How Do Fixes Become Bugs?

A Comprehensive Characteristic Study on Incorrect Fixes in Commercial and Open Source Operating Systems

Zuoning Yin‡, Ding Yuan‡, Yuanyuan Zhou†, Shankar Pasupathy*, Lakshmi Bairavasundaram*
‡Department of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{zyin2, dyuan3}@cs.uiuc.edu
†Department of Computer Science and Engineering, Univ. of California, San Diego, La Jolla , CA 92093, USA
yyzhou@cs.ucsd.edu
*NetApp Inc., Sunnyvale, CA 94089, USA
{pshankar, lakshmib}@netapp.com

ABSTRACT

Software bugs affect system reliability. When a bug is exposed in the field, developers need to fix them. Unfortunately, the bug-fixing process can also introduce errors, which leads to buggy patches that further aggravate the damage to end users and erode software vendors' reputation.

This paper presents a comprehensive characteristic study on incorrect bug-fixes from large operating system code bases including Linux, OpenSolaris, FreeBSD and also a mature *commercial* OS developed and evolved over the last *12* years, investigating not only the mistake patterns during bug-fixing but also the possible *human reasons* in the development process when these incorrect bug-fixes were introduced. Our major findings include: (1) at least 14.8%∼24.4% of sampled fixes for post-release bugs [1] in these large OSes are incorrect and have made impacts to end users. (2) Among several common bug types, concurrency bugs are the most difficult to fix correctly: 39% of concurrency bug fixes are incorrect. (3) Developers and reviewers for incorrect fixes usually do not have enough knowledge about the involved code. For example, 27% of the incorrect fixes are made by developers who have never touched the source code files associated with the fix. Our results provide useful guidelines to design new tools and also to improve the development process. Based on our findings, the commercial software vendor whose OS code we evaluated is building a tool to improve the bug fixing and code reviewing process.

Categories and Subject Descriptors: D.2.0 [Software Engineering]: General

General Terms: Reliability

[1]These only include those fixes for bugs discovered after software releases.

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

Keywords: Incorrect fixes, software bugs, bug fixing, human factor, testing

1. INTRODUCTION

1.1 Motivation

As a man-made artifact, software suffers from various errors, referred to as software bugs, which cause crashes, hangs or incorrect results and significantly threaten not only the reliability but also the security of computer systems. Bugs are detected either during testing before release or in the field by customers post-release. Once a bug is discovered, developers usually need to fix it. In particular, for bugs that have direct, severe impact on customers, vendors usually make releasing timely patches the highest priority in order to minimize the amount of system down time.

Unfortunately, fixes to bugs are not bullet proof since they are also written by human. Some fixes either do not fix the problem completely or even introduce new problems. For example, in April 2010, McAfee released a patch which incorrectly identified a critical Windows system file as a virus [8]. As a result, after applying this patch, thousands of systems refused to boot properly, had lost their network connections, or both. In 2005, Trend Micro also released a buggy patch which introduced severe performance degradation [22]. The company received over 370,000 calls from customers about this issue and eventually spent more than $8 million to compensate customers. The above two incidents are not the only cases in recent history. As a matter of fact, there were many other similar events [2, 15, 4] in the past which put the names of big companies such as Microsoft, Apple and Intel under spotlight.

We had also conducted a study on *every* security patch released by Microsoft in its security bulletin [1] since January 2000 to April 2010. Surprisingly, out of the total 720 released security patches, 72 of them were buggy when they were first released. These patches were expected to fix some severe problems. Once released, they were usually applied to millions of users automatically. Therefore, they would have enormous impacts and damages to end users as well as software vendors' reputation.

Mistakes in bug fixes may be caused by many possible reasons. First, bug fixing is usually under very tight time schedule, typically with deadlines in days or even hours,

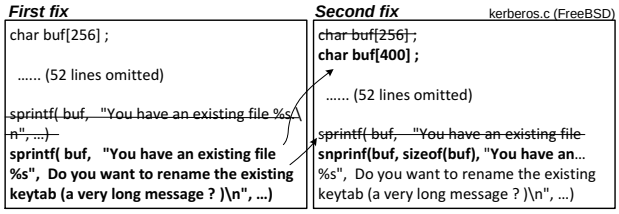

Figure 1: An incorrect fix example from FreeBSD. A part of the first fix appended a console message with some additional information, unfortunately introducing a buffer overflow (The added lines are in bold while the deleted lines are crossed out).

definitely not weeks. Such time pressure can cause *fixers* [2] to have much less time to think cautiously, especially about the potential side-effects and the interaction with the rest of the system. Similarly, such time pressure prevents testers from conducting thorough regression tests before releasing the fix. Figure 1 shows a real world example from FreeBSD, the original bug fix appended a log message with additional information. Unfortunately, the fixer did not pay attention to the buffer length defined 52 lines upwards in the same file and introduced a buffer overflow.

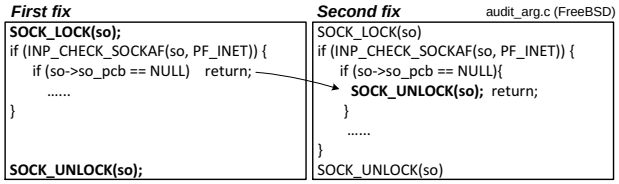

Figure 2: An incorrect fix example from FreeBSD. The first fix tried to fix a data race bug by adding locks, which then introduced a deadlock as it forgot to release the lock via *SOCK_UNLOCK* before *return*.

Second, bug fixing usually has a narrow focus (e.g., removing the bug) comparing to general development. As such, the fixer regards fixing the target bug as the sole objective and accomplishment to be evaluated by his/her manager. Therefore, he/she would pay much more attention to the bug itself than the correctness of the rest of the system. Similarly, such narrowly focused mindset may also be true for the testers: Tester may just focus on if the bug symptom observed previously is gone, but forget to test some other aspects, in particular how the fix interacts with other parts and whether it introduces new problems. As shown in Figure 2, the fixer just focused on removing the data race bug by adding locks. While the data race bug was removed, the fix unfortunately introduced a new bug: a deadlock. This deadlock was obviously not discovered during regression testing.

Third, the two factors above can be further magnified if fixers or reviewers are not familiar with the related code. While an ideal fixer could be someone with the most knowledge about the related code, in reality it may not always be the case. Sometimes, it may be difficult to know who is the best person to do the fix. Even if such person is known, he/she may be busy with other tasks or has moved to other projects, and is therefore unavailable to perform the fix. Sometimes, it is due to the development and maintenance process. Some software projects have separate teams for developing and maintaining software. All these real world situations can lead to the case that the fixer does not have enough knowledge about the code he/she is fixing, and consequently increases the chance of an incorrect fix. This might help explaining the incorrect fix shown in Figure 3 from the commercial OS we evaluated. When we measure the fixer's knowledge based on how many lines he had contributed to the file involved in the patch, we found that he had never touched this file in the past, indicating that he may not have sufficient relevant knowledge to fix the bug correctly.

