source code is not a trivial task. This has largely been ignored, mainly because the data related to the origin of a bug is embedded in the evolution of the software [15]. *Gregorio says: clarificar la frase anterior... no ha quedado muy claro.*

This is the reason why many studies in the area of mining software repositories start with this implicit assumption. So, for instance, we have found following reasonings in several areas of research, such as:

- in bug seeding studies, e.g., *"This earlier change is the one that caused the later fixed"* [18] or *"The lines affected in the process of fixing a bug are the same one that originated or seeded that bug"* [6],

- in bug fix patterns, e.g., *"The version before the bug fix revision is the bug version"* [13],

- and in tools that prevent for future bugs, e.g., *"We assume that a change/commit is buggy if its modifications has been later altered by a bug-fix commit"* [2].

But although the assumption can be found frequently in the research literature, in our opinion there is not empirical evidence supporting it. That is the reason why we have conducted an observational study on fix-bugs, devoting a significant effort to locate the origin of a bug in the source code and understanding the possible causes. For this we had to take into account the moment in which the line was inserted and the general context of the project.

Figure 1 shows a clear example of what we understand as the cause of the bug. Let's assume that we have three different versions of the same file in the history of the control version of the project.

1. The code on the left (subfigure *(1)*) is the one written to fix the bug.

2. The code in the middle (subfigure *(2)*) shows the moment in which the bug was introduced (being *31f08423* the id of change), the previous commit.

3. The code on the right (subfigure *(3)*) ensures that in previous versions of the file, the bug did not exist.

According to the description in the log of the commit that fixed the bug (see Figure 2), commit *31f08423* was the one where the bug was introduced, as it used a variable as string when a Boolean had to be used, keeping the concordance

After Fix Bug		Fix-inducing (Before fix Bug)		Before fix-inducing	
31f208423 711)	if rescue_auto_disk_config is None:	31f208423 711)	if rescue_auto_disk_config is None:	31f208423 701)	if rescue_auto_disk_config is None:
e30b45f69 712)	LOG.debug("auto_disk_config value not found in"	e30b45f69 712)	LOG.debug("auto_disk_config value not found in"	e30b45f69 702)	LOG.debug("auto_disk_config value not found in"
e30b45f69 713)	"rescue image_properties. Setting value to %s",	e30b45f69 713)	"rescue image_properties. Setting value to %s",	e30b45f69 703)	"rescue image_properties. Setting value to %s",
e30b45f69 714)	auto_disk_config, instance=instance)	e30b45f69 714)	auto_disk_config, instance=instance)	e30b45f69 704)	auto_disk_config, instance=instance)
31f208423 715)	else:	31f208423 715)	else:		
20847c25a 716)	auto_disk_config = strutils.bool_from_string(	31f208423 716)	auto_disk_config = rescue_auto_disk_config		
20847c25a 717)	rescue_auto_disk_config)				
	(1)		(2)		(3)

Figure 1: Example of a change in which the bug was introduced in the previous commit. More recent versions of the code are on the left.

```
Xen: convert image auto_disk_config value to bool before compare

During rescue mode the auto_disk_config value is pulled from the rescue
image if provided.  The value is a string but it was being used as a
boolean in an 'if' statement, leading it to be True when it shouldn't
be.  This converts it to a boolean value before comparison.

Change-Id: Ib7ffcab235ead0e770800d33c4c7cff131ca99f5
Closes-bug: 1481078
```

Figure 2: Description of the bug-fix commit for a case in which the previous commit caused the bug.

with the rest of the code. So, in this case, the bug was introduce in the previous commit.

On the other hand, Figure 3 shows a clear example of a case where the cause of the bug is not to be attributed to the previous commit. In this example, the bug fix commit log (see Figure 4) describes that the name of an argument changed when updating the version causing the failure in the software. This change was done because of the new requirements in the software version, and is unrelated to the changes performed in the previous commit. When the modified lines where introduced in the first time, they did not contain the bug.

Based on anecdotal evidence like the one presented in Figures 3 and Figure 4, we argue that in projects that are continuously evolving, with an ample developer community, code that before was correct could be buggy at some time. So, changes in other parts of the code may induce wrong behavior (bugs) in places that were correct in the past. This happens often in situations like changes to the API. In the moment the code ws written, it was correct and the woftware worked fine. Additions of new features or enhancements to the API had as a side effect that the formerly correct code presents a wrong behavior, making the software fail. But in such cases, the source of the error cannot be *Gregorio says: assigned* to a change performed in the previous commit, as in that moment it referred to a different API.

