Empir Software Eng (2014) 19:1665–1705
DOI 10.1007/s10664-013-9258-8

Bug characteristics in open source software

Lin Tan · Chen Liu · Zhenmin Li · Xuanhui Wang · Yuanyuan Zhou · Chengxiang Zhai

Published online: 7 June 2013

Abstract To design effective tools for detecting and recovering from software failures requires a deep understanding of software bug characteristics. We study software bug characteristics by sampling 2,060 real world bugs in three large, representative open-source projects—the Linux kernel, Mozilla, and Apache. We manually study these bugs in three dimensions—root causes, impacts, and components. We further study the correlation between categories in different dimensions, and the trend of different types of bugs. The findings include: (1) semantic bugs are the dominant root cause. As software evolves, semantic bugs increase, while memory-related bugs decrease, calling for more research effort to address semantic bugs; (2)

Communicated by: Thomas Zimmermann

L. Tan(✉) · C. Liu
University of Waterloo, 200 University Avenue West, Waterloo, Ontario, N2L 3G1, Canada
e-mail: lintan@uwaterloo.ca

C. Liu
e-mail: c92liu@uwaterloo.ca

Z. Li
VMware, Inc., 3401 Hillview Ave, Palo Alto, CA 94304, USA
e-mail: zhenmin.li@gmail.com

X. Wang
Facebook, Inc., 1601 Willow Rd, Menlo Park, CA 94025, USA
e-mail: xuanhui@gmail.com

Y. Zhou
University of California, San Diego & Pattern Insight, Inc, 9500 Gilman Drive, M/C 0404,
La Jolla, CA 92093, USA
e-mail: yyzhou@cs.ucsd.edu

C. Zhai
University of Illinois, Urbana-Champaign, 201 North Goodwin Avenue, Urbana, IL 61801,
USA
e-mail: czhai@cs.illinois.edu

the Linux kernel operating system (OS) has more concurrency bugs than its non-OS counterparts, suggesting more effort into detecting concurrency bugs in operating system code; and (3) reported security bugs are increasing, and the majority of them are caused by semantic bugs, suggesting more support to help developers diagnose and fix security bugs, especially semantic security bugs. In addition, to reduce the manual effort in building bug benchmarks for evaluating bug detection and diagnosis tools, we use machine learning techniques to classify 109,014 bugs automatically.

Keywords Software bug characteristics · Empirical study · Software reliability · Open source · Bug detection

1 Introduction

Software defects severely hurt software dependability. To design effective tools for improving software dependability requires a good understanding of software defect characteristics in representative software. In this paper, we use "defect" and "bug" interchangeably, which is in accordance with their definitions (Avizienis et al. 2004). Such characteristics include bug root cause, bug impact, bug location, their distributions, and their correlation. For example, if simple typos or copy-pastes cause many bugs (Li et al. 2004), software development tools can provide more support to help detect such bugs automatically.

Empirical studies (Basili and Perricone 1984; Chillarege et al. 1991; Endres 1975; Glass 1981; Lu et al. 2008; Ostrand and Weyuker 1984; Sullivan and Chillarege 1991, 1992) have been performed to understand the characteristics of software bugs. For example, researchers have studied software errors occurring during software development, testing, and validation phases (Endres 1975; Glass 1981). Sullivan and Chillarege analyzed error type, defect type and error trigger distribution for shipped code of two IBM database management products and one IBM operating system (Sullivan and Chillarege 1991, 1992). They found that undefined state errors dominated but did not have high impact on availability, while memory allocation errors, pointer errors, and synchronization errors had high impact. A recent study (Sahoo et al. 2010) on bug reports in server applications shows that about 77 % of the failures are caused by bugs that can be reproduced with just one input request. These studies provide insights and guidelines for software engineering tool designers and reliable system builders.

Given the benefits of empirical studies, we want to answer new research questions (RQ) that have not been studied or not thoroughly studied before to provide new inspirations and guidance for building high quality software. Therefore, we study bug characteristics in three large, representative open source projects—one client-side application, Mozilla, one server-side application, Apache web server, and one operating system (OS), the Linux kernel. We first randomly sample 1,135 fixed *bug reports* from Mozilla, Apache, and Linux kernel bug databases, manually examine them, and identify 583 *bugs*. To ensure correct results, we only study *fixed* bug reports, because root causes described in unfixed reports can be wrong. A detailed explanation is in Section 2. The number of bugs is smaller than the number of bug reports, because many bug reports are invalid bugs, new feature requests, etc. (Section 2). We study the bug distribution in three dimensions—root causes, impacts, and components. In addition, we study the statistical correlation between these dimensions, which has not been systematically studied before (to the best of our knowledge). Furthermore, we study 1,387 security bugs and 90 concurrency bugs to understand the characteristics of these emerging types of bugs.

In summary, we classified and studied **2,060** bugs in the three large open source projects. In addition, we applied machine learning techniques to automatically classify a large number (**109,014**) of bugs.

The findings and their implications include:

- **RQ1**: As software becomes mature, what are the dominant types of bugs, e.g., memory bugs, concurrency bugs, or semantic bugs? Memory bugs are bugs that are caused by improper handling of memory objects. "Concurrency bugs are synchronization problems among the concurrent tasks in concurrent programs" (Lu 2008), including data races and deadlocks. Semantic bugs are inconsistencies with the requirements or the programmers' intention that are not memory bugs or concurrency bugs. The definition and detailed explanation of memory bugs, concurrency bugs, and semantic bugs are in Tables 3, 4 and Section 3.1.

  The motivation of studying RQ1 is as follows. Software such as the Mozilla suites, the Linux kernel, and the Apache web server has been used for decades. As software becomes more mature and stable, many bugs are still reported everyday (ASF Bugzilla 2013; Kernel Bug Tracker 2013; Mozilla.org Bugzilla 2013). Maybe more effort should be investigated into addressing the dominant types of bugs in mature and stable software.

  Memory bugs—bugs caused by improper handling of memory objects—only account for 11.8 ± 3.4 % in Mozilla, 12.4 ± 6.6 % in Apache, and 16.3 ± 6.0 % in the Linux kernel, much less than the 28–38 % reported in previous studies (Sullivan and Chillarege 1991, 1992). In this paper, we calculate and present the statistical margin of error with a 95 % confidence level to indicate the random sampling error in our results. In addition, memory bugs decrease as the evaluated software evolves. These results indicate that memory bugs are becoming less pervasive as the evaluated software becomes mature and stable. This may be attributed to the available techniques to automatically detect them, as developers actively use bug detection tools to find bugs during the development process (Debugging Memory Leaks 2013) (Section 4.1).

  Despite the decreasing trend of memory bugs as a whole, Mozilla, Apache, and the Linux kernel contain many simple memory bugs such as NULL pointer dereferences and uninitialized memory reads. These simple bugs can be detected by bug detection tools such as Coverity (2013), Hastings and Joyce (1992), and Valgrind (2013). This situation indicates that memory bug detection tools may have not been used with their *full* capacity. Therefore, it is important for researchers and tool builders to understand why these tools have not been fully utilized and address the relevant issues, e.g., reducing false positives, simplifying the usage procedure, promoting these tools to more developers, improving the detection capability of the current bug detection tools, etc (Section 4.1).

  Semantic bugs are the major root cause, accounting for 87.0 % in Mozilla, 82.5 % in Apache, and 70.1 % in the Linux kernel; and their percentages increase with the maturity of these software projects. In addition, semantic bugs have severe impacts on system availability, contributing to 18.2–59.3 % of crashes in the evaluated software. Our results suggest that more effort should be put into automatically detecting and diagnosing semantic bugs (Section 4.2).

Careless programming causes many bugs in the evaluated software. For example, simple bugs such as typos account for 7.8–15.0 % of semantic bugs in Mozilla, Apache, and the Linux kernel. The result indicates that it would be beneficial for software development environments such as Microsoft Visual Studio and Eclipse to provide more support to avoid them in the early stages of software development (Section 4.2).

- **RQ2**: What are the impact of bugs and what types of bugs have severe impact? One may want to spend more effort on addressing bugs with severe impact.
  Most bugs result in incorrect functionality in Mozilla, Apache, and the Linux kernel due to the dominance of semantic bugs. In contrast, only 11.3–27.2 % bugs in these projects result in crashes. The main root cause of crashes is memory bugs, accounting for 38.9–72.7 % of crashes, which may indicate that the diagnosis of crashes should focus on memory bugs (Section 5).
- **RQ3a**: Are there many graphical user interfaces (GUI) related bugs? **RQ3b**: Since GUI modules and their development process are quite different from other modules, do they have different root cause distributions?
  In order to provide friendly user interface, GUIs have become one of the major components in many systems. Although GUIs have become more complex and widely used, GUI testing techniques still significantly lag behind (Memon 2002). Therefore, we would like to study the questions above regarding GUI-related bugs.
  GUI bugs have become the major type of bugs in graphical interface software, accounting for 50.1 % of bugs in Mozilla, and resulting in 45.6 % of all crashes. In addition, GUI bugs are more likely to be caused by semantic bugs. The results imply that effective GUI bug avoidance and detection techniques (Amir and Tao 2005) are needed, and the focus of GUI testing should be on semantic bugs (Section 6).
- **RQ4**: Does operating system (OS) code such as the Linux kernel have different characteristics from its non-OS counterparts, and require different bug detection and diagnosis techniques? As operating systems inherently need to manage shared resources, do operating system code contain more concurrency bugs?
  An OS runs between hardware and application software. They can have different roles and characteristics from application software, because they by nature are unique, e.g., the daunting complexity and size, hardware dependence, the ability to manage interrupts, and many different kinds of resources. Therefore, we want to compare and contrast OS bugs with non-OS bugs.
  The Linux kernel has many similar bug characteristics as its non-OS counterparts, e.g., similar percentage of memory bugs. One striking difference is that the Linux kernel has significantly more concurrency bugs (13.6 %) than Mozilla (1.2 %) and Apache (5.2 %) probably due to its inherent nature of dealing with concurrent activities and shared resources. The results imply that more effort should be put into detecting OS concurrency bugs, especially given that many current concurrency bug detection techniques are ineffective or not applicable to OS code due to the complexity of OS synchronization and the difficulty of instrumenting an OS. In addition, 52.9 % of the reported Linux kernel bugs are in device driver code, which is similar to the results from prior studies (Chou et al. 2001; Swift et al. 2003). Different from the previous study (Chou et al.