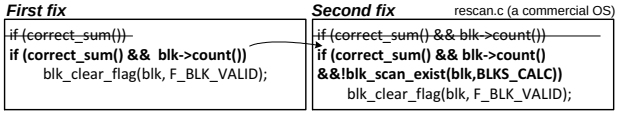

Figure 3: An incorrect fix that hadn't fixed the problem completely. *This example is from the large commercial OS we evaluated.* The first fix tried to address a semantic bug by modifying the *if* condition. Unfortunately, the revised condition was still not restrictive enough.

Regardless what is the reason for introducing these errors during bug fixing and why they were not caught before release, their common existences and severe impacts on users and vendors have raised some serious concerns about the bug fixing process. In order to come up with better process and more effective tools to address this problem, we need to first thoroughly understand the characteristics of incorrect fixes, including:

- *How significant is the problem of incorrect fixes?* More specifically, what percentages of bug fixes are incorrect? How severe are the problems caused by incorrect fixes?
- *What types of bugs are difficult to fix correctly?* Are some types of bugs just more difficult to fix correctly so that fixers, testers and code reviewers for these types of bug fixes should pay more attention and effort to avoid mistakes?
- *What are the common mistakes made in bug fixes?* Are there any patterns among incorrect bug fixes? If there are some common patterns, such knowledge would help alerting developers to pay special attention to certain aspects during bug fixing. Additionally, it may also inspire new tools to catch certain incorrect fixes automatically.
- *What aspects in the development process are correlated to the correctness of bug fixing?* For example, does fixers and reviewers' relevant knowledge have a high correlation to incorrect fixes?

A few recent studies had been conducted on certain aspects of incorrect fixes [6, 36, 33, 13]. For example, Śliwerski *et al.* [36] proposed an effective way to automatically locate fix-inducing changes and studied the incorrect fix ratios in Eclipse and Mozilla. They found developers are easier to make mistakes during bug fixing on Friday. Purushothaman *et al.* [33] studied the incorrect fix ratio in a switching system from Lucent, but their focus was on the impact of one-line changes. Gu *et al.* [13] studied the incorrect fix ratio in three Apache projects, but they focused on providing a new patch validation tool.

While these studies have revealed some interesting findings, most of them focused more on incorrect fix ratios and studied only open source code bases, providing one of the first steps toward understanding incorrect bug fixes. This

[2] we will refer the developer who fixes the bug as the "fixer" in the rest of the paper.

Importance of Incorrect Fixes	Implications
(1) At least 14.8%∼24.4% of examined fixes for post-release bugs are incorrect. 43% of the examined incorrect fixes can cause crashes, hangs, data corruptions or security problems.	Although the ratio of incorrect fixes is not very high, the impact of the incorrect fixes indicate that the problem of incorrect fixes is significant and worth special attention.
(2) Among common types of bugs and based on our samples, fixes on concurrency bugs (39% of them) are most error-prone, followed by semantic bugs (17%) and then memory bugs (14%).	Developers and testers should be more cautious when fixing concurrency bugs.
Incorrect fixes to Concurrency bugs	**Implications**
(3) Fixes on data race bugs can easily introduce new deadlock bugs or do not completely fix the problem.	The synchronization code added for fixing data races need to be examined in more detail to avoid new deadlock. Knowing all the access locations to the shared objects is the key to fix data race completely.
(4) Fixes to deadlock bugs might reveal bugs which were hidden by the previous deadlock.	Fixers need to further examine the path after deadlock in case there are some bugs hidden due to the existence of the deadlock.
Incorrect fixes to Memory bugs	**Implications**
(5) Fixing buffer overflows by statically increasing the buffer size is still vulnerable to future overflows. Fixing buffer overflows by dynamically allocating memory could introduce null pointer dereference bugs if the allocated memory is used without check.	It is better to use safe string functions (e.g., *snprintf*) or bound checking to fix buffer overflow. Fixers need to be aware of the potential memory leaks and the failure of allocation when fixing buffer overflows by dynamically allocating memory.
(6) Fixing memory leaks can introduce dangling pointer bugs when freeing the memory without nullifying the pointer, and memory corruption when freeing something that should not be freed, or do not solve the problem completely when forgetting to free the members of a structure.	It is good to nullify the pointer after freeing the memory. It is also important to clearly understand what and when should be freed to avoid overreaction. Fixers should remember to free the structure members when freeing a complex structure to avoid an incomplete fix.
Human reasons to incorrect fixes	**Implications**
(7) Comparing to correct fixes, the developers who introduced incorrect fixes have less knowledge (or familiarity) with the relevant code. *27%* of the incorrect fixes are even made by fixers who previously had never touched the files involved in the fix.	Code knowledge has influence on the correctness of bug fixes. It is dangerous to let developers who are not familiar with the relevant code to make the fix.
(8) Interestingly, in most of the cases, the developers who are most familiar (5∼6 times of the actual fixers) with the relevant code of these incorrect fixes are still working on the project, but unfortunately were not selected to do the fixes.	Having a right software maintenance process and selecting the right person to fix a bug is important.
(9) The code reviewers for incorrect fixes also have very poor relevant knowledge.	It is also important to select a developer who is familiar with the relevant code as the code-reviewer.

Table 1: Our major findings of real world incorrect bug fix characteristics and their implications. Please take our methodology and potential threats to validity into consideration when you interpret and draw any conclusions.

paper goes much beyond prior work, studying both commercial and open source, large operating system projects, and investigating not only incorrect fix percentages, but also other characteristics such as mistake patterns during bug fixing, types of bugs that are difficult to fix correctly, as well as the potential reasons in the development process for introducing incorrect bug fixes.

1.2 Our Contribution

To the best of our knowledge, this paper presents one of the most comprehensive characteristic studies on incorrect fixes from large OSes including a mature *commercial* OS developed and evolved over the last 12 years and three open-source OSes (FreeBSD, OpenSolaris and Linux), exploring not only the mistake patterns but also the possible *human reasons* in the development process when these incorrect fixes were introduced. More specifically, from these four OS code bases, we carefully examined each of the 970 randomly selected fixes for post-release bugs and identified the incorrect fixes. To gain a deeper understanding of what types of bugs are more difficult to fix correctly as well as the common mistakes made during fixing those bugs, we further sampled another set of 320 fixes on certain important types of bugs. The details of our methodology and potential threats to validity are described in Section 2.

Our major findings are summarized in Table 1. These findings provide useful guidelines for patch testing and validations as well as bug triage process. For example, *motivated from our findings, the large software vendor whose OS code was evaluated in our study is building a tool to improve its bug fixing and code review process.*

While we believe that the systems and fixes we examined well represent the characteristics in large operating systems, we do not intend to draw any general conclusions about all the applications. In particular, we should note that all of the characteristics and findings in this study are associated with the types of the systems and the programming languages they use. Therefore, our results should be taken with the specific system types and our methodology in mind.

Paper outline: In Section 2, we discuss the methodology used in our study and threats to validity. Section 2. After that we present our detailed results on the incorrect fix ratio in Section 3. Then we further study which types of bugs are more difficult to fix and what common mistakes could be made in Section 4. After that we study the human factors which could lead to incorrect fixes in Section 5. Section 6 is the related work and we conclude in Section 7.

2. METHODOLOGY

In this section, we first discuss the software projects used in our study (Section 2.1), the techniques to find incorrect fixes (Section 2.2), how we select bug samples (Section 2.3) and how we study the influence of human factors on bug

fixing(Section 2.4). At the end, we talk about the threats to the validity of our study (Section 2.5).