Gregorio says: poner el siguiente parrafo bien

The goal of this paper is to find if the cause of the bug can be *Gregorio says: assigned* to the previous commit, understanding that at the time when the previous commit is introduced

not all times the code analyzed was so easy as the code showed in Figures 1 and 3. So the principal aim of this paper

to find the responsible of the bug is to know if the previous commits contained buggy code at the moment in which were added/modified or, another change in the software, related to the update of the software and its evolution, caused the bug.

In detail, in this paper we attempt to address the following research question regarding who introduced the bug in the source code:

Gregorio says: rephrase RQs

- RQ1: How can we know that a change was done to fix a bug in the source code? How can we identify them?

- RQ2: When did the previous commit introduce the line with the bug into the source code?

Gregorio says: Aqui deberiamos poner la contribucion de nuestro articulo al estado del arte.

The remainder of this paper is structured as follows. Next, we present the current body of knowledge in section 2. Section 3 describes the methodology used to identify the moment in which the bug was introduced in the source code, followed by the results obtained after applying our approach to a selection of OpenStack bug fixes in Section 5. Section 6 answers the research questions and discusses potential applications and improvements of our approach. After reporting the limitations and threats to validity in Section 7, we draw some conclusions and point out some potential future work in Section 8.

2. RELATED WORK

The first algorithm to identifying bug-introducing code changes automatically was proposed by Sliwersky et al. [16]. Currently, it is a well-known algorithm called SZZ, which is based on text differences to discover modified, added and deleted lines between the bug-fix and its previous version. The SZZ algorigltm uses the CVS `annotate` command[1] to identify the last commit that touched these lines.

An improvement to the SZZ algorithm is described by Kim et al. [8]. There the authors used annotation graphs instead of CVS annotation to locate, in the previous versions, the lines affected by modification and deletion. Also, they avoid

---

[1]Other versioning systems provide similar functionality to CVS `anotate`; for instance, git offers `blame`.

After Fix Bug	Fix-inducing (Before fix Bug)	Before fix-inducing
0dc91bed 318) if VERSIONS.active < 3:	0dc91bed 318) if VERSIONS.active < 3:	68a55e3f 303) if VERSIONS.active < 3:
0dc91bed 319) user = manager.create(name, password, email, project, enabled)	0dc91bed 319) user = manager.create(name, password, email, project, enabled)	68a55e3f 304) user = manager.create(name, password, email, enabled)
0dc91bed 320) return VERSIONS.upgrade_v2_user(user)	0dc91bed 320) return VERSIONS.upgrade_v2_user(user)	68a55e3f 305) return VERSIONS.upgrade_v2_user(user)
0dc91bed 321) else:	0dc91bed 321) else:	68a55e3f 306) else:
0dc91bed 322) return manager.create(name, password=password, email=email,	0dc91bed 322) return manager.create(name, password=password, email=email,	68a55e3f 307) return manager.create(name, password=password, email=email,
49f9d154 323) default_project=project, enabled=enabled,	0dc91bed 323) project=project, enabled=enabled,	68a55e3f 308) enabled=enabled)
cbd63f27 324) domain=domain, description=description)		cbd63f27 324) domain=domain, description=description)
(1)	(2)	(3)

Figure 3: Example of a change where the previous commit, 0dc91bed, did not insert the bug. More recent versions of the code are on the left.

Update default_project param on create user

In keystone v3, the parameter to create user for the the default project has changed from project to default_project and is no longer honored and throws an exception. Also passing in '' rather than None causes keystone issues, so moving to None.

Closes-Bug: #1478143
Change-Id: I73423433a42bf46769065a269a3c35f27175f185

Figure 4: Description of the bug-fix commit for a case in which the previous commit did not cause the bug.

some false positives by not considering blank spaces, changes in the format or changes in the comments.

Sinha et al. present another technique to identify the origins of a bug in [15]. Their technique is not text-based technique, as the SZZ algorithm, as the authors analyze the effects of bug-fix changes on program dependencies. So, taking into account the semantics of the source code they achieved higher accuracy in identifying the origins of a bug.

The two approaches have some metodological patterns in common:

1. They find the differences between the bug-fix version and the previous version of the file to recognize those changes done by the bug-fix commit.

2. They look back in the code revision history until they identify which version touched the lines affected in the bug-fix for the last time.