2001), we study field kernel bugs that are *reported* by users and developers to show that driver bugs dominate. In addition, a significant proportion (10.2 %) of the Linux kernel bugs are related to interrupt handling, e.g., forgetting to disable or enable interrupts, missing certain interrupts, not ignoring certain interrupts, etc. We may want to provide more support to help the Linux kernel developers write correct driver code and interrupt-related code (Section 7).

- **RQ5a**: Does the number and proportion of reported security bugs increase? **RQ5b**: Is the common belief “buffer overflows are the most common type of security vulnerabilities” (Cowan et al. 2000) true? If not (as suggested by a recent study (Neuhaus and Zimmermann 2010)), what types of bugs cause more security vulnerabilities? Answers to these questions can provide guidance in addressing the important types of security vulnerabilities.

  We are interested in the trend of security bugs, because over recent years, security is becoming increasingly important, as many malicious users exploit software vulnerabilities to tamper system integrity, steal confidential data, and make systems unavailable (Cowan 2003; Payne 2002).

  Our results show that security bugs increased significantly over time in terms of number and proportion in Mozilla, Apache, and the Linux kernel. Among different root causes of security vulnerabilities, memory bugs contribute to only 27.7–34.5 % but are usually severe, while semantic bugs are the dominant cause, accounting for 61.8–71.5 %. This distribution is against the common belief that buffer overflows are the major cause of security vulnerabilities (Cowan et al. 2000), which is consistent with a recent study (Neuhaus and Zimmermann 2010). The result suggests that while it is important to detect buffer overflows to reduce security vulnerabilities, we should also provide support for detecting, diagnosing, and fixing security vulnerabilities caused by semantic bugs (Section 8).
- **RQ6a**: Are concurrency bugs hard-to-reproduce? **RQ6b**: Do these concurrency bugs cause severe impacts on software systems? If concurrency bugs cause severe impacts and are hard-to-reproduce, maybe we should call for new techniques in detecting and reproducing concurrency bugs.

  Due to the recent advances in hardware, modern computer systems, especially server systems, are configured with multi-processors. As a result, many software systems are multithreaded or multiprocessed to exploit the parallelism provided by hardware. This shift will continue especially due to the trend of multithreaded/multicore micro-architecture. To exploit these technologies, increasingly more software is becoming multithreaded. Therefore, it would be interesting to study the questions above regarding concurrency bugs.

  Most Mozilla concurrency bugs are hard-to-reproduce: They could not be reproduced by developers with an acceptable probability (e.g., more than once out of ten times). Therefore, to help developers diagnose and fix concurrency bugs, it would be beneficial to provide support to help developers reproduce concurrency bugs or increase the probability of manifesting concurrency bugs Park et al. (2009a, b). In addition, to help reproduce reported concurrency bugs, it is important for bug reporters to include more information about how to reproduce concurrency bugs. We also found that 55.5% of Mozilla concurrency bugs cause hangs or crashes, indicating that simple generic recovery techniques such as restart or rollback and re-execute can be effective for concurrency bugs (Section 9).

- **RQ7**: Is it possible to automatically identify memory bugs and semantic bugs from bug databases to reduce the manual effort in building bug benchmarks (Lu et al. 2005)?
  Bug benchmarks can help one evaluate bug detection and diagnosis tools (Tang et al. 2008); however, building bug bechmakrs requires a significant amount of manual effort to find many real-world bugs of certain types. In addition, bug benchmarks may help developers understand certain categories of bugs for better bug diagnosis and fixing. Different from previous work that automatically collects bugs from version control systems and test cases (Dallmeier and Zimmermann 2007) and records software development history to collect bugs (Edwards et al. 2010), we want to study how to automatically find bugs of certain types, e.g., memory bugs and semantic bugs. The reason is that typical bug detection and diagnosis tools can only address certain types of bugs. For example, if we compare several memory bug detection tools, we need to evaluate them on memory bugs only.
  We show that it is feasible to automatically identify memory bugs and semantic bugs in Mozilla bug database with reasonable precisions and recalls. This technique could reduce the manual effort in building bug benchmarks for evaluating bug detection and diagnosis tools (Section 10.2).

Overall, our results not only provide software development with a good understanding of software bugs, but also enlighten bug detection and recovery techniques, suggesting where effort should be put into.

The rest of the paper is laid out as follows. Section 2 describes how we collect bugs from various sources. Our bug taxonomies, classification, and analysis methods are presented in Section 3. Section 4 discusses root cause related findings and implications; Section 5 presents our results regarding impacts and their correlation with the root causes; and Section 6 shows the findings and implications related to software components, and their correlation with the root causes. The comparison between OS and non-OS bugs are summarized in Section 7. The discussion of security bugs and concurrency bugs is presented in Section 8 and Section 9 respectively. Section 10 describes our automatic bug classification approach and results. Related work is discussed in Section 11, and Section 12 concludes the paper. Appendix A describes how we combine the two data sets to maintain the pure randomness of sampling. Appendix B presents bug examples and their classification. Appendix C gives the detailed numbers for the bar graphs whose bars show the break down of different categories.

2 Bug Sources

To answer the research questions, we study bugs from three large widely-used open source projects, Mozilla, Apache, and the Linux kernel (Table 1). Bug reports (BR) opened between the dates shown in Column “Date of First BR” and “Sampling Date” are the population of bug reports studied in this paper.

Table 2 summarizes the number of bugs studied in this paper; the rest of this Section describes how we collect these bugs. In total, we have studied 2,060 real-world bugs in three large and popular open source projects. The 2,060 bugs come from three data sets: BugSet1—randomly sampled bugs from Bugzilla databases, BugSet2—all

Table 1 Software studied

Software	#BR	#Fixed BR	Date of first BR	Sampling date	Language	Description
Mozilla	515.8K	189.1K	Apr. 1998	Sep. 2010	C/C++	Browser suite
Apache	4.9K	1.5K	Jan. 2001	Sep. 2010	C	HTTP/Web server
Linux	17.8K	4.7K	Nov. 2002	Feb. 2010	C	Operating system

BR denotes bug reports

classified security bugs in the National Vulnerability Database (NVD 2013), and BugSet3—concurrency bugs in Bugzilla databases retrieved using keyword searches.

2.1 Randomly Collecting Bugs (**BugSet1**)

In our study, we focus on the characteristics of software defects that manifest at *runtime* (referred to as *runtime bugs*), that is, excluding new feature requests, compile-time errors, configuration errors, environmental errors, and software maintenance requests. We focus on runtime bugs because they can cause production software to fail, e.g., crashes and data corruption. To ensure correct classification, we only study *fixed* runtime bugs whose root causes can be identified from bug reports and the relevant commit messages, because unfixed bugs may be invalid and the root causes described in the reports can be wrong. Similarly, if the root causes cannot be identified from the bug report, we cannot correctly classify the bug report. In the rest of this paper, we simply use the term *fixed runtime bugs* to refer to fixed runtime bugs whose root causes can be identified from bug reports and the relevant commit messages.

In this way, we randomly sampled 635 fixed bug reports (Column “Sampled BR” in Table 2) from the Mozilla Bugzilla database (Mozilla.org Bugzilla 2013). As not every fixed bug report describes a valid runtime bug, we manually read the 635 bug reports and find that 339 of (Column “Bugs” in Table 2) them are fixed runtime bugs, and the rest are not, e.g., invalid bug reports, new feature requests, compile-time errors, bug reports with insufficient information, etc. We then manually classify the 339 bugs to the categories defined in Section 3.1. We first study the characteristics of these manually classified bug reports, and then use them to train and evaluate automatic classifiers for the whole Bugzilla database as described in Section 10.1. Similarly, we randomly sample 200 fixed bug reports from the Apache Bugzilla database (ASF Bugzilla 2013), and then manually classify 97 fixed runtime bugs.

To understand whether our findings apply to operating system code, we manually study fixed runtime bugs in the Linux kernel (Kernel Bug Tracker 2013) in a manner

Table 2 Bugs studied

Software	Sampled BR	BugSet1:	BugSet2:	BugSet3:
		Bugs	Security Bugs	Concurrency Bugs
Mozilla	635	339	640	90
Apache	200	97	65	/
Linux	300	147	682	/
Total	1,135	583	1,387	90
Sum of all bugs				**2,060**

BR denotes bug reports. “Bugs” is the number of bugs in the set of randomly “sampled BR” after our manual examination. “Security Bugs” and “Concurrency Bugs” are the number of security bugs and concurrency bugs that we studied in addition to the randomly sampled bugs that is similar to our non-OS bug collecting and analyzing process. We first randomly sampled 300 fixed bug reports from the Linux kernel Bugzilla database (Kernel Bug Tracker 2013). We then manually examine them, narrow down to 147 fixed runtime bugs. As we focus on studying bugs in high-level programming languages, we exclude bugs in assembly code (only 1 in our 300 sampled bug reports).

To show that we have sampled enough number of bug reports, we present the standard statistical margin of errors with a 95 % confidence level in our results. The margin of errors is reported as part of a range $x \pm i$, which means that if we obtain the value x in a random sample, the actual value x in the entire population will fall in the range $[x - i, x + i]$ with a probability of 95 %. For example, later Fig. 1 shows that there are 11.8±3.4 % memory bugs and 87.0±3.6 semantic bugs in Mozilla, indicating we are 95 % confident that in the entire Mozilla Bugzilla, the portion of memory bugs is between 11.8–3.4 % and 11.8+3.4 %, and the portion of semantic bugs is between 87.0–3.6 % and 87.0+3.6 %. The result indicates that it is statistically significant to state that semantic bugs dominate in Mozilla, because 87.0–3.6 % is still bigger than 11.8+3.4 %.