2.1 Software projects under study

App	LoC	Open src?
The commercial OS	confidential	N
FreeBSD	9.97M	Y
Linux	10.94M	Y
OpenSolaris	12.99M	Y

Table 2: The four OSes that our study uses.

Table 2 lists the four code bases we studied, including a commercial, closed-source OS from a large software vendor [3] and three open-source OSes (FreeBSD, Linux and OpenSolaris). We chose to study OS code because they are large, complex and their reliability is critically important. Additionally, as OS code is developed by many programmers, contains lots of components, uses a variety of data structures and algorithms, it could provide us a broad base to understand incorrect fix examples.

The four OSes have different architectures. The commercial OS is especially designed for high-reliability systems with many enterprise customers like big financial companies and government agencies. It has evolved for almost 12 years. The other three open-source OSes have different origins. FreeBSD originates from academia (Berkeley Unix). OpenSolaris originates from a commercial OS (Solaris). Linux completely originates from the open-source community. We think the variety in data sources would help us find general software laws or interesting specificities.

These OSes usually have multiple branches (series) in their OS families. We focus on those branches which are both stable and widely deployed. For the commercial OS, we chose the branch which is most widely deployed. For FreeBSD, we chose FreeBSD 7 series. For Linux, we chose Linux 2.6 series. For Opensolaris, it has a different release model so we just studied the releases since its 2008.5 version.

In order to further preserve the privacy and reputation for the software vendor, we anonymized the results in Section 3, 4 and 5. The four code bases will be just referred as A, B, C and D with the mapping information hidden. We know that such anonymization may prevent us from making some interesting comparison between open source and commercial code bases, but fortunately we can still make many other findings like the ones summarized in Introduction.

2.2 Finding incorrect fixes

The definition of incorrect fix: A bug fix f_x is defined as an incorrect fix if there is another following bug fix f_y that fixes either a new problem introduced by f_x, or the original problem that was not completely fixed by f_x.

Note that we consider only fixes to bugs, not any general or non-essential changes [16] (e.g., feature addition or renaming). This is identified by checking whether a fix is associated with a bug report. This screening criteria is important to the fidelity of our study since bug reports often contain rich information which is important for us to understand the fix. It is hard to obtain a complete picture from the bug fix itself alone.

Unfortunately, the link between fixes and bug reports is not always systematically maintained [5]. For the commercial OS and OpenSolaris, the SCM (software configuration management) systems record the link between every bug report and every change. However, for FreeBSD and Linux, such links are only documented voluntarily by developers in an unstructured way. To identify such links, we use a method similar to the methods used in [11, 36, 40]. The main idea is to leverage the verbal information in the bug reports or change logs to reconstruct the links. For example, developers may write "*the bug is fixed by change 0a134fad*" in a bug report. Then we can link the bug to the change 0a134fad.

After the above process, we will have a set of bug fixes linked with bug reports. We then randomly select a target number of bug fixes and then semi-automatically check whether each one is an incorrect fix or not. We call the process semi-automatic because we use a two-step process: first step automatically selects potential incorrect fix candidates, while the second step manually verifies each candidate.

Two techniques are used in the first step to automatically identify potential incorrect fix candidates. First, we look at the *source code overlap* between changes. This technique is similar to the methods used in [36, 33]. If there is source code overlap between two changes, then the latter change may be made to correct the previous change. More specifically, if a latter change f_y overwrites or deletes the code written in the previous change f_x or f_y just adds code in the proximity (+/-25 lines) of f_x, we regard f_x as an incorrect fix candidate for manual examination. The second technique is to search for specific keywords in the bug report and change log of each fix that may suggest an incorrect fix. For example, if we find "*this patch fixed a regression introduced by the fix in Bug 12476*" in the bug report linked with f_y, we regard the fix in "Bug 12476" as an incorrect fix candidate. In general, we find the first technique to be more comprehensive.

Please note that the first step is only identifying candidates. We still need to manually examine each candidate, which is the unique challenge in our study. We examined all the relevant code related to each fix. We also examined the bug reports and change logs to get proof from developer's explanation. For some fixes, we even discussed with developers of these systems to ensure the correct understanding on them. Then based on all the evidences we got, we finally decide whether a fix is incorrect or not.

Also note that the first step may prune a few incorrect fixes out, especially those incorrect fixes whose next fixes in a completely different location without any overlap at all (i.e. beyond the +/-25 lines proximity). But we expect such incorrect fixes are very rare as two subsequent fixes to the same problem usually has good locality in terms of code changes. And we did try to relax the proximity requirement to be "within" the same file but did not find more incorrect fixes.

2.3 The target bugs to study

In this study we used two sets of bug fixes with different focuses.

Sample set 1: To get this sample set, we first randomly sampled a total of 2,000 bug fixes (500 from each OS) that are associated with bug reports. From these 2,000 bug fixes, we further select only those fixes to *post-release* bugs (970 in total). Selecting such post-release bug fixes allows us to focus on fixes to bugs that made high impacts to both customers and vendors. Post-release bugs are selected after

[3]Due to confidentiality agreement, we can neither mention the company's name nor the LoC of its OS.

the random sampling instead of before the sampling because the manual effort in verifying all bug fixes would be too huge. We then use the process described in the previous subsection to identify and study incorrect fixes.

Sample set 2: This sample set is used to further zoom in certain bug types observed in *sample set 1* whose fixes are most error-prone. Specifically, we chose to study the fixes to memory leak, buffer overflow, data race and deadlock bugs. However, it is difficult to reuse the bugs in *sample set 1*, since there are not statistically sufficient number of bug fixes for these types of bugs. Therefore, we deliberately sampled more bug fixes focusing on these four types. Specifically, we used all the related keywords to search for the bug fixes of a specific type. Keyword search is enough to get these bug fixes since there are only limited ways to name them. Then we randomly selected 20 from each type for each code base. In total, we sampled 320 fixes which provide us a richer base-set to study the incorrect fixed patterns. This set is only used in Section 4.

2.4 Measuring code knowledge

To understand why a programmer cannot fix a bug correctly, we also dive deeper into his/her knowledge about the relevant code. In this study, we measure code knowledge by checking the cumulative "authorship" of each line of code at a particular version, which can be systematically measured. From SCM, we obtain the authorship of each line for a file at a given version by using commands such as "svn annotate". Assume a developer d, a file F, a function f and a version v, we calculate code knowledge at two levels of granularity:

$$K\_File_{d,F,v} = \frac{\text{The LoC written by } d \text{ for } F \text{ at } v}{\text{The total LoC in } F \text{ at } v}$$

$$K\_Func_{d,f,F,v} = \frac{\text{The LoC written by } d \text{ for } f \text{ in } F \text{ at } v}{\text{The total LoC in the } f \text{ at } v}$$

We use percentage as the unit of K_File and K_Func. For example, "$K\_File_{d,F,v}$=75%" means 75% of code lines in F at version v are written by d. d may write these code lines in any version that is not later than v. Both fixers' and reviewers' knowledge are measured in our study in this way.

2.5 Threats to validity

Real world empirical studies are all subject to validity problems, so is our study. Potential threats to the validity of our study are the representativeness of the selected software projects, the representativeness of the incorrect fix samples, our classification process and evaluation methodology.