Gregorio says: This last paragraph should be clarified, and more on papers using the SZZ algorithm! Maybe we should state how many papers in total exist based or using SZZ in Google Scholar. Gema says: He buscado en google schoolar las entradas que tiene SZZ algoritm y aparecen 1170 resultados desde 2006 que es cuando sacaron el articulo, pero alguna de las entradas solo contine algorithm y no SZZ algoritm

The SZZ algorithm (and its *successors*) have been widely used in the researc community. Williams et al. revisited the SZZ algorithm to track bug-inducing changes and identify types of changes [18]. Yang et al. applied SZZ to find what kind of bug-inducing changes are likely to become a great threat after being mared as bug-fix changes Finally,

some bug prediction algorithms are based on SZZ; Kim et al. showed how to classify file changes as buggy or clean using change information features and sour code terms [7].

Gregorio says: maybe talk about my paper with dmg and ahmed, where bugs could be found elsewhere [3]. There the talk is precisely about those bugs whose origina are elsewhere. Title of the paper: Change impact graphs: Determining the impact of prior codechanges

Gema says: Hay que incluir lo que nuestro articulo aporta al estado actual Gregorio says: Buena idea, pero eso es mejor ponerlo en la intro.

3. METHODOLOGY

All data needed to analyze when the bug was introduced can be obtained from the issue tracking systems and the code review systems used generally by free/open source software (FOSS) projects. In our analysis, we have focused on Launchpad[2] as issue tracking system, and Gerrit[3] as code review supporting tool, as they are widely used by FOSS projects nowadays, but our methodology should be generalizable to any such tool.

The Launchpad of each project works with issue reports called tickets, which describe bug reports, feature requests, maintenance tickets, and even design discussions. In our study, however, we are only interested in those tickets that have following properties:

1. They describe a bug report, and

2. They have been closed and merged in the code source to fix the described bug.

In these bug reports we can find a comment with the link to Gerrit where the bug was fixed. It is in Gerrit where we can see all the patchsets proposed and the comments done by the reviewers.

3.1 Fist Stage: Filtering

First, we have to identify what issues found in Launchpad are bug reports. This is not a trivial task and is labour intensive as it has to be done manually. As the process is repetitive, we developed a web-based tool[4] that helps in

---

[2]https://launchpad.net/
[3]https://www.gerritcodereview.com/
[4]bugtracking.libresoft.es

the classification process. This tool offers all relevant information required to decide if an issue corresponds to a bug report or not. The tool uses information extracted automatically from the project repositories, and offers a web-based interface which allows for collaboration, traceability and transparency in the identification of bug reports.

During the identification of the issues, we have to take into account the next parameters for each ticket:

- The title of the issue report

- The description of the issue report

- The description of the fix commit

- The changes to the source code, as sometimes neither the descriptions nor the comments by developers in the Launchpad and Gerrit of each ticket, clarified the underlying ticket. *Gregorio says: what comments?*

We can see a screenshot of the web interface of the tool in Figure 5. The left side is used to display the information extracted from Launchpad and Gerrit, and the right part is the one in which the researchers can write and classify the ticket into one of the three groups. Additional meta-data, such as keywords, comments and the reviewer are included in the database.
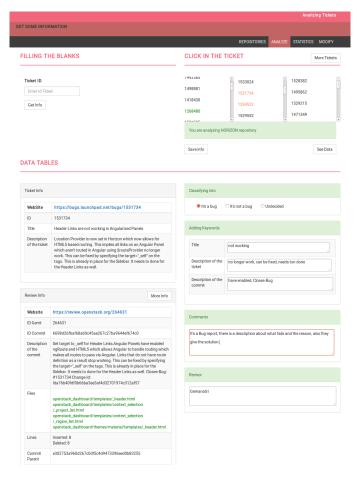

Figure 5: Screenshot of the tool used to classify the tickets.

Each ticket was then categorized into one of three following groups:

1. Group 1 (*Bug Report*): The ticket describes a bug report.

2. Group 2 (*Not Bug Report*): The ticket describes a feature, an optimization code, changes in test files or other not bug reports.

3. Group 3 (*Undecided*): The ticket presents a vague description and cannot be classified without doubts.

From the experience of analysing a small number of tickets, we agreed on following four criteria:

1. When the ticket title described the program as not working as expected, our criteria indicated that it was a bug report.