In the rest of this paper, we use the general name "bugs" to refer to *fixed runtime bugs*. Section 4, Section 5, and Section 6 present the findings and implications on the non-OS bugs. Section 7 summarizes the comparison between OS bugs and non-OS bugs. Section 10.2 describes our automatic classification technique that uses Mozilla bugs for building classifiers, and the relevant results.

2.2 Collecting Security Bugs (**BugSet2**)

We collect security vulnerabilities from the National Vulnerability Database (NVD 2013). Most NVD security vulnerabilities have already been classified by NVD into one of the NVD root cause categories, such as buffer errors, authentication issues, race conditions, SQL injection, etc. To ensure correctness of classification, we only study NVD security vulnerabilities that have already been classified into these categories. There are 640 security vulnerabilities in Mozilla, 65 security vulnerabilities in Apache, and 682 security vulnerabilities in the Linux kernel (Column "Security Bugs" in Table 2), and we study all of them. The discussion of security bugs is presented in Section 8.

2.3 Collecting Concurrency Bugs (**BugSet3**)

Concurrency bugs only constitute a small percentage of all reported bugs. Therefore, random sampling cannot provide enough concurrency bugs for a representative study. To collect enough concurrency bugs, we use keywords, i.e., "race", "lock", "deadlock", "synchronization", "starvation", and "atomic", to search bug reports and extract reports that contain these keywords as potential concurrency bugs. We then manually verify whether they are concurrency bugs. Using this method, we collect 90 concurrency bugs from Mozilla (Column "Concurrency Bugs" in Table 2). The Apache and Linux kernel Bugzilla databases only contain 1.5K–4.7K fixed bug reports, which are two orders of magnitude smaller than that of Mozilla (Table 1). In addition, there is only a small percentage of concurrency bugs as shown in Fig. 1a. Therefore, due to the small number of concurrency bugs in the Apache and Linux kernel Bugzilla databases, the results for Apache and the Linux kernel concurrency bugs are not discussed in this paper. The discussion of Mozilla concurrency bugs is presented in Section 9.

3 Bug Classification and Analysis

In this section, we first define the basic terminology used in this paper, and describe the bug categories. We then describe how we study the trend and the correlation of different bug categories. Lastly, we discuss the threats to validity.

3.1 Terminology and Bug Taxonomies

For the definitions of our basic terminology, we follow the literature (Avizienis et al. 2004). Specifically, a **failure** is "an event that occurs when the delivered service deviates from correct service" (Avizienis et al. 2004). Crashes and performance degradation are examples of software failures. A **fault** is the cause of a failure, and

Table 3 Bug categories of the three dimensions: Root cause, impact, and component

Dimension	Category	Description	Abbreviation
Root cause	Memory	Bugs caused by improper handling of memory objects.	Mem
	Concurrency	"Synchronization problems among the concurrent tasks in concurrent programs" (Lu 2008), including data races and deadlocks.	Con
	Semantic	Inconsistencies with the requirements or the programmers' intention that do not belong to the categories above.	Sem
Impact	Hang	Program keeps running but does not respond.	Hang
	Crash	Program halts abnormally.	Crash
	Data corruption	Mistakenly change user data.	Corrupt
	Performance degradation	Functions correctly but runs/responds slowly.	Perf
	Incorrect functionality	Not behaving as expected.	Func
	Other	Bugs that cause other impacts.	OtherImp
Software component (Mozilla & Apache)	Core	Bugs related to core functionality implementations.	Core
	GUI	Bugs related to graphical user interfaces.	GUI
	Network	Bugs related to network environment and network communication	Network
	I/O	Bugs related to I/O handling.	I/O
OS component (Linux)	Drivers	Bugs related to device drivers.	Drivers
	Core	Bugs in the kernel directory 'mm', 'kernel', and 'include'.	Core
	Network	Bugs related to network environment and network communication.	Network
	File system	Bugs related to file system.	FS
	Architecture	Bugs related to hardware architecture.	Arch

"**bugs**" and "**defects**" are synonyms of faults (Avizienis et al. 2004). Failures are the observable impact of software faults.

Classifying OS and non-OS Bugs (*BugSet*1) Based on our experience with many real-world bug reports and the inspirations from *BugBench* (Lu et al. 2005) and the software testing book by Beizer (1990), we designed bug taxonomies in three dimensions, **Root Cause** (the fault), **Impact** (the failure caused by the bug), and **Component** (the location of the bug). Each bug is classified in all three dimensions. For example, a bug could be a semantic bug according to the root cause dimension, a performance bug according to the impact dimension, and a GUI bug according to the component dimension.

According to root causes, we classify bugs into three disjoint categories, *Memory Bugs*, *Concurrency Bugs*, and *Semantic Bugs*, whose definitions are shown in Table 3. To ensure disjoint classification, we give these categories an order shown in the Table. A bug that belongs to a higher category will not be considered for a lower category. An alternative is to allow one bug to belong to multiple categories. It is uncommon for one bug to belong to multiple root cause categories (no such cases were found in our random samples); therefore either design choice should produce similar results. Memory bugs and semantic bugs are further classified into subcategories shown in Table 4.

In addition, Table 3 shows the impact and component categories and their definitions. Regarding components, we use different taxonomies for OS bugs and

Table 4 Subcategories of memory and semantic bugs

Category	Subcategory	Description	Abbreviation
Memory Bug	Memory leak	Failures to release unused memory.	MLK
	Uninitialized memory read	Read memory data before it is initialized.	UMR
	Dangling pointer	Pointers still keep freed memory addresses.	Dangling
	NULL pointer dereference	Dereference of a null pointer.	NULL
	Overflow	Illegal access beyond a buffer boundary.	Overflow
	Double free	One memory location is freed twice.	2Free
	Other	Other memory bugs.	OtherMem
Semantic Bug	Missing features	A feature is supposed to be but is not implemented.	MissF
	Missing cases	A case in a functionality is not implemented.	MissC
	Corner cases	Some boundary cases are considered incorrectly or ignored.	CornerC
	Wrong control flow	The control flow is incorrectly implemented.	CtrlFlow
	Exception handling	Does not have proper exception handling.	Except
	Processing	Processing such as evaluation of expressions and equations is incorrect.	Process
	Typo	Typographical mistakes.	Typo
	Other wrong functionality implementation	Any other semantic bug that does not meet the design requirement.	FuncImpl

non-OS bugs as operating system components are inherently different from non-OS software: we classify non-OS bugs into the categories defined in the **Software Component** dimension, and the **OS Component** categories are used for OS bugs. Different from root causes, it is common for bugs to have multiple failure symptoms (multiple impact categories). For example, a bug can cause a crash and data corruption. When we count the impact distribution, multiple impact categories per bug are considered, this is why the percentages may add to over 100 %. Specifically, 11 Mozilla bugs have more than one impact category, 17 Linux kernel bugs have more than one impact category, and no Apache bugs have more than one impact category in our random samples (BugSet1). Similarly, one bug may require fixes across several components (multiple component categories). In our random samples (BugSet1), two Mozilla bugs have more than one component category, four Apache bugs have more than one component category, and one Linux kernel bug has more than one component category.

The authors read the bug reports to determine the categories of each bug report. When the information in the bug reports is not enough to make a decision, the authors consult the commits related to the bug report in a manner similar to prior work (Bird et al. 2009; Kim et al. 2006; Śliwerski et al. 2005; Wu et al. 2011). Specifically, if a bug report contains a link to the commit or a commit ID, we simply follow the link or use the commit ID to locate the commit in the software repository. Otherwise, we search the bug ID or the bug report title in the software repository. As discussed before, we only study bugs whose root causes can be identified from these sources to ensure correct classification. We minimize the subjectivity in manual examination through double verification: each bug report is examined at least twice by two different people independently. If they disagree, they will discuss and reach a consensus. Appendix B presents bug examples and their classification. In addition, we apply *machine learning* techniques to automatically classify around **109,014** bugs (Section 10).

Classifying Security Vulnerabilities (BugSet2) Security vulnerabilities are classified in three dimensions. The root cause dimension is the same as that of OS and non-OS bugs. As mentioned earlier, to ensure correctness of classification, we only study NVD security vulnerabilities that have already been classified into NVD root cause categories. We then map these NVD root cause categories to our root cause categories—memory bugs, semantic bugs, and concurrency bugs. Similar to general bug reports, we focus on runtime bugs whose root causes are known and exclude vulnerabilities of the following NVD root cause categories—Insufficient Information, Not in CWE, Configuration, and Other. NVD database lists the categories for two additional dimensions—impacts and severity. Based on impact, vulnerabilities are classified into four categories as shown in Table 5, *confidentiality* (allows unauthorized disclosure of information), *integrity* (allows unauthorized modification), *availability* (allows disruption of service), and *access* (provides unauthorized access). The third dimension is the NVD severity, which contains three levels, *High*, *Medium*, and *Low*, as defined in the NVD databases based on the Common Vulnerability Scoring System (CVSS) scores (Nvd Common Vulnerability Scoring System 2013). We automatically collect the classifications of

Table 5 Bug categories for security bugs of the impact dimension

Category	Description
Confidentiality	Allows unauthorized disclosure of information
Integrity	Allows unauthorized modification
Availability	Allows disruption of service
Access	Provides unauthorized access

NVD root causes, impacts, and severity, as the classifications are already available in the NVD database.

3.2 Trend of Different Categories of Bugs

Previous work investigates the trend of bug density over releases (Ostrand and Weyuker 2002). It would be interesting to zoom in and learn the trend of different categories of bugs, e.g., how the number of memory bugs and semantic bugs change over time, which would help us answer our first research question. To do so, we calculate the percentage of memory bugs and semantic bugs reported in each year.