Representativeness of software: Both commercial and open-source software are covered in this study, so we believe that we have a good coverage for at least OS code. We do not intend to draw any general conclusions in all software, but some of the findings such as fixers and reviewers' knowledge would also apply to other applications.

Representativeness of bug fix samples: We studied only those bug fixes that could be linked to a bug report. The set of fixes that cannot be linked were not covered, and some of our findings might not hold in them [5]. Fortunately, the results from the commercial OS and OpenSolaris are immune to this threat since every bug fix is linked to a bug report. Since the results from FreeBSD and Linux show a similar trend as the commercial OS and OpenSolaris, which may ease the concern on this threat for this problem.

As discussed earlier, there is also a potential problem in our automatic filtering process, i.e., we can potentially filter out an incorrect fix if its subsequent fix does not have any proximity in code location. Fortunately our exercises of relaxing the proximity constraint did not discover any more incorrect fixes, which indicates that the amount of missed incorrect fixes should be very low.

Threats of manual classification: Our study involves manual classification on bug reports and fixes which cannot be replaced by automatic techniques. Therefore, subjectivity is inevitable. However, we tried our best to minimize such subjectivity by using double verification. Also the authors have previous experience in studying bugs and familiar with OSes [19, 42, 38, 21]. For every incorrect fix candidate, we examined all the information sources we could have, including source code, bug reports, change logs, etc. Besides, for some fixes we also discussed with developers of these systems to ensure the correct understanding on them. Since we manually examined each incorrect fix candidate and classify it as incorrect only if we have concrete evidence, we are confident that the number of false positives should be very low.

Limitation in measuring the knowledge: The way we measure code knowledge is relatively simple since we only want to check the fixers and reviewers' knowledge in a coarse-grain, qualitative way. A more sophisticated knowledge model might provide us more accurate results in Section 5, which remains as our future study.

3. IS INCORRECT FIX REALLY A SIGNIFICANT PROBLEM?

The ratio of incorrect fixes among all bug fixes and the impact of bugs introduced by incorrect fixes are important for us to accurately understand whether incorrect bug fix is a significant problem. As described in Section 2, we first randomly sampled 2,000 bug fixes from the four OSes (500 from each OS), among them 970 are fixes to post-release bugs.

App	# of post-release bug fixes	# of incorrect fixes	Ratio
A	189	39	20.6%±3.0%
B	309	46	14.8%±2.9%
C	267	41	15.3%±2.6%
D	205	50	24.4%±3.7%

Table 3: The ratio of incorrect fixes on sampled post-release bugs in the four OSes. A 95% confidence interval is used.

Table 3 shows the ratio of incorrect fixes based on our samples is 14.8%∼24.4% among the four OSes. As discussed in Section 2.5, this is only a lower-bound estimation. Considering that the fixes on post-release bugs would be applied by a lot of customers and users, even this ratio can still have significant impact. Since regression testing had already been applied before releasing the fixes, it also indicates general testing techniques may need to be tailored to be more effective in capturing the errors in patches.

We further studied the impact of the bugs introduced by the examined incorrect fixes. We judge the impact based on the symptoms described in the bug reports. We found 14.0% of them introduced crash, 8.4% caused system to hang, 15.4% led to data corruption or data loss, 5.6% caused security problem, 7.0% degraded the performance, and 45.1% introduced incorrect functionality. Some bugs introduced

are actually more severe than the original bugs. Moreover, for some bugs, they could even be incorrectly fixed for several times.

Finding: At least 14.8%∼24.4% of the sampled bug fixes are incorrect. Moreover, 43% of the incorrect fixes resulted in severe bugs that caused crash, hang, data corruption or security problems.

Implication: Incorrect fix is indeed a significant problem that requires special attentions from software vendors.

4. INCORRECT FIX PATTERNS

Though bug fix patterns had already been studied in [17, 31], few had studied the patterns of incorrect fixes before. In this section, we first study which types of bugs are more likely to introduce incorrect fixes. Then we probe deeper into each of these bug types and try to understand their incorrect fix patterns via case studies. Finally, we discuss how we can leverage those patterns to detect incorrect fixes in the testing process.

4.1 Which types of bugs are more difficult to fix correctly?

We classified all the 970 sampled fixes into three categories based on the bugs they fix: memory bug, concurrency bug or semantic bug. Semantic bugs are those bugs that cannot be classified as memory or concurrency bug and are usually application specific problems. This classification is adopted from previous literature [19].

App	Concurrency	Memory	Semantic
A	4/13 (31%)	3/17 (18%)	32/159 (20%)
B	9/21 (43%)	5/44 (13%)	32/244 (13%)
C	7/19 (37%)	6/43 (14%)	28/205 (14%)
D	10/23 (44%)	5/30 (17%)	35/152 (23%)
Overall	30/76 (39%)	19/134 (14%)	127/760 (17%)

Table 4: The number of incorrect fixes among all the sampled fixes and the incorrect fix ratio for the three categories of bugs in the four OSes.

Table 4 shows the ratio of incorrect fixes to each type of bug. Based on our samples, concurrency bugs have the largest incorrect fix ratio (39% overall), indicating concurrency bugs are the hardest to fix. Semantic bugs and memory bugs have similar ratio, 17% and 14%, respectively.

We focus on studying concurrency bugs and memory bugs, while only providing some high level discussion on semantic bugs (Section 4.2.5). This is because semantic bugs have very diverse root causes so that it is difficult to observe general patterns from their fixes and the mistakes in the fixes.

Bug types and their percentages			
data race	33%	deadlock	29%
buffer overflow	8%	memory leak	6%
uninitialized read	4%	null pointer deref	4%

Table 5: The most observed bug types among all the concurrency bugs and memory bugs being fixed incorrectly. Only top six are shown.

To select the important types of bugs for a detailed study, we further dive into all the concurrency bugs and memory bugs being fixed incorrectly to see which sub-types are most observed. The result is shown in Table 5. Among all the bug types, data race (33%), deadlock (29%), buffer overflow (8%) and memory leak (6%) are the top four types of the most observed concurrency bugs and memory bugs which were fixed incorrectly. Therefore, we just focused on the characteristics of bug fixes to these four types of bugs.

App	Bug types			
	race	deadlock	buf overflow	mem leak
A	9/20 (45%)	5/20 (25%)	2/20 (10%)	1/20 (5%)
B	11/20 (55%)	6/20 (30%)	1/20 (5%)	3/20(15%)
C	11/20 (55%)	8/20 (40%)	3/20 (15%)	0/20 (0%)
D	8/20 (40%)	9/20 (45%)	1/20 (5%)	4/20 (20%)
All	39/80 (49%)	28/80 (35%)	7/80 (9%)	8/80 (10%)

Table 6: The number of incorrect fixes among the all the fixes and the incorrect fix ratio for the four important types of bugs from *sample set 2* .

Table 6 further shows the ratio of incorrect fixes for these four types of bugs. The result is from 320 fixes only to these bug types (*sample set 2* mentioned in Section 2.3), where for each type we randomly sampled 20 fixes from each code base. We use this data set instead of the one used above because among the original 970 fixes (*sample set 1* mentioned in Section 2.3), there are not statistically sufficient number of bug fixes for these four types of bugs. *sample set 2* provides us a richer base-set of incorrect fixes to conduct our case studies in Section 4.2. As indicated in Table 6, fixes to data race and deadlock are most error-prone, with an incorrect fix ratio of 49% and 35% respectively, which is generally 4x∼6x of buffer overflow and memory leak.