2. When the title described optimization, deletion of a dead code or the implementation of new characteristics, we agreed not to classify it as a bug report because there is no failure.

3. When the ticket title described that updates were required, the ticket is a bug report. We consider all tickets that require updating as bug reports, because updating a software hints to the software not operating as expected.

4. When only test files are affected in a ticket, we classified it as not being a bug report. Test files in a ticket were not be analyzed, as we consider they are not a core part of the software and are used as a testing method to determine whether the code is fit for use. When the bug exclusively lies in a test file, the ticket was not considered as bug report because the software still works as expected, only test fails. *Gregorio says: no se repiten las frases?*

Sometimes we were unable to answer all the questions due to having insufficient data or because of the complexity of the issue. In this case, the ticket was classified into the *Undecided* group.

3.2 Second Stage: Who caused the Bug?

The next part is focused on analyzing the previous commit exclusively in the *Bug Report* group to identify which line contained the bug and why the software failed, keeping in mind the context of the project. For that, we had to analyze the lines involved in the bug fix and in the *parent* commit of the bug fix commit, being sure that the lines were added,inserted or modified in the previous commit. This way, we can be sure that we are looking the correct change, because some times although the commit added many lines, if you look the code before the commit you can check that some of the lines added was there, and in that case, is a false positive where the previous commit didn't cause the bug. *Gregorio says: aclarar este parrafo!*

The analysis was done manually. We used *git blame* to see the previous commit for each line of the involved file. Also, we used *diff* to see the differences between the two files, in our case as the file is going to be the same, between the file in two different moments in the control version system.

The procedure for each file involved in a bug fix is as follows:

1. git checkout *commit that fixed the bug*, git blame *file involved*. In this step we can see the lines added, modified or deleted by the commit that fixed the bug.

2. git checkout *parent of commit that fix the bug*, git blame *file involved*. In this step we can see the previous commits for the different lines touched in the fixed bug.

3. git checkout *parent of previous commit*, git blame *file involved*. With this step we can ensure that the previous commit inserted these lines.

Finally we need to discard some noise presents in our final results according to the responsibility of the previous commit inserting the bug in the code source. Due to they were not responsible for cause the bug, we delete those previous commit which presents the following criteria:

- Blank lines
- Format changes
- Copied lines
- Changes in the comment.
- Updates in the version of a file.

4. EVALUATION

We have validated our methodology analyzing tickets from OpenStack. OpenStack is a cloud computing platform with a huge developing community (more than 5,000 developers) and significant industrial support from several major companies such as Red Hat, Intel, IBM, HP, etc. OpenStack was particularly of interest because of its continuously evolving due to its very active community. Currently it has more than 233,000 commits with more than 2 million lines of code [5]. All its history is saved and available in a version control system, as well as its issue tracking system (Launchpad[6]) and the source code review system (Gerrit[7]).

OpenStack is composed by 9 projects, but we only focused on the main four of them: Nova, Cinder, Neutron and Horizon. As can be seen from Table 1, these projects have been very active during their entier history, and in the last year.

	All History	Last Year (2015)
Nova	14,558	3,283
Fuel	9,139	5,123
Neutron	8,452	3,855
Horizon	4,871	1,994
Cinder	4,556	1,832
Keystone	4,874	1,795
Heat	6,395	2,372
Glance	2,651	723
Tempest	4,141	1,312

Table 1: Commits per Project in OpenStack

For these four projects we analyzed if bug fixes where introduce in their previous commits. For the first stage, we

[5]http://activity.openstack.org/dash/browser/
[6]https://launchpad.net/openstack
[7]https://review.openstack.org/

	Bug Report	Not Bug Report	Undecided	Total
R1	(184) 55%	(115) 34%	(35) 11%	334
R2	(188) 76%	(54) 22%	(7) 3%	249
R3	(188) 56%	(116) 35%	(30) 9%	334
Agree	(209) 72%	(74) 25%	(9) 3%	292

Table 2: Statistics for each researcher as a result of the classification process. For each researcher R, the number of tickets (and percentages) classified into the three groups is given. The *Agree* row gives the number of tickets (and percentages) were two researchers agreed.

used the tool described in 3.1. Each ticket was analyzed by two researchers independently. The second stage was done manually by the first author.

5. RESULTS

A total of 459 different tickets from the Launchpad of the four main projects in OpenStack: 125 tickets from Nova, 125 tickets from cinder, 125 tickets from Horizon and 84 tickets from Neutron.