3.3 Correlation of Bug Categories

To study the correlation between two categories in different dimensions, we use a statistical metric called *lift* (Han and Kamber 2001). The *lift* of category A_i in dimension A and category B_j in dimension B, $lift(A_i, B_j)$, is calculated as $\frac{P(A_iB_j)}{P(A_i)*P(B_j)}$, where $P(A_iB_j)$ is the probability that a bug belongs to both category A_i and B_j. For example, if there are a total of 100 bugs, 10 of which are memory bugs, 20 of which cause crashes, and 5 of which are memory bugs that cause crashes, we can calculate the *lift* correlation as follows. $P(A_iB_j)$, where A_i is memory bugs and B_j is crashes, is 5/100. $P(A_i)$ is 10/100, and $P(B_j)$ is 20/100. The lift correlation $lift(A_i, B_j) = \frac{P(A_iB_j)}{P(A_i)*P(B_j)} = \frac{5/100}{(10/100)*(20/100)} = 500/200 = 2.5$.

If $lift(A_i, B_j)$ is equal to 1, it means $P(A_iB_j) = P(A_i)P(B_j)$, which indicates that category A_i and B_j are not correlated. If it is greater than 1, category A_i and B_j are positively correlated, which means that if a bug belongs to A_i, it is more likely to also belongs to B_j. Symmetrically, if it is less than 1, A_i and B_j are negatively correlated, which means that if a bug belongs to A_i, it is less likely to also belongs to B_j. In the example above, since $lift(A_i, B_j)$ is 2.5, we can learn that memory bugs are more likely to cause crashes, and that crashes are more likely to be caused by memory bugs. Specifically, in the entire population of 100 bugs, the probably of a bug causing crashes is 20/100, which is 20 %. But if we know that a bug is a memory bug, the probably of that bug causing a crash is 5/10 (50 %), which is much higher than the 20 % average in the population. Similarly, in the entire population of 100 bugs, the probably of a bug being a memory bug is 10/100, which is 10 %. However, if we know that a bug causes crashes, then the probably of that bug being a memory bug is 5/20 (25 %), which is much higher than the 10 % average in the population.

3.4 Threats to Validity

While we believe that the bugs from the examined open source software well represent bugs in many software projects, we do not intend to draw any general conclusions about bugs in all software. Similar to any characteristic study, our findings should be considered together with our evaluation methods.

All three subject systems are open source projects developed in C/C++. Therefore, our results may not generalize to commercial software or software written in other programming languages.

Since we randomly sample bug reports from Bugzilla databases, the distribution of bug categories may be different in the entire Bugzilla databases from that of our sample. However, the reported margins of error give us confidence that the disagreement would be low. For some subcategories, the confidence intervals are large, e.g., for the memory leak (MLK) categories and Null pointer dereference categories (NULL). Therefore, we do not have 95 % confidence that the results regarding these categories hold in the entire Bugzilla databases. However, our findings regarding our research questions RQ1–RQ7 (other than the ones about MLK and NULL mentioned above) are based on main categories with small confidence intervals, e.g., 11.8 ± 3.4 % memory bugs in Mozilla. Therefore, these results should hold in the entire Bugzilla databases with 95 % confidence.

Some bugs are reported in mailing lists instead of the Bugzilla databases, or are fixed in the source code repository directly without being recorded in the Bugzilla databases. As we only study bugs in the Bugzilla databases, we did not sample from all the possible bugs in the evaluated software, which is prohibitively expensive, if possible. However, as the Bugzilla databases contain up to hundreds of thousands of bug reports, and are actively updated, bugs we study should be a representative view of post-release software bugs.

Since we examined bug reports manually, subjectivity is inevitable. However, we tried our best to minimize such subjectivity through double verification: each bug report is examined at least twice by two different people independently. If they disagree, they will discuss and reach a consensus.

We compare our root cause distribution results against previous work. For example, semantic bugs account for 87.0 % in Mozilla and 82.5 % in Apache, which are much higher than the 55–66 % reported in a previous study (Sullivan and Chillarege 1992). While we tried our best to map their bug types with ours, it is possible that our definition of semantic bugs is different from theirs. However, due to the big difference in the percentage numbers, it is unlikely that the mismatch in the definitions of semantic bugs will revert the "higher than" relation. In addition, our trend results show that semantic bugs increase as the evaluated software becomes mature.

When a bug report does not contain enough information for classification, the authors consult the commits related to the bug report. Specifically, if a bug report contains a link to the commit or a commit ID, we simply follow the link or use the commit ID to locate the commit in the software repository. Otherwise, we search the bug ID or the bug report title in the software repository. A prior study (Bird et al. 2009) shows that relying on that a commit message contains a bug ID may miss related commits. To minimize such misses, we also follow the link in the bug report and search for the bug report title in the software repository. In addition, most of the bug reports contain enough information for us to classify, and do not require us to look for the related commits; therefore, the impact of missing links should be small.

Some security bugs are initially only accessible by authorized users to prevent attackers from exploiting the security vulnerabilities before they are fixed. The private security bugs are often made publicly available after they have been fixed.

Therefore, the actual number of reported security bugs in the recent years are likely to be bigger than the number of security bugs we studied.

Similar to another concurrency bug study (Lu et al. 2008), we use keywords to search for more concurrency bugs, which may miss some concurrency bugs in the Bugzilla databases. Since it is prohibitively expensive to manually examine all hundreds of thousands of bug reports in the bug databases, keyword search is our best effort approach to understand reported concurrency bugs.

4 Root Cause Analysis

This section answers **RQ1**: As software becomes mature, what are the dominant types of bugs, e.g., memory bugs, concurrency bugs, or semantic bugs? In addition, we study the subcategories of memory bugs and semantic bugs. The bug set used for this section is **BugSet1**—randomly sampled 583 bugs from Bugzilla databases (Section 2.1).

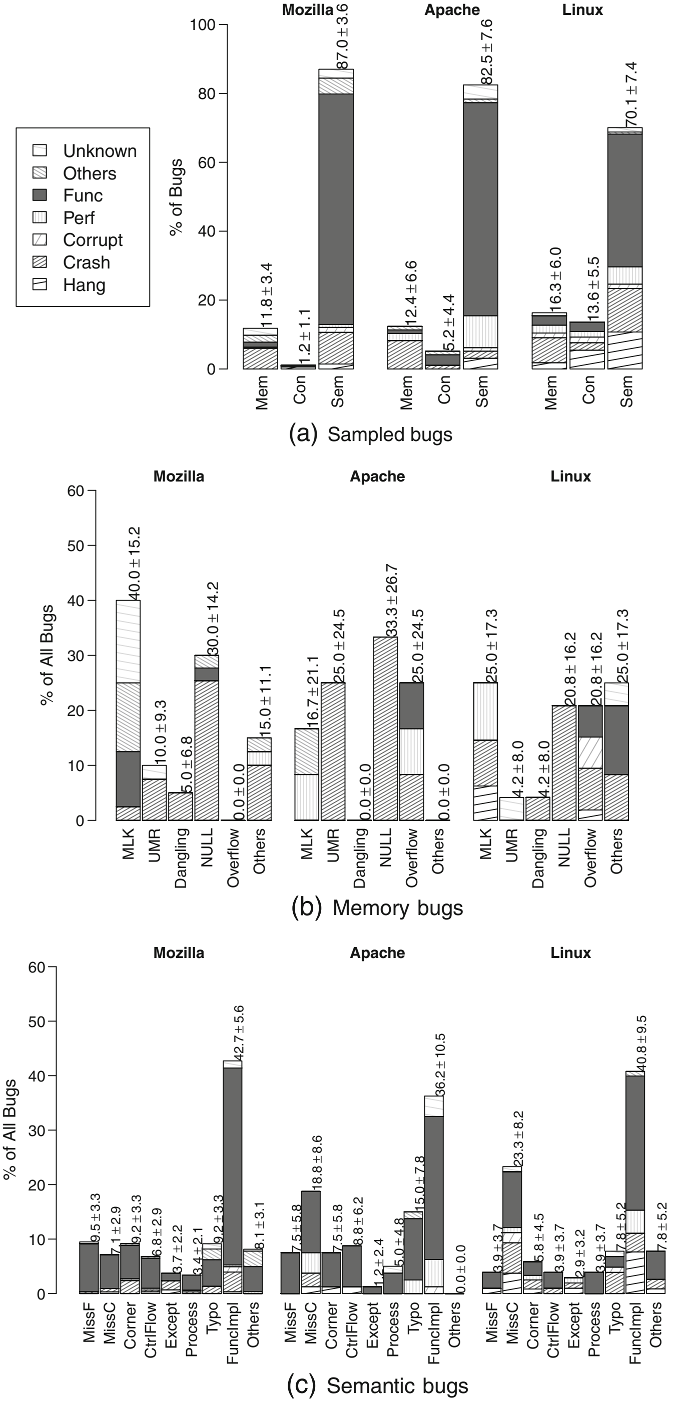
Figure 1 summarizes the distribution of bugs with different root causes and their corresponding impacts on the bug sets that are randomly sampled and manually analyzed by us. The statistical margin of error is shown with a 95 % confidence level. While the bar graphs can better visualize the big picture (e.g., which categories are dominant), we provide the detailed numbers in the tables in Appendix C.

Figure 2 shows how the percentage of memory bugs and semantic bugs changes over time on the bug sample in Mozilla and the Linux kernel. Since there is only a small number of concurrency bugs (Fig. 1), there is not enough data to study the trend of concurrency bugs. As described in Section 2, we use keyword based approach to study concurrency bugs (BugSet3), the results are presented in Section 9. To focus on memory and semantic bugs, the triangles are the ratio of *semantic bugs* to the sum of semantic bugs and memory bugs; and the circles are the ratio of *memory bugs* to the sum of semantic bugs and memory bugs. Since the first year and last year of each project contain only a few months' data, they are excluded from the figures (1998 and 2010 for Mozilla; and 2002 and 2010 for the Linux kernel). If there is only a couple of bugs per year, then the trend results are not meaningful; therefore, we also show the total number of semantic and memory bugs per year and the number of sampled bug reports per year in Fig. 2c and d. There are dips in these two figures (e.g., year 2003–2005 for Mozilla), which are in proportion to the total number reported bug reports in the entire Bugzilla databases of the corresponding years. We did not perform a trend study on Apache due to the relatively small number of bug reports in Apache (only 0–2 memory bugs per year in our random sample BugSet1).