Finding: Based on our samples, concurrency bugs are the most difficult (39%) to fix right. Among concurrency and memory bugs which were fixed incorrectly, the four most observed bug types are: data race, deadlock, buffer overflow and memory leak.

Implication: Developers and testers should be more cautious when fixing concurrency bugs. The allocation of fixing and testing resources could consider the types of bugs to be fixed.

4.2 Mistakes in bug fixing

After understanding which types of bugs are more difficult to fix right, it would be interesting to understand the common mistakes (patterns) when fixing a particular type of bugs and the consequence introduced by those incorrect fixes. In this Section, we use the incorrect fix examples got from *sample set 2*. Generally, we find there are two types of incorrect fixes: *incomplete fixes* and *introducing new problems*, while each type of bug also has its own incorrect fix patterns. We also discuss techniques to detect or reveal these mistakes: either extending current techniques or suggesting new approaches.

4.2.1 Fixing data races

The most common practice for fixing data race is to add synchronization primitives (e.g., locks) to create mutual exclusion on shared resources. However, delivering a correct fix requires deep reasoning on all the side-effects of the newly added synchronization, which is often error-prone.

Specifically, adding locks might introduce deadlock. This incorrect fix pattern is observed in all the four code bases we evaluated and in 16.4% (6 out of 39) of the incorrect fixes to data race bugs. Figure 4 shows one of the examples. In the first fix, a lock *sc* was added to avoid a race. However, the function *bus_teardown_intr* is not supposed

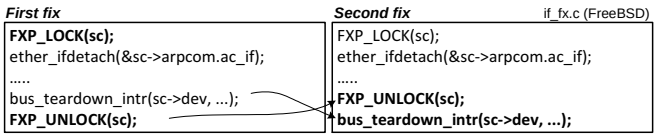

Figure 4: Incorrect fix to a data race introduced a deadlock. The function *bus_teardown_intr* cannot be called with lock held, otherwise deadlock will be introduced.

to be called inside the critical section, otherwise it can lead to deadlock. Unfortunately, developers were not aware of this rule and made the incorrect fix. To fix this deadlock, *bus_teardown_intr* was moved out of the critical section. Figure 2 (in Section 1) is another example of this pattern that fixing data race introduces deadlock. The fixer forgot to release the lock via *SOCK_UNLOCK* before a *return* statement therefore a deadlock happened.

Implications: When adding synchronization primitives, fixers need to make sure the newly added primitives (e.g., lock) will not introduce deadlocks with the existing synchronization code. This can be checked by extending deadlock detectors to only focus on the synchronization primitives newly added. Besides, lock and unlocks should be added in pairs along all the execution paths in the newly formed atomic region. This can be checked automatically by extending some existing path-sensitive bug detection tools such as RacerX [9] or PR-Miner [20] to only scan the code regions touched by the fix.

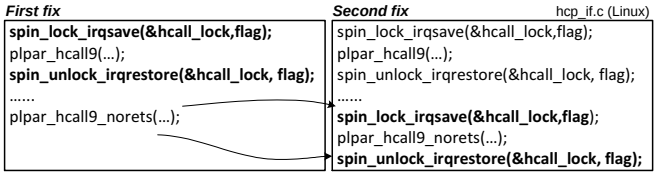

Figure 5: Fix to a data race was not complete. The first fix only added locks to protect function *plpar_hcall9*, while forgot to protect *plpar_hcall9_noret* (which contains the access to the same shared objects in *plpar_hcall9*).

Fixing data races could be incomplete such that not all the data races are fixed. This incorrect fix pattern is observed in three of the code bases we evaluated and in 10.2% (4 out of 39) of the incorrect fixes to data race bugs. For example, as shown in Figure 5, when adding locks, the developer forgets to lock all the places she should lock.

Implications: For a complete fix to data race, it is important to know all the accesses to the shared objects which could race with each others. We can design checkers to detect where the same shared objects are protected by lock in some paths but unprotected in some others [10], and make these checkers focus only on checking the patched code.

4.2.2 Fixing deadlocks

To fix a deadlock, developers may either reverse the order of locks, or even drop some locks. However, these means need to be applied with caution.

Specifically, fixing deadlocks could still lead to deadlock bugs. This incorrect fix pattern is observed in three of the code bases we evaluated and in 14.3% (4 out of 28) of the incorrect fixes to deadlocks. Figure 6 shows such an example. The root cause of this incorrect fix is similar to the one

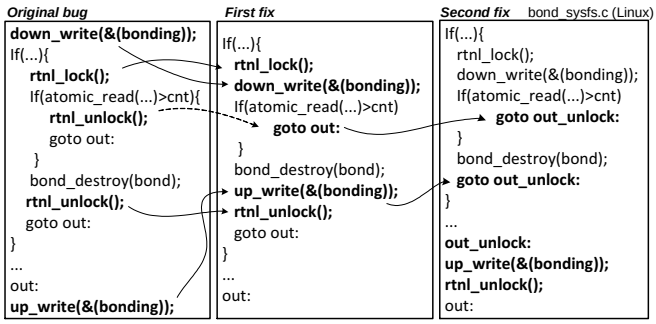

Figure 6: Incorrect fix to a deadlock introduced a new deadlock. The first fix reversed the order of locks to prevent deadlock, but forgot to release locks before taking a *goto* path.

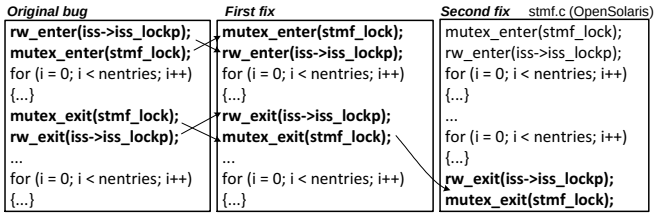

Figure 7: A fix to a deadlock exposed a hidden data race bug.

in Figure 2. Therefore we can again extend some current path-sensitive bug detection tools to spot the deadlock.

Additionally, fixing deadlock may reveal some other bugs that were originally hidden by the deadlock, especially data race bugs. This incorrect fix pattern is observed in two of the code bases we evaluated and in 7.1% (2 out of 28) of the incorrect fixes to deadlocks. Though we only spotted 2 such cases, we think this is still an interesting pattern. Figure 7 shows one of the examples. There are two bugs in the original code: a deadlock caused by the wrong order of the two locks, and a data race caused by an unprotected shared variable in the second *for* loop. However, the data race is hidden by the existence of the deadlock since the execution would not even reach the second *for* loop due to the deadlock. The first fix resolved the deadlock. However, it also enables the execution to proceed so that the data race is much easier to manifest.

Implications: The hang introduced by deadlock bugs might prevent some execution paths from being exercised thoroughly, which could make some bugs hidden in those paths difficult to manifest. After removing deadlock bugs, fixers should further test those execution paths.