5.1 Fist Stage

We classify a total of 459 tickets using the tool, resulting in 918 reviews. Only those tickets classified as bug reports by both researchers were considered in the next stage, which analyzes if the cause of the bug was introduced in their previous commits.

Table 2 shows the classification percentages for each researcher after analyzing the tickets, and the number of tickets classified by two different researchers in the same group. As a result, researchers identified 292 tickets in the same group, that is, their results matched in over 70% of the cases. Of those, 209 tickets had been classified in the *Bug report* group, 74 in the *Not Bug Report* group and 9 tickets classified in the *Undecided* group.

We also measured the concordance in the classification of each developer according to the project analyzed (see table 3). Values obtained by the three researchers are very similar, in general around a 70%. The concordance values were always above 60%.

	Nova	Cinder	Horizon	Neutron	Total
R1 − R2	(44) 70%	(40) 77%	(37) 60%	-	68%
R1 − R3	-	(46) 73%	(48) 76%	(26) 62%	71%
R2 − R3	(41) 66%	(10) 100%	-	-	71%

Table 3: Concordance among researchers for each repository.

> **RQ1: Using all the information available in the bug tracking system and code review systems related to a fix-bug, we have obtained that in at least 72% of the tickets analyzed the fix-bugs were real bug reports.**

5.2 Second Stage

In this stage we analyzed the 189 tickets classified as bug reports and their previous commits. The possible outcome was one of the following three options:

- Cause
- No Cause
- Undecided

This analysis takes into account that the bug could have been inserted in many lines belong to many differnt previous commits, but in fact, not all of them could cause the bug. It could have happened that in the previous commit lines had been copied from further previous commits, or comments may have been modified, or blank spaces/lines have been added. Thus, the cause could be found in a single previous commit, in many or even in none. *Gregorio says: repasar este parrafo*

We identified a total of 348 previous commits which could be the cause of the 189 bug reports under analysis, we analyzed the bug reports together with their previous commits discarting these cases where the previous commit was false positives such as blank lines, changes in comments or even a change in the version of the file., so in total we had analyzed 308 previous commit.

As can be seen in Table 4, from the 308 previous commit, 152 of them have been considered the cause of the bug, whereas 114 were not the cause. We were unable to identify the cause of the bug in 42 cases, due to there was only added lines or our knowledge about the code.

As can be seen in Table 4, in 152 cases the previous commits have been considered the cause of the bug, whereas 114 times they were not the cause. We were unable to identify the cause of the bug in 42 cases, and discarded 40 more because they ere false positives such as blank lines, changes in comments or even a change in the version of the file. *Gregorio says: explicar que es eso de noise! Gema says: Con Noise me referia a los falsos positivos que puede haber en los commits previos, las lineas copiadas de otros commits o lineas en blanco, comentarios ...*

	Before Deleting Noise	After Deleting Noise
Cause	(152) 44%	(152) 49%
Not Cause	(154) 44%	(114) 37%
Undecided	(42) 12%	(42) 14%

Table 4: Number of times (and percentage) where the previous commit is the cause, not the cause or could not be classified, before and after deleting noise.

If we attend to how many previous commits each of the 189 bug reports analyzed had, we see that 131 only had a previous commit, whereas 58 had more than one previous commit. As can be seen from Table 5, from the 131 unique previous commits, 65 were the cause of the bug, and 30 not caused the failure. For the 58 bugs that had more than one previous commit, a total number of 179 previous commits were identified; of them 86 were the cause of the bug, while 82 were not.

Gema says: Los resultados que he obtenido y que queria plasmar en las tablas son: Gema says: - He analizado 189

	One previous commit	More than one previous commit
Cause	(65)50%	(86)48%
Not cause	(30)23%	(82)46%
Undecided	(36)27%	(11)6%

Table 5: Probability of the cause of the bug when the bug report present one previous commit or more than one previous commit.