From Figs. 1 and 2, we can learn the following findings and implications.

4.1 Memory Bugs have been Decreasing

Figure 2 shows that as Mozilla and the Linux kernel become mature and stable, their memory bugs decrease. Figure 1a shows that memory bugs account for a small fraction of all bugs, 11.8 % in Mozilla, 12.4 % in Apache, and 16.3 % in the Linux kernel. These percentages are much lower than the 28–38 % reported in previous work (Sullivan and Chillarege 1991, 1992). One possible reason for this reduction could be the use of debugging tools during the development process in recent years.



(a) Sampled bugs

(b) Memory bugs

(c) Semantic bugs

Fig. 1 Distribution of root causes with impacts (BugSet1). The numbers show 95 % confidence intervals. ($x \pm i$ means that the value would fall in the range $[x - i, x + i]$ with a probability of 95 % based on our sampling.)

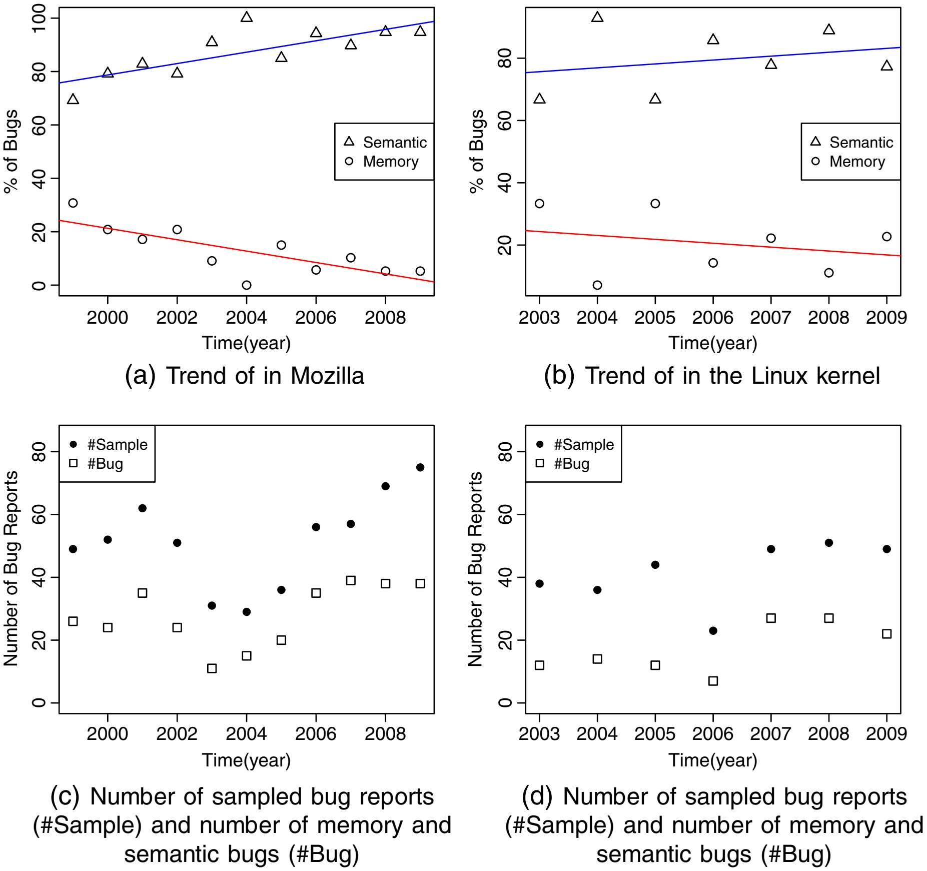


Fig. 2 Trend of memory and semantic bugs in Mozilla and the Linux kernel from the random sample (BugSet1)

A developers' document (Debugging Memory Leaks 2013) indicates that recent debugging tools, such as Coverity (2013), Hastings and Joyce (1992), and Valgrind (2013), are used by Mozilla developers during the development process, and the Mozilla developers have even augmented tools including Purify for more effective memory bug detection. As part of the Mozilla software process, "static analysis tools (Coverity and another similar tool) were run over the codebase, ..." as pointed out by a known Mozilla developer in the mozilla dev-quality mailing list. We have spotted bug reports and commit messages written by Apache and Linux kernel developers stating that they used tools such as Coverity (2013) and Valgrind (2013) to catch bugs.

Figure 1b shows that among memory bugs in Mozilla, Apache, and the Linux kernel, NULL pointer dereference is a major cause, accounting for 20.8–33.3% of the memory bugs, and most of them result in a crash. Memory leak is another major cause in the three projects, accounting for 16.7–40.0 %, which is much more than the 8 % reported previously (Sullivan and Chillarege 1992). Since many of these memory bugs can be detected by the existing tools such as Hastings and Joyce (1992), Valgrind (2013), and Coverity (2013), and developers indicated that they used these tools during the development process as shown earlier, our results indicate that these debugging tools have not been used in the *development* process with their *full* capacity yet. Therefore, it is important to understand why these tools have not been used with their full capacity and address the relevant issues, e.g., improving the detection capability of the current bug detection techniques, reducing false positives, simplifying the usage procedure, promoting these tools to more developers, etc.

4.2 Semantic Bugs are Dominant Root Causes

Figure 1a shows that the dominant root causes are semantic bugs in Mozilla, Apache, and the Linux kernel, accounting for 87.0 % in Mozilla, 82.5 % in Apache, and 70.1 % in the Linux kernel. These numbers are much higher than the 55–66 % reported in a previous study (Sullivan and Chillarege 1992) (bugs excluding memory bugs and concurrency bugs). Figure 2 shows that semantic bugs increase, as Mozilla and the Linux kernel become mature and stable. In summary, semantic bugs are not only the dominant root causes, but also have become more dominant.

One possible reason may be that most semantic bugs are application-specific, and thus a programmer can easily introduce semantic bugs due to a lack of thorough understanding of the system, its requirements, or its specifications. Additionally, it is harder to automatically detect semantic bugs because they are application-specific, which is different from memory bugs which are general across applications.

In order to further understand what are the major causes among semantic bugs, we show the breakdown of semantic bugs into subcategories in Fig. 1c. Most semantic bugs are caused by wrong functionality implementation that does not meet the design requirements. In addition, missing features and missing cases also account for a large portion of semantic bugs, which is consistent with the previous study (Chillarege et al. 1991). Since knowledge about the target system is critical for avoiding and detecting such semantic bugs, these results suggest that it would be beneficial to develop techniques to automatically extract specifications from programs (Ernst et al. 2001; Tan et al. 2007, 2011).

Interestingly, there are quite a few *simple* semantic bugs in the evaluated software. For example, typos account for 7.8–15.0 % of the semantic bugs in Mozilla, Apache and the Linux kernel. It indicates that careless programming is causing many bugs, suggesting that the development environment should provide some tools for programmers to check for simple errors such as typos.

5 Impact Analysis

In this section, we study the distribution of impacts and the correlation between impacts and root causes. We answer **RQ2**: What are the impact of bugs and what types of bugs have severe impact? The bug set used for this section is **BugSet1**—randomly sampled 583 bugs from Bugzilla databases (Section 2.1).

5.1 Impact Distribution

Figure 3 summarizes the distribution of different impacts with the corresponding root causes. It shows that incorrect functionality is the dominant impact in Mozilla, Apache, and the Linux kernel: the percentage of incorrect functionality is

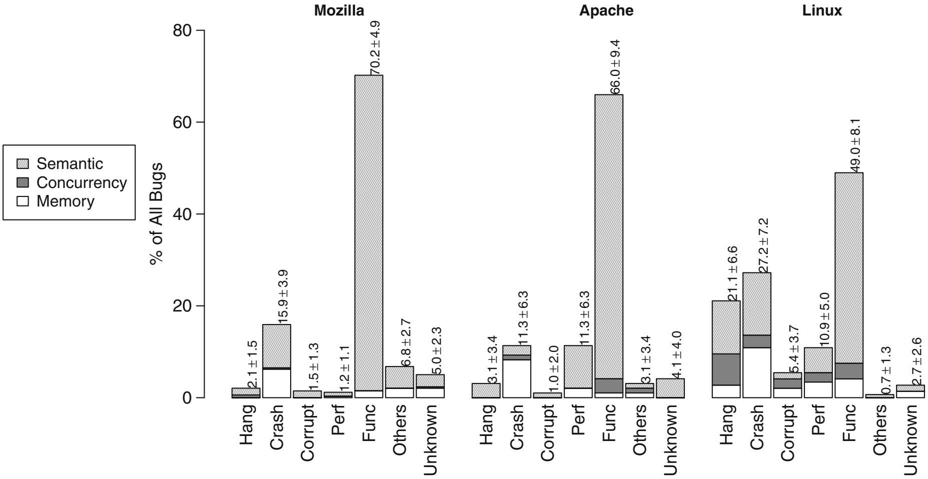


Fig. 3 Distribution of bug impacts (BugSet1)

49.0–70.2 %, much larger than the 35 % reported in the previous work (Sullivan and Chillarege 1991).

The severe impacts that compromise software availability, i.e., crashes and hangs, account for 14.4–48.3 % of bugs in the three projects, which is a considerable portion. Thus, generic recovery techniques such as restart or checkpoint and replay are still needed to provide highly available services.