4.2.3 Fixing buffer overflow

We also found some interesting incorrect fixes examples for memory bugs. However, since the total numbers of incorrect fixes to buffer overflows and memory leaks in *sample set2* is not statistically large enough (7 and 8 respectively), we do not claim those examples are frequently observed incorrect fix patterns. However, we assume these examples could be common for the incorrect fixes to buffer overflows and memory leaks (shown in Section 4.2.4) if we can further enlarge our sample set.

Common techniques to fix buffer overflow include: a) restrict the length of the data which will be stored into buffer by using safe string functions (e.g., *snprintf*) or do bound

checking b) increase the buffer size statically from stack c) allocate larger buffer dynamically from heap to replace a stack buffer.

Based on our observation, technique a) is usually safe and seldom introduces any further incorrect fixes since it eradicates the chance of a buffer overflow in the future.

Implications: The good practice to fix buffer overflow is to use safe string functions or do bound check when possible.

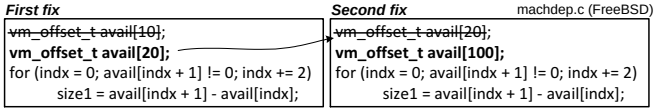

Figure 8: Incorrect fix to a buffer overflow by increasing static buffer size. The first fix enlarged the buffer size to 20, but the size was still not big enough. Under certain input, *avail* was still overflown.

Technique b) is potentially problematic if the developer cannot anticipate the input size accurately. The buffer size after increasing may still be not enough for an untested input in the future. As shown in Figure 8, the first fix was incomplete. After it increased the size of *avail* to 20, *avail* was still overflown later. Actually even the second fix might still be flawed. Since the developer does not add a bound check, a future input beyond the size 100 could still overflow *avail*.

Implications: Increasing the static buffer size can be dangerous if the input size cannot be accurately estimated.

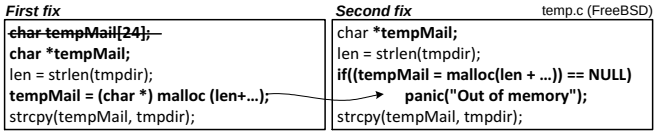

Figure 9: Incorrect fix to a buffer overflow by allocating heap memory. The first fix allocated heap memory to replace stack buffer, but the return value of malloc was unchecked.

For technique c), developers need to be aware of the rules to use memory allocation functions. The memory allocated needs to be freed after use, otherwise it may introduce a memory leak. Besides, developers need to consider to do error handling if memory allocation fails. As shown in Figure 9, the fixer did fix the buffer overflow, but introduced a potential invalid memory access.

Implications: When allocating memory dynamically to fix buffer overflow, developers also need to follow the safety rules of using memory allocation functions.

4.2.4 Fixing memory leak

Once a memory leak is detected, writing fixes may be straightforward, but mistakes can still be made.

Specifically, fixing memory leak can introduce dangling pointer or null pointer dereference if the pointer would still be accessed after the free. Figure 10 shows an example where a dangling pointer bug was introduced.

Implications: It is a good practice to nullify the pointer after freeing it, which can avoid dangling pointer bugs.

Developer may also not be aware of the condition to free an object. They should only free an object when it is no longer used. If they overreact, they could mistakenly free an object still in use under certain conditions. Figure 11 shows such an example which led to data corruption.

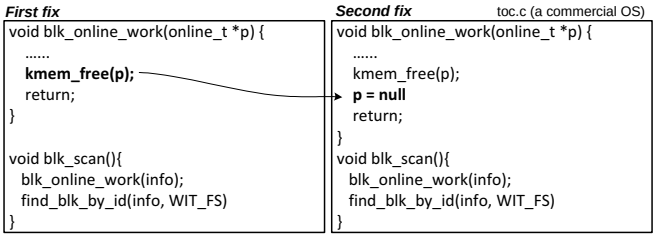

Figure 10: Incorrect fix to memory leak introduced a dangling pointer. The pointer *p* was later used in function *find_blk_by_id* with null pointer check. However, the first fix simply freed *p* without nullifying it.

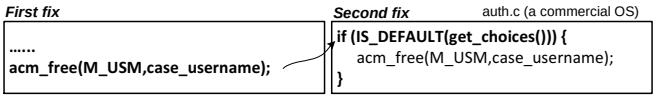

Figure 11: Incorrect fix to a memory leak introduced data corruption. The first fix freed the data indexed by *case_username* unconditionally. However, the data should be freed only under certain conditions.

Implications: Before fixing memory leak, developers should make sure when and what should be freed.

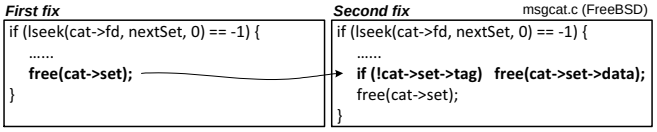

Figure 12: Incomplete fix to a memory leak. The first fix only freed *cat->set* but forgot to free its member *data*.

Besides, fixing memory leak can be incomplete. For some complex data structures, fixers may forget to free all their members. Figure 12 shows such an example.

Implications: For complex data structures, fixers should remember to free all their members.

4.2.5 Fixing semantic bugs

Semantic bugs have very diverse root causes, so the ways to fix them are also diversified. However, we still observed one common incorrect fix pattern for semantic bugs: conditions (e.g., *if* condition) are difficult to fix correctly. As shown in Figure 3 in Section 1, the first fix to the *if* condition was still not restrictive enough. Though this pattern is frequently observed, it is not easy to leverage current techniques to detect them. We think fixing semantic bugs correctly may require more application specific knowledge from fixers.

4.2.6 General approaches to detect incorrect fixes

Understanding the impact of the change: A fundamental reason for developers to make mistakes during bug fixing is that they do not know all the potential impacts of the newly fixed code. For example, in Figure 4, the fixer was not aware that the newly added lock *sc* would deadlock with the function *bus_teardown_intr*. If all such potential "influenced code" (either through control- or data-dependency) is clearly presented to developers, they may have better chances to detect the errors. We envision compiler techniques such as program slicing [41, 12, 14, 43] can be extended to analyze such information, using the dependencies to the patch as the slicing criterion.

App	Actual fixer for incorrect fixes		Actual fixer for correct fixes		Potential optimal fixer	
	K_File	K_Func	K_File	K_Func	K_File	K_Func
A	13.2% (*0.022*)	18.1% (*0.046*)	18.3% (*0.019*)	20.5% (*0.012*)	65.0% (*0.043*)	75.1% (*0.031*)
B	9.5% (*0.013*)	11.5% (*0.023*)	15.4% (*0.016*)	27.9% (*0.031*)	39.7% (*0.024*)	51.4% (*0.022*)
C	12.8% (*0.024*)	16.1% (*0.037*)	17.2% (*0.021*)	18.4% (*0.023*)	69.8% (*0.031*)	78.1% (*0.026*)
D	7.9% (*0.017*)	12.5% (*0.035*)	15.5% (*0.024*)	16.4% (*0.021*)	78.0% (*0.023*)	78.4% (*0.039*)
AVG	10.9%	14.6%	16.6%	20.8%	63.1%	70.8%

Table 7: The fixers' average code knowledge on the buggy files/functions. The *variance* of the code knowledge is shown in the parentheses. Code knowledge is shown in the form of percentage (e.g., 13.2% means a knowledge value of 0.132). "Potential optimal fixer" is the developer with the most knowledge on the buggy files/functions but might not be always assigned the bug fixing task.