bugs reports
Gema says: - De los 189, 131 presentaban el mismo identificador de commit previo para todas las lineas que habian sido modificadas, mientras que 58 presentaban mas de un identificador de commit previo.
Gema says: - Dentro de los bugs reports que tienen mas de un commit previo analizando el total de todos los commits para saber quien causo el bug, he encontrado que en total de los 179 comit previos 86 causaron el bug mientarsa que 82 no lo causaron y 11 no he podido decidir si eran causantes o no.
Gema says: - Ademas he mirado como o porse distribuyen los commits en cada bug report dentro de cada proyecto analizado, y he visto que por ejemplo Neutron presentaba 11 bugs report con solo un ticket previo, 3 bugs report con dos identificadores diferentes para los commit previos, 2 bug reports que presentaban 3 commit previos... etc.
Gema says: - Finalmente realice ora clasificacion en la que tenia en cuenta el numero de commits que fueron causantes del error para un mismo bug report. Al analizar los bug reports encontre que en aquellos que presenstaban mas de un commit previo no todos los commit eran causantes del error, y en ocasiones solamente habia un commit que causo el error y los otros no o por el contrario habia bugs report en los que ninguno de sus commits previos era responsable y otras ocasiones en las que todos los commits previos eran responsables. Por tanto la tabla 7 muestra cual es la responsabilidad (tods responsables, al menos uno ningun responsable y desconozco la responsabilidad) de los coommits previos presentes en un bug report. Por ejemplo, bugs reports que tenian dos commits previos, en 4 de ellos los dos commits previous eran responsables, mientras que en 9 bug report habia al menos un responsable y en otros 4 bug report ninguno de los dos commit previos eran responsables. We also studied the distribution of the number of previous commits for each Bug Report. As shown in Table 6, the number of commits that can be considered as previous is 1, followed by 2 commits.

Finally, we were interested in analyzing, for those cases where more than one previous commit exist, how many of them inserted the bug in the code source. Results are given in Table **??**; in 8 bug reports all the previous commits were identified as the cause, in 30 bug reports at least one of the previous commits caused the bug, and in 11 bug reports none of the previous commits introduced the bug.

> **RQ2: Only 50% of the previous commits analyzed caused the failure in the system, whereas the 37% of them did not introduce the bug in the code source.**

	One previous commit	two previous commit	three previous commit	four previous commit	+five previous commit
Neutron	11	3	2	2	0
Horizon	39	8	3	2	4
Nova	44	5	2	4	4
Cinder	37	9	6	2	2
Total	131	25	13	10	10

Table 6: Distribution of number of previous commit per Bug Report in each project

	two previous commit	three previous commit	four previous commit	+five previous commit	Total
All Responsible	4	3	0	1	8
At least one responsible	9	7	5	9	30
None Responsible	4	2	4	1	11
Undecided	1	0	1	0	2

Table 7: Number of previous commit responsibles per bug report

6. DISCUSSION

Gema says: Deberia hablar sobre los siguientes puntos: Gregorio says: Puedes elaborar un poco mas cada punto, aunque sea en castellano?

1. No todos los casos son tan claros como los mostrados en los ejemplos, hay ocasiones en las que solo se ha anadido codigo y no somos capaces de encontrar al responsable por que no existe ninguna manera de poder identificar al commit previo.

2. Hemos sido conservadores, en ocasiones cuando se han modificado lineas como if/else y se han anadido comprobacione que anteriormente no estaban, se puede pensar que el commit previo que toco la linea se olvido de realizar dichas comporbaciones, y en ese caso seria el causante del bug. Pero existen casos que la comprobacion que se necesita en el if ha aparecido porque se introdujo una nueva funcionalidad, por stanto en el momento original en el que se escribio la linea por primera vez no habia ningun error. Por tanto a menos que se descrisba esta ultima situacion en la descripcion del commit o del bug report, he considerado la idea treadicional, que el commit anterior provoco el error, porque realmente era dificil saberlo.

3. Hemos utilizado la herramienta porque es un proceso complicado. Para la primera parte de decidir si es o no es un bug report hemos tenido que desarrollar la herramienta.

Gema says: Algunos de los problemas que nos hemos encontrado durante el analisis y que podrian ser resueltos de otra forma diferente a la nuestra y quizas puede que varie algo los resultados no lo se Once we have all the tickets analyzed by diferents researchers who have used a double blind, how to proceed if there are discordances between them:

1. Should they discuss after their analysis to reach a better classification?, Should the tool provide this?

2. Does the Bug report only the same ticket classified as Bug report for all the researchers?

How to proceed if looking for the responsibility of a bug when only added lines are inserted? And we are talking about a bug report not a new feature, these kinds of cases use to be when a researcher forgot check some case inside a function.