5.2 Correlation between Causes and Impacts

Figure 3 also shows that the major cause of crashes in Mozilla, Apache, and the Linux kernel is memory bugs, accounting for 38.9–72.7 %, which is similar to what has been found in the previous work (Sullivan and Chillarege 1991). Semantic bugs are more likely to cause incorrect functionality, accounting for 84.7–97.9 %, which is

Table 6 Correlation between root causes and impacts in Mozilla (BugSet1)

Impact	Memory	Concurrency	Semantic	Memory subcategories		
				MLK	UMR	NULL
Hang	0.00	**24.21**	0.82	0.00	0.00	0.00
Crash	**3.30**	**1.57**	0.68	0.39	**4.71**	**5.75**
Func	0.18	0.00	**1.13**	0.36	0.00	0.12

Impact	Semantic subcategories							
	MissF	MissC	CornerC	CtrlFlow	Except	Process	Typo	FuncImpl
Hang	0.00	**2.31**	0.00	**2.42**	**8.81**	0.00	0.00	0.38
Crash	0.22	0.60	**1.63**	0.31	**2.85**	0.63	0.93	0.55
Func	**1.32**	**1.36**	0.95	**1.21**	0.52	**1.28**	0.79	**1.24**

The correlation metric *lift* is defined in Section 3.3. Categories with too few examples (fewer than 3) are not shown due to statistical insignificance. MLK is memory leak, UMR is uninitialized memory read, and NULL is NULL pointer dereference

Table 7 Correlation between root causes and impacts in Apache (BugSet1)

Impact	Memory	Concurrency	Semantic	Memory subcategories	
				UMR	NULL
Hang	0.00	0.00	**1.21**	0.00	0.00
Crash	**5.88**	**1.76**	0.22	**8.82**	**8.82**
Func	0.13	0.91	**1.14**	0.00	0.00

Impact	Semantic subcategories						
	MissF	MissC	CornerC	CtrlFlow	Process	Typo	FuncImpl
Hang	0.00	**2.16**	**5.39**	**4.62**	0.00	0.00	0.00
Crash	0.00	**1.18**	0.00	0.00	0.00	0.00	0.00
Func	**1.52**	0.91	**1.26**	**1.30**	**1.14**	**1.14**	**1.10**

The correlation metric *lift* is defined in Section 3.3. Categories with too few examples (fewer than 3) are not shown due to statistical insignificance. UMR is uninitialized memory read, and NULL is NULL pointer dereference

also consistent with the previous studies. However, 18.2–59.3 % of crashes are caused by semantic bugs, which is higher than that reported in the previous work (Sullivan and Chillarege 1991). It indicates that although most semantic bugs result in incorrect functionality, they are also one of the important factors of causing unavailability.

In order to further understand the correlation between causes and impacts, we show the correlation metric *lift* (defined in Section 3.3) in Tables 6, 7, and 8. Numbers greater than 1 (positive correlation) are shown in bold. Not surprisingly, the crash symptom have a strong correlation with memory bugs, while the hang symptom has an extremely strong correlation with concurrency bugs. The incorrect functionality impact (Func) has a positive correlation with semantic bugs, though no specific subcategory of semantic bug has an exceptionally high correlation.

6 Bugs in Different Components

In this section, we study in which software components the dominant bugs are located, including core, GUI, network, I/O, and others. Since friendly user interfaces

Table 8 Correlation between root causes and impacts in the Linux kernel (BugSet1)

Impact	Memory	Concurrency	Semantic	Memory subcategories	
				MLK	NULL
Hang	0.79	**2.37**	0.78	**2.37**	0.00
Crash	**2.45**	0.73	0.71	**2.45**	**3.67**
Func	0.51	0.51	**1.21**	0.00	0.00

Impact	Semantic subcategories							
	MissF	MissC	CornerC	CtrlFlow	Except	Process	Typo	FuncImpl
Hang	**1.19**	0.79	0.79	0.00	**1.58**	0.00	0.00	**1.02**
Crash	0.00	0.92	**1.23**	0.92	**1.23**	0.00	**1.84**	0.35
Func	**1.53**	0.94	**1.02**	**1.53**	0.00	**2.04**	0.51	**1.41**

The correlation metric *lift* is defined in Section 3.3. Categories with too few examples (fewer than 3) are not shown due to statistical insignificance. MLK is memory leak, and NULL is NULL pointer dereference

Table 9 Correlation between root causes and software components in Mozilla (BugSet1)

Component	Memory	Concurrency	Semantic	Memory subcategories		
				MLK	UMR	NULL
Core	**1.89**	**1.40**	0.87	**1.93**	**1.40**	**2.10**
GUI	0.42	0.52	**1.09**	0.26	**1.04**	0.35

Component	Semantic subcategories							
	MissF	MissC	CornerC	CtrlFlow	Except	Process	Typo	FuncImpl
Core	0.70	0.67	**1.35**	**1.26**	0.76	0.56	0.62	0.87
GUI	**1.34**	**1.09**	0.77	0.83	0.95	**1.04**	**1.31**	**1.11**

The correlation metric *lift* is defined in Section 3.3. Categories with too few examples (fewer than 3) are not shown due to statistical insignificance. MLK is memory leak, UMR is uninitialized memory read, and NULL is NULL pointer dereference

are becoming increasingly important for software, we are particularly interested in studying characteristics of GUI bugs. We answer the following research questions: **RQ3a**: Are there many GUI-related bugs? **RQ3b**: Since GUI modules and their development process are quite different from other modules, do they have different root cause distributions? The bug set used for this section is **BugSet1**—randomly sampled 583 bugs from Bugzilla databases (Section 2.1). As operating system components are inherently different from non-OS software components, we present and discuss the component-related results for Mozilla and Apache in this Section, and show the results for the Linux kernel in Section 7.

Figure 4 shows the distribution of bugs within different components and their impacts. As we can see, GUI modules are critical for software reliability in graphical interface software Mozilla. GUI bugs account for 50.1 % of bugs in Mozilla and also cause around 45.6 % of Mozilla crashes. It is understandable that Apache contains a smaller percentage of GUI bugs as it is mostly a command-line server application.

The correlation between software components and root causes for Mozilla and Apache are shown in Tables 9 and 10. The *lift* numbers greater than 1 (positive correlation) are shown in bold. Interestingly, the results indicate that bugs in GUI and core modules have quite different root causes. The major root cause of bugs in core modules is memory related, while that of GUI bugs is semantic bugs. Such difference is likely because the GUI modules and their development process are quite different from other modules.

Table 10 Correlation between root causes and software components in Apache (BugSet1)

Component	Memory	Concurrency	Semantic	Memory subcategories	
				UMR	NULL
Core	**1.43**	0.63	0.96	**1.56**	**1.56**
GUI	0.00	0.00	**1.21**	0.00	0.00

Component	Semantic subcategories						
	MissF	MissC	CornerC	CtrlFlow	Process	Typo	FuncImpl
Core	**1.04**	**1.25**	**1.04**	**1.12**	**1.17**	**1.04**	0.70
GUI	**4.04**	0.00	0.00	0.00	0.00	**4.04**	0.84

The correlation metric *lift* is defined in Section 3.3. Categories with too few examples (fewer than 3) are not shown due to statistical insignificance. UMR is uninitialized memory read, and NULL is NULL pointer dereference

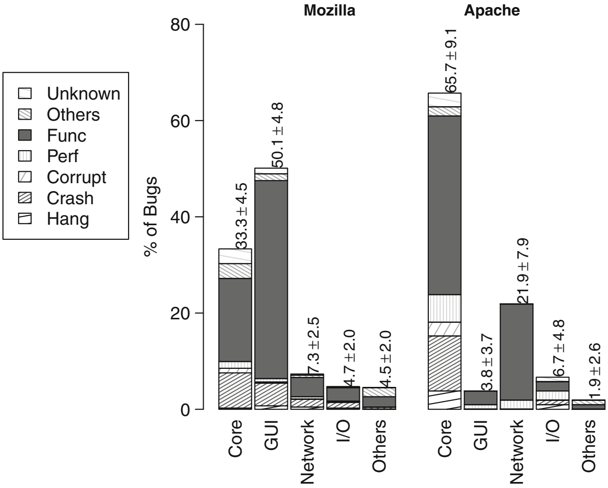


Fig. 4 Distribution of Mozilla and Apache bugs in software components (BugSet1)

In summary, Mozilla contains a significant number of GUI bugs, which can cause severe damage including crashes. The result calls for more support for GUI-related testing and debugging for GUI applications. In addition, Mozilla and Apache GUI bugs are more likely to be caused by semantic bugs instead of memory bugs. Therefore, the existing debugging tools aiming at memory bugs may be unsuitable for GUI bugs, while support for semantic bugs (Amir and Tao 2005) can be helpful for GUI bugs. In the future, it would be interesting to study what percentage of Mozilla and Apache code is GUI-related to understand the density of GUI bugs.

7 Bugs in Operating Systems

We are interested in learning the similarity and difference between OS bugs and non-OS bugs. In this section, we answer **RQ4**: Does operating system code such as the Linux kernel have different characteristics from its non-OS counterparts, and require different bug detection and diagnosis techniques? As operating systems inherently need to manage shared resources, do operating system code contain more concurrency bugs? The bug set used for this section is **BugSet1**—randomly sampled 583 bugs from Bugzilla databases (Section 2.1).

Similar Percentage of Memory Bugs Among all the studied Linux kernel bugs, 16.3 % are memory bugs (Fig. 1a). This result is similar to that for Mozilla (11.8 %) and Apache (5.2 %). Although it may be impractical to run dynamic memory bug detection tools such as Purify and Valgrind on the Linux kernel due to the prohibitively expensive runtime overhead, a static tool developed by Engler et al. (2001) can and are used by the kernel developers. This static tool become available around 2001 and we have spotted bug reports and commit messages written by the kernel developers stating that they used this tool to catch memory bugs.

More Concurrency Bugs and Hang Bugs Figure 1a shows that the Linux kernel has significantly more concurrency bugs (13.6 %) than Mozilla (1.2 %) and Apache (5.2 %). This difference is understandable as the Linux kernel inherently have to deal with concurrent activities and shared resources. By and large, the impact distribution of OS bugs is similar to non-OS bugs (Fig. 3). The only main difference is that the Linux kernel has many more *hang* bugs (21.1 %) than non-OS software (2.1–3.1 %). As concurrency bugs are more likely to cause deadlocks or hangs (discussed later this Section) and the Linux kernel contains more concurrency bugs, the difference in the number of hang bugs is not surprising.