Apply checkers incrementally As discussed before, it is possible for some existing bug detection tools (checkers) [20, 9, 10] to detect some types of incorrect fixes. However, applying these tools directly on the full code base after the fix is not practical: it may take a very long time for them to scan the entire code base, which may be redundant with the original testing steps, or not always necessary. Also it may produce too many false positives. Instead, developers may want to check the code influenced by the patch first. One observation is that sometimes, just checking within the function boundary is enough to detect problems in the patch. For example, in Figure 2 (Section 1), a path-sensitive checker that simply checks the rule "lock is always paired with an unlock" can easily detect the missed *SOCK_UNLOCK* by only scanning the function that the patch modified.

Dealing with incomplete fixes Some incomplete fixes are introduced by the fact that fixers may forget to fix all the buggy regions with the same root cause. This types of incomplete fixes can be mitigated by using technique [30, 28] which searches for other places that have the same patterns or usage scenarios in entire code. For example, in Figure 5, the first fix that suggested certain shared objects need to be protected. Then developers can try to find the other places where those objects are accessed without protection. However, this technique is less effective when a consistent pattern is difficult to learn. Moreover, those incomplete fixes related to conditions (Figure 3 in Section 1) cannot be solved by this technique.

5. LACK OF KNOWLEDGE

Multiple factors can influence developers to make an incorrect fix. In this section, we focus on programmers' code knowledge. Intuitively, if a file or a function is mostly written by a developer, the developer may have higher chance to give a correct fix to a bug rooted in that file or function.

We first measured the K_File and K_Func (defined in Section 2.4) for the fixers who made the incorrect fixes. The incorrect fixes are from *sample set 1*. The results are shown in Table 7. We found that in general these fixers who made the incorrect fixes were not knowledgeable about the buggy files/functions. Specifically, they had only contributed on average 10.9% to the files and 14.6% to the functions involved in the patch before they made the incorrect fix. In comparison, Table 7 also shows the fixers' knowledge in *correct* fixes. The correct fixes are from the complement of the incorrect fix set (Table 3). We found those fixers who made the correct fixes had contributed on average 16.6% to the files and 20.8% to the functions. In other words, the knowledge of the fixers who made the correct fixes is 1.5 times of that of the fixers who made the incorrect fix based on our code knowledge metrics, indicating source code knowledge could be a factor to incorrect fixes.

But can we really find a developer who is *more knowledgeable* than the actual fixer of the incorrect fix? Table 7 also answered this question: surprisingly, by selecting the most knowledgeable developer who is still active in the development when the bugs need to be fixed as the fixer, the K_File and K_Func can reach as high as 63.1% and 70.8% respectively, which is 5∼6 times of the knowledge of the actual fixers in incorrect fixes. Additionally, by selecting the two of the most knowledgeable developers as reviewers (two is the average number of reviewers in the OSes we studied), the K_File and K_Func can reach 68.2% and 78.8% respectively, which is 6∼7 times of the knowledge of the actual reviewers on each incorrect fix. Note that these "potential optimal fixers" are still reachable when the bugs were opened, which suggests that the current bug fixing and reviewing process is not always assigning the problem to the developers who could be most "knowledgeable" with the bug.

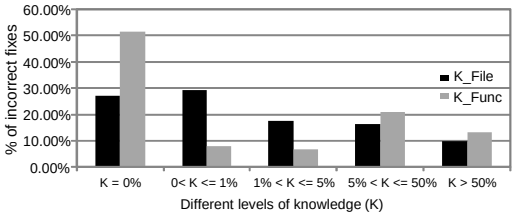

Figure 13: The distribution of incorrect fixes in different knowledge scales.

Figure 13 further studies why the K_File and K_Func in incorrect fixes are low by probing deeper into the distribution of incorrect fixes in different knowledge scales. We found the low K_File and K_Func are caused by a large portion of incorrect fixes that were made by fixers with *zero* prior knowledge to the buggy files/functions. As shown in Figure 13, *27.2% of the fixers had not contributed any lines to the file ($K\_File = 0\%$) they were about to fix.* It is even worse at the function level. 51.4% of fixers had not contributed any lines of code to the function they were about to fix. These "*first touches*" could be dangerous since the developer could have little knowledge about the particular part of code when they are doing the fix.

Besides studying the effect of code knowledge in terms of K_File and K_Func, we further study the code knowledge in terms of whether the fixer actually fixed the lines of code previously written by him. The intuition behind this is that even though a fixer had written small amount of code (i.e., small K_File/K_func), as long as the fixer was modifying the code regions that he had written, he might be still considered as the knowledgeable person for the fix. Specifically, for each fix, if any code line modified by the fix was also

previously written by the same fixer, we count this fixer is fixing his own code. The result is shown in Table 8.

App	for incorrect fixes	for correct fixes
A	7.7%	26.2%
B	16.4%	44.8%
C	18.2%	25.9%
D	8.0%	24.1%
AVG	12.6%	30.3%

Table 8: The percentage of fixes that fixer is fixing his own code. For example, the number 7.7% in the first cell means: among all the incorrect fixes from OS A, 7.7% of them were actually fixed by developers who were fixing their own code.

Table 8 shows large difference between correct fixes and incorrect fixes. The ratio of the fixers who fixed their own code for correct fixes is 2.5 times of the ratio for incorrect fixes (30.3% v.s. 12.6%). This suggests fewer fixers (in term of ratio) are fixing their own code for incorrect fixes than for correct fixes, which further suggests that fixing code written by others might be prone to incorrect fixes.

Based on our study in code knowledge, a software vendor is building a tool to find knowledgeable fixers and reviewers. Sometimes when the bug report just arrives it may be unclear which files/functions contain the bug. In these cases the knowledge is more useful to assign *reviewers* who are knowledgeable to the files/functions involved in the fix after the fix is made. This knowledge can also be used in prioritizing patch testing efforts to pay more attentions to the patches fixed by less-knowledgeable fixers.

Finding: The fixers' knowledge on the buggy files/functions in correct fixes is 1.5 times of the fixers' knowledge in incorrect fixes. Fewer fixers (in term of ratio) are fixing their own code for incorrect fixes than for correct fixes. Moreover, nearly 27% of the incorrect fixes are made by developers who have not contributed a single line to the entire file they are about to fix. The potential "Optimal" developer who is most familiar with the buggy code has 5∼6 times knowledge of that of the actual fixer in incorrect fixes.

Implication: It might be beneficial to assign the bugs to developers with more knowledge during the bug-triage process. The knowledge can also be considered as a factor in prioritizing the testing efforts on patches.

6. RELATED WORK

Studying incorrect fixes As briefly discussed in Introduction, several previous studies [6, 36, 33, 13] had also studied incorrect fixes. Our work is complementary in several ways: (1) we focus on large OS code, while previous studies focused on certain types of applications. (2)We study both commercial and open source code bases, while previous work studied only either open source or commericial. (3) Previous studies more focused on measuring incorrect fix ratios, while we went much beyond and also studied what types of bug fixes are more error-prone, the common mistake patterns, as well as the possible human reason in the development process for introducing incorrect fixes.