1. Is responsible the function where these lines are content?

2. Is responsible the last commit that modify something in the function?

Gema says: anadir ademas que los resultados optenidos presentan una oposicion a la idea actual de que el previo commit introdujo el error

7. THREATS TO VALIDITY

The size of the tickets extracted form the Launchpad is medium, but doing the analysis manually we are sure that the results present in this paper are valid, being sure that the previous commits classified as not responsible, they are.

Although, we understand that the model presented has some threats, external and internal, that make our model not 100% valid. The internal threats related to the researchers that have conducted the study are following:

- We have not taken into account errors that have been classified into *Undecided*, and probably we are lost some real bug reports belonging this group .

- There could be some lax criteria involving the subjective opinion of the researchers.

- The researchers are not experts in OpenStack, and our inexperience may have influenced the results of the analysis.

- We are only using part of the information that the ticket provides, like comments and text. There could

be a recognized pattern in the data, unknown at first sight, that involves other parts of the information.

- Although we use a random script to extract the tickets reported in during the last year, 2015, from the launchpad, in this year could be some bias unidentified.

- In case of the researchers didn't find the information to know if the previous commit inserted the bug or in contrast, it was caused by the evolution of the software. They keep the traditionalist thought, classify these previous commits are responsible.

The external threats, related to the case of the project, are following:

- The word *bug* is continuously mentioned in the description and commit of a ticket even when we found it is not an error. This could lead to the incorrect classification during the reviewing process.

- Some tickets are not explicitly described, which could increase the percentage of *Undecided*. This is especially true if the reviewers are not from OpenStack.

- OpenStack is a special project put down a constant evolution due to their active community of developers. Maybe, in other projects with less commits per year the statistics about the responsibility of previous commit change.

Gema says: Incluir un link con la web en la que pondre todos los datos y scripts utilizados para los datos.

8. CONCLUSIONS AND FUTURE WORK

The empirical experiment carried out in OpenStack, supported that the current premise assumed does not hold for a large fraction of the analyzed bugs, because around the 40% of the previous commits were not responsible inserting the bug. With our results we can identify which ones are real changes that introduced the bug, and this could be useful to improve the accuracy of those tools developed to prevent bugs. Also, the software developers stand to benefit from identifying where the bug was inserted, improving their methodology.

A final field in our future work concerns the full automation on the methodology could developer an automatic classifier base on the idea that not all the previous commit injected the bug. Another interesting investigation could perform the same empirical study in a project with a community less active, to can prove if our idea is fulfill in other projects.

9. ACKNOWLEDGMENTS

---

[8]http://bitergia.com/

10. REFERENCES

[1] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM, 2010.

[2] M. Fejzer, M. Wojtyna, M. Burzańska, P. Wiśniewski, and K. Stencel. Supporting code review by automatic detection of potentially buggy changes. In *Beyond Databases, Architectures and Structures*, pages 473–482. Springer, 2015.

[3] D. M. German, A. E. Hassan, and G. Robles. Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology*, 51(10):1394–1408, 2009.

[4] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401. IEEE Press, 2013.

[5] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108. ACM, 2008.

[6] D. Izquierdo-Cortazar, A. Capiluppi, and J. M. Gonzalez-Barahona. Are developers fixing their own bugs?: Tracing bug-fixing and bug-seeding committers. *International Journal of Open Source Software and Processes (IJOSSP)*, 3(2):23–42, 2011.

[7] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, 2008.

[8] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[9] S. Koch. *Free/open source software development*. Igi Global, 2005.

[10] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd., 2003.

[11] E. W. Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[12] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Multi-layered approach for recovering links between bug reports and fixes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 63. ACM, 2012.

[13] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[14] L. Prechelt and A. Pepper. Why software repositories are not used for defect-insertion circumstance analysis more often: A case study. *Information and Software Technology*, 56(10):1377–1389, 2014.

[15] V. S. Sinha, S. Sinha, and S. Rao. Buginnings: identifying the origins of a bug. In *Proceedings of the 3rd India software engineering conference*, pages 3–12.

ACM, 2010.

[16] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[17] E. Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1):100–118, 1985.

[18] C. Williams and J. Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM, 2008.

[19] 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*, pages 15–25. ACM, 2011.

[20] H. Yang, C. Wang, Q. Shi, Y. Feng, and Z. Chen. Bug inducing analysis to prevent fault prone bug fixes. In *Proceedings of the Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE 2014)*, pages 620–625, 2014.

[21] 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*, pages 26–36. ACM, 2011.

[22] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead Jr. Mining version archives for co-changed lines. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 72–75. ACM, 2006.

[23] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.