Dominating Driver Bugs Figure 5 shows that 52.9 % of the bugs are in device driver code, which confirms the results from prior studies (Chou et al. 2001; Swift et al. 2003). Previous study (Chou et al. 2001) studied *injected* bugs in the Linux kernel. In contrast, we study field kernel bugs that are *reported* by users and developers to show that driver bugs dominate.

Interrupt-related Bugs A significant proportion (10.2 %) of the kernel bugs are related to interrupt handling, e.g., forgetting to disable or enable interrupts, missing certain interrupts, not ignoring certain interrupts, etc. Among these interrupt-related bugs, 20.0 % of them are concurrency bugs, and 80.0 % are semantic bugs. As correct interrupt handling are important for the correctness of an operating system and the applications running on top of the OS, and they cause severe damage, such as hang, crash, performance degradation and wrong functionality, we may want to provide more support to help developers write correct interrupt related code.

Correlation Between Causes, Impacts, and Components Table 8 shows that similar to non-OS software, the hang symptom has a strong correlation with concurrency bugs, while the crash symptom has a strong correlation with memory bugs. The incorrect functionality impact has a relatively strong correlation with semantic bugs. These results are similar to our non-OS bug results (Tables 6 and 7).

Table 11 shows that null pointer dereference bugs have a strong correlation with the file system component. This implies that it can be beneficial to apply null pointer

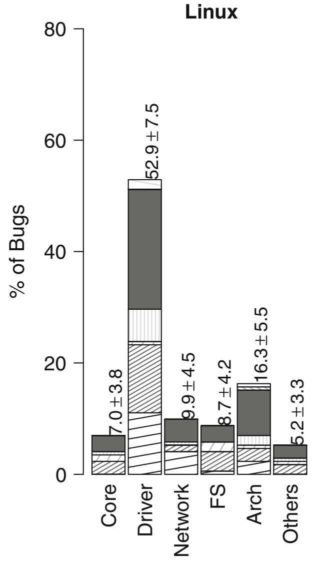


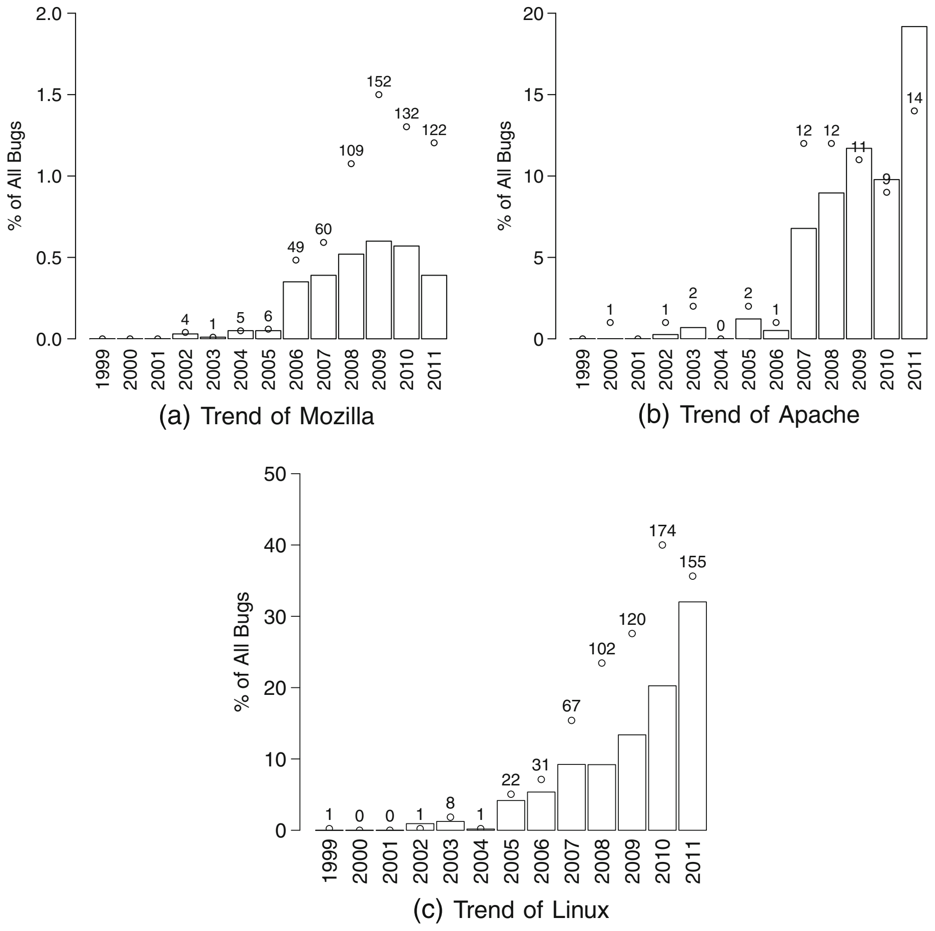
Fig. 5 Distribution of Linux kernel bugs in OS components (BugSet1). *FS* stands for file systems. *Arch* denotes architecture. *Core* includes bugs in 3 kernel directories: 'kernel', 'mm' (memory management), and 'include'

Table 11 Correlation between root causes and components in the Linux kernel

Component	Memory	Concurrency	Semantic	Memory subcategories		
				MLK	Overflow	NULL
Drivers	0.86	0.75	**1.08**	0.94	**1.13**	0.38
Core	**1.84**	**2.21**	0.57	**2.45**	0.00	0.00
Network	0.00	**2.30**	0.98	NA	NA	NA
FS	**1.31**	**2.10**	0.71	0.00	0.00	**6.30**
Arch	**1.11**	0.33	**1.10**	**1.11**	**1.34**	0.00

The correlation metric *lift* is defined in Section 3.3. Categories with too few examples (fewer than 3) are not shown due to statistical insignificance. MLK is memory leak, and NULL is NULL pointer dereference

dereference bug detection techniques specifically on the file system module instead of blindly on the entire Linux kernel. Such an approach may increase the scalability of the bug detection techniques. Similarly, we may want to focus our concurrency bug detection techniques on the network and file system modules.



(a) Trend of Mozilla

(b) Trend of Apache

(c) Trend of Linux

Fig. 6 Trend of security related bugs collected from NVD (BugSet2)

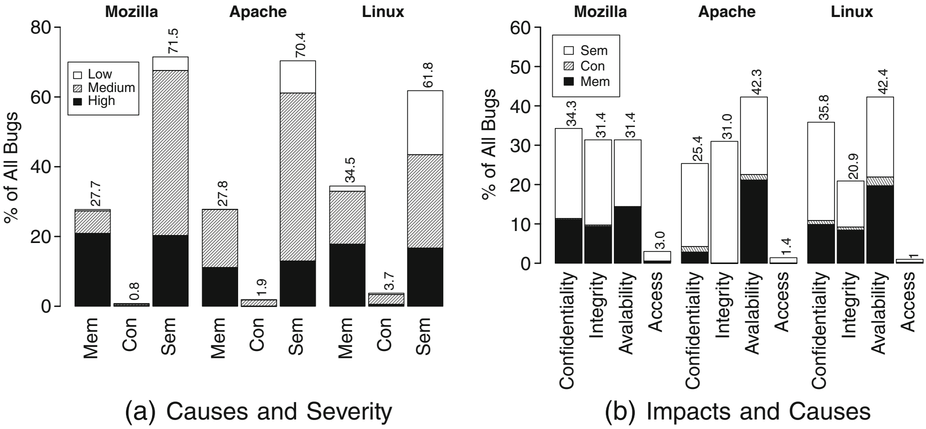


Fig. 7 Distributions of causes and impacts of security related bugs in NVD (BugSet2)

8 Security Related Bugs

We answer the following research questions: **RQ5a**: Does the number and proportion of reported security bugs increase? **RQ5b**: Is the common belief "buffer overflows are the most common type of security vulnerabilities" (Cowan et al. 2000) true? If not (as suggested by a recent study (Neuhaus and Zimmermann 2010)), what types of bugs cause more security vulnerabilities? Answers to these questions can provide guidance in addressing the important types of security vulnerabilities.

The bug set used for this section is **BugSet2**—all 1,387 classified security bugs in Mozilla, Apache, and the Linux kernel in the National Vulnerability Database (NVD 2013) (Section 2.2). Most NVD security vulnerabilities have already been classified by NVD into one of the NVD root cause categories, such as buffer errors, authentication issues, race conditions, SQL injection, etc. To ensure correctness of classification, we only study NVD security vulnerabilities that have already been classified into these categories: 640 security vulnerabilities in Mozilla, 65 security vulnerabilities in Apache, and 682 security vulnerabilities in the Linux kernel (Column "Security Bugs" in Table 2).

8.1 Security Related Bugs have been Increasing

Figure 6a shows the number of vulnerabilities over time in Mozilla (the dots). We also normalize the numbers to relative percentage by comparing with the total fixed

Table 12 Correlation between root causes and severity/impacts for security related bugs (BugSet2) in Mozilla

Cause	Severity			Impact			
	High	Medium	Low	Confidentiality	Integrity	Availability	Access
Mem	**1.82**	0.43	0.31	0.92	0.85	**1.29**	0.47
Con	0.60	0.92	**5.57**	**1.10**	**1.20**	0.80	0.00
Sem	0.69	**1.22**	**1.22**	**1.04**	**1.08**	0.85	**1.30**

The correlation metric *lift* is defined in Section 3.3. Categories with too few examples (fewer than 3) are not shown due to statistical insignificance

Table 13 Correlation between root causes and severity/impacts for security related bugs (BugSet2) in Apache

Cause	Severity			Impact		
	High	Medium	Low	Confidentiality	Integrity	Availability
Mem	**1.67**	0.90	0.00	0.46	0.00	**2.10**
Sem	0.77	**1.03**	**1.42**	**1.14**	**1.37**	0.64

The correlation metric *lift* is defined in Section 3.3. Categories with too few examples (fewer than 3) are not shown due to statistical insignificance

bugs reported in Bugzilla during the corresponding year (the bars). Similarly, Fig. 6b and c shows the number and percentage of vulnerabilities over time in Apache and the Linux kernel. The trend shows that *the numbers and the percentage of vulnerabilities are increasing for Mozilla, Apache, and the Linux kernel*. However, it does not necessarily indicate that software is less secure. Instead, it may demonstrate that people have paid more attention to security issues and security issues is becoming increasingly important for client, server, and OS software.