Śliwerski *et al.* [36] proposed an effective way to automatically locate fix-inducing changes by linking a version archive to a bug database. They studied the incorrect fix ratio in Eclipse and Mozilla and also found developers are easier to make incorrect changes on Friday. Purushothaman *et al.* [33] studied the incorrect fix ratio in a switching system from Lucent, but their focus was the impact of one-line changes. Gu *et al.* [13] studied the incorrect fix ratio in three Apache projects, but they focused on providing a tool to validate the patch. Baker *et al.* [6] visualized the incorrect fix ratio for a switch system in AT&T.

Human factors The influence of code knowledge on general code changes had been explored in [34, 26]. Mockus *et al.* [26] found that changes made by more experienced developers were less likely to induce failures. Rahman *et al.* [34] found file owner with higher knowledge is less associated with fix-inducing code. Our study focused on *bug fixes* and measured the knowledge of the fixers who made the incorrect fixes in commercial and widely used open source OSes. We found 27% of the incorrect fixes are made by fixers with *zero* knowledge, suggesting there might be some flaws in the overall bug assignment process. Some work [3, 24] also studied human factors for designing recommendation systems. Anvik *et al.* [3] suggested to assign fixer based on bug history. McDonald *et al.* [24] suggested to find the person who last modified the code. We proposed to assign fixer/reviewer based on code knowledge defined at line level. Besides, other aspects of human factors had also been studied. Meneely *et al.* [25] found that independent developer groups were more likely to introduce a vulnerability. Bird *et al.* [7] found that a binary might be more buggy if more developers are working on it. Nagappan *et al.* [27] studied the organizational structure and used it to build model to predict the failure proneness in Windows Vista.

Taming incorrect fixes There are different ways to solve the problem of incorrect fixes including predicting or isolating buggy changes [37, 18, 23, 44], patch validation [13, 39], automatic patching [32] and regression testing [35, 29]. Śliwerski *et al.* built a plug-in for Eclipse which shows the risk of changing a particular code location based on previous revision information. Kim *et al.* [18] also leveraged the historical source repository data to train models for predicting the correctness of a future change. McCamant *et al.* [23] compared operational abstractions generated from the old component and the new component to predict the safety of a component upgrade. Zeller *et al.* [44] proposed automated delta debugging to locate the bug introducing changes. ClearView [32] automatically generates patches without human intervention, which can reduce the chance of incorrect fixes. Besides, regression testing [35, 29] is also a common practise to ensure patches don't break the previously working functionalities. Our study discovered some incorrect fix patterns which are helpful for detecting/exposing/avoiding incorrect fixes. We studied what mistakes programmers should be aware of during bug fixing, which are also useful to design new detection tools to detect errors in fixes. Besides, we also proposed a bug assignment process based on code knowledge, which is being implemented by a large software vendor.

7. CONCLUSION AND FUTURE WORK

This paper presents one of the most comprehensive characteristic studies on incorrect bug-fixes from large operating system code bases, including a commercial OS project. We first studied the ratio and impact of incorrect fixes, and found incorrect fix is a significant problem that requires spe-

cial attention. We also studied the common patterns of mistakes made in incorrect fixes that can be used to alert the programmers as well as to design detection tools to catch incorrect fixes. We finally studied the code knowledge of developers and found 27% of incorrect fixes are made by developers who have not contributed a single line to the entire file they are about to fix. A tool based on our findings to assign the most-knowledgeable developer to fix/review the bug is being built into the bug assignment process of a large software vendor.

Though we had already done some preliminary study on the the fixes to semantics bugs, there are still some interesting questions waiting to be answered considering their diversity. Therefore, we plan to have a more comprehensive study on the fixes to semantics bugs in the future. Besides, we also plan to extend the characteristic study to non-OS applications, such as server applications and client applications. Another possible direction we want to proceed is to build some checkers to detect incorrect fixes based on the patterns we learned.

8. REFERENCES

[1] Microsoft security bulletin. `http://www.microsoft.com/technet/security/current.aspx`.

[2] After buggy patch, criminals exploit Windows flaw. `http://www.infoworld.com/d/security-central/after-buggy-patch-criminals-exploit-windows-flaw-848`.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE'06*.

[4] Apple updates leopard again. `http://voices.washingtonpost.com/fasterforward/2008/02/apple_updates_leopardagain.html`.

[5] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *FSE'10*.

[6] M. J. Baker and S. G. Eick. Visualizing software systems. In *ICSE'94*.

[7] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality? An empirical case study of Windows Vista. In *ICSE'09*.

[8] Buggy McAfee update whacks Windows XP PCs. `http://news.cnet.com/8301-1009_3-20003074-83.html`.

[9] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP'03*.

[10] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP'01*.

[11] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM'03*.

[12] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[13] Z. Gu, E. T.Barr, D. J.Hamilton, and Z. Su. Has the bug really been fixed? In *ICSE'10*.

[14] M. Harman, D. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, October 2003.

[15] Intel reissues buggy patch. `http://www.pcworld.com/businesscenter/article/126918/rss.html`.

[16] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *ICSE'11*, May 2011.

[17] S. Kim, K. Pan, and J. E. James Whitehead. Memories of bug fixes. In *FSE'06*, November 2006.

[18] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Trans. Software Engineering*, 34(2):181–196, March 2008.

[19] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *ASID'06*.

[20] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE'05*.

[21] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, March 2008.

[22] McAfee to reimburse customers for bad patch. `http://www.computerworlduk.com/technology/security-products/prevention/news/index.cfm?newsId=20005`.

[23] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *FSE'03*.

[24] D. W. McDonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *CSCW'00*.

[25] A. Meneely and L. Williams. Secure open source collaboration: An empirical study of linus's law. In *CCS'09*.

[26] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5:169–180, 2000.

[27] N. Nagappan, B. Murphy, and V. R. Basili. The influence of organizational structure on software quality. In *ICSE'08*.

[28] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ICSE'10*, May 2010.

[29] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *FSE'04*.

[30] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Eurosys'08*.

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

[32] J. H. Perkins, S. Kim, S. Larseng, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pachecod, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP'09*, October 2009.

[33] R. Purushothaman and D. E. Perry. Towards understanding the rhetoric of small changes. In *MSR'04*.

[34] F. Rahman and P. Devanbu. Ownership and experience in fix-inducing code. In *UC Davis Department of Computer Science, Technical Report CSE-2010-4*, 2010.

[35] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27:929–948, 2001.

[36] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR'05*.

[37] J. Śliwerski, T. Zimmermann, and A. Zeller. Hatari: Raising risk awareness (research demonstration). In *FSE'05*, September 2005.

[38] L. Tan, D. Yuan, and Y. Zhou. /* icomment: Bugs or bad comments? */. In *SOSP*, October 2007.

[39] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validationwith delta execution. In *ASPLOS'09*.

[40] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *ICSE'03*.

[41] M. Weiser. Program slicing. In *ICSE'83*.

[42] Z. Yin, M. Caesar, and Y. Zhou. Towards understanding bugs in open source router software. *ACM SIGCOMM Computer Communication Review*, 40(3):34–40, July 2010.

[43] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ASPLOS'10*.

[44] A. Zeller. Yesterday, my program worked. today, it does not. why? In *FSE'99*.