Despite the increasing percentage, the absolute number of Mozilla classified security bugs in the last two years decreases. This is understandable because it takes time for security bugs to be classified since this step is manual; therefore, recently reported security bugs are less likely to be classified, contributing to the relatively smaller number of classified security bugs in the last two years of Mozilla.

8.2 Semantic Bugs are the Dominant Cause of Security Vulnerabilities

Figure 7a shows the distribution of root causes and severity for security bugs. Surprisingly, memory bugs account for only 27.7–34.5 %, while semantic bugs cause 61.8–71.5 % of vulnerabilities. This finding is against the belief that buffer overflows are the most common form of security vulnerability (Cowan et al. 2000); and it agrees with a recent study (Neuhaus and Zimmermann 2010). The result suggests that while it is important to detect buffer overflows to reduce security vulnerabilities, we should also provide support for detecting, diagnosing, and fixing security vulnerabilities caused by semantic bugs.

On the other hand, Fig. 7a shows that compared to semantic bugs, memory bugs are more likely to cause more severe security vulnerabilities (High). This result has been supported by the correlation metric *lift* shown in Tables 13–14 as well. In addition, Tables 13–14 show that memory bugs are likely to cause unavailability. Since many memory bugs can be detected by the existing tools such as Purify,

Table 14 Correlation between root causes and severity/impacts for security related bugs (BugSet2) in the Linux kernel

Cause	Severity			Impact			
	High	Medium	Low	Confidentiality	Integrity	Availability	Access
Mem	**1.47**	0.98	0.21	0.72	**1.05**	**1.22**	NA
Con	0.43	**1.68**	NA	0.70	0.93	**1.31**	0.00
Sem	0.77	0.97	**1.47**	**1.21**	0.97	0.83	**1.34**

The correlation metric *lift* is defined in Section 3.3. Categories with too few examples (fewer than 3) are not shown due to statistical insignificance

Valgrind, and Coverity (2013), and developers indicated that they used these tools during the development process (Section 4), our results indicate that it is important to understand why these tools have not been used with their full capacity and address the relevant issues (Hafiz 2010), e.g., improving the detection capability of the current bug detection techniques, reducing false positives, simplifying the usage procedure, promoting these tools to more developers, etc.

The different distribution of impacts shown in Fig. 7b indicates that vulnerabilities have different impacts on client, server, and OS systems. For servers (Apache) and operating systems (the Linux kernel), unavailability is the most common type of vulnerabilities, which is not the case for client systems (Mozilla).

9 Concurrency Bugs

In addition to program inputs, the scheduling affects the manifestation of concurrency bugs. Therefore, it is often expected that concurrency bugs are hard-to-reproduce. Is this true for the reported bugs in real-world software such as Mozilla? In this section, we answer the following research questions: **RQ6a**: Are concurrency bugs hard-to-reproduce? **RQ6b**: Do these concurrency bugs cause severe impacts on software systems?

The bug set used for this section is **BugSet3**—concurrency bugs in Bugzilla databases retrieved using keyword searches (Section 2.2). Concurrency bugs only constitute a very small percentage of all reported bugs. Therefore, random sampling cannot provide enough concurrency bugs for a representative study. To collect enough concurrency bugs, we use keywords, i.e., “race”, “lock”, “deadlock”, “synchronization”, “starvation”, and “atomic”, to search bug reports and extract reports that contain these keywords as potential concurrency bugs. We then manually verify whether they are concurrency bugs. Using this method, we collect 90 concurrency bugs from Mozilla. Apache and the Linux kernel Bugzilla databases only contain 1.5K–4.7K fixed bug reports, which are two orders of magnitude smaller than that of Mozilla (Table 1). In addition, there is only a very small percentage of concurrency bugs as shown in Fig. 1a. Therefore, due to the small number of concurrency bugs in the Apache and Linux kernel Bugzilla databases, the results for Apache and Linux kernel concurrency bugs are not discussed in this paper.

9.1 Concurrency Bugs are Hard-to-Reproduce

We found that most Mozilla concurrency bugs could not be reproduced by developers with an acceptable probability (e.g., more than once out of ten times). Therefore, even though Mozilla still contains many concurrency bugs, failures caused by concurrency bugs were unlikely to be reported by users or to be fixed by developers because they were difficult to reproduce. Techniques to help reproduce concurrency bugs and increase the probability of manifesting concurrency bugs Park et al. (2009a, b) would be helpful for diagnosing concurrency bugs. In addition, it would be beneficial to encourage users and developers to include more information about how to reproduce concurrency bugs when they report such bugs.

9.2 Most Concurrency Bugs Lead to Hangs or Crashes

Figure 8 shows the distribution of impacts from Mozilla concurrency bugs. Compared to the general Mozilla bugs shown in Fig. 3, Mozilla concurrency bugs cause

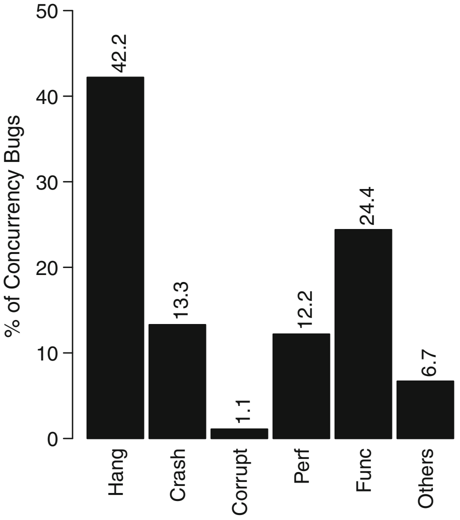


Fig. 8 Impact distribution of concurrency bugs in Mozilla (BugSet3)

much more severe impacts on software systems, which is consistent with previous work (Sullivan and Chillarege 1992). Specifically, 42.2 % of Mozilla concurrency bugs cause system hangs due to synchronization errors and deadlocks, 21 times higher than general Mozilla bugs. Further, 55.5 % of Mozilla concurrency bugs lead to crashes and hangs, which are likely to be detected and recovered by generic recovery techniques.

10 Automatic Bug Classification

In this section, we answer **RQ7**: Would it be possible to automatically find memory bugs and semantic bugs? Automatic classification can help reduce the manual effort in building bug benchmarks (Lu et al. 2005) and evaluating bug detection and diagnosis tools (Tang et al. 2008). In addition, automatic classification may help developers understand certain categories of bugs for better bug diagnosis and fixing. To automatically find a large number of bugs of a certain type, we leverage machine learning techniques to automatically classify a large number of bug reports into different bug types.

10.1 Approach

As fixed bug reports do not always describe valid runtime bugs as discussed earlier, we must first automatically identify fixed runtime bugs from the fixed bug reports in Bugzilla databases. Therefore, we employ a two-level classification approach. The first level classifier classifies whether a given fixed bug report is a fixed runtime bug. The second level classifiers classify fixed runtime bugs according to their root causes. We build one multi-class second level classifier for memory bugs and semantic bugs. Our method consists of the following steps: (1) preprocessing bug reports; (2) training; (3) evaluating classification performance; and (4) applying classification models on the entire Bugzilla databases.

- **Preprocessing.** In Bugzilla databases, a typical bug report contains the following information: bug ID, summary, reporting time, priority, bug severity, product, component, bug description, discussion comments, and reporter. We use all these fields except the bug ID for classification. We represent the summary, the bug description, and the discussion comments in the word level, called *bag-of-words* approach. Each word in bug documents is parsed into an index. Each bug document is represented by a vector.
- **Training.** We use the manually-labeled bug reports in our random sample (BugSet1) for automatic classification. To avoid evaluation bias caused by tuning, we *randomly* divide the whole sample into two *halves*: *training set* for learning and tuning, and *test set* for performance evaluation. As the test set does not affect the learning and tuning, and the test set is a random sample of the entire Bugzilla databases, the performance evaluated on the test set should be representative for the entire Bugzilla database. We experimented with four classification methods in Hall et al. (2009)—Support Vector Machines using Sequential Minimal Optimization SMO (2013), libSVM Chang and Lin (2001), and Bayes Net (2013), and J48 decision tree (Quinlan 1993). In addition, we explore the different parameter settings of these classification methods.

  We use 10-fold cross validation (Joachims 2002) on the training set to find out the best classification method with the best parameter setting based on the performance metrics described next. The 10-fold cross validation is a standard approach to mitigate the randomness in the data set. It randomly divides the data set into 10 equal parts, i.e., fold1, fold2, ..., and fold10, It builds 10 classifiers by learning from 9 folds at a time, i.e., using fold1–9 to build a classifier, using fold1–8 and fold10 to build another classifier, and so on and so forth. The performance is measured on the remaining fold, e.g., we use the classifier built from fold1–9 to classify fold10 and measure its performance. The performance from all 10 folds are combined to indicate the overall performance. We use this overall performance to evaluate the effectiveness of the different parameters and classification algorithms. Note that we use 10-fold cross validation on the *training set* to find the best parameters and classification algorithms to build the classifier; and then we use the classifier to classify bug reports in the test set to evaluate the performance of the classifier. This approach ensures that the test set does not affect the parameter tuning for a fair evaluation. We run SMO with all six kernels (-K). We apply libSVM with two different kernels (-k)—linear kernel (0) and RBF kernel (2). We run Bayes Net with the maximum number of parents (-P) from 1 to 3 in increment of 1. We run J48 decision tree with confidence factor (-c) from 0.1 to 0.3 in increment of 0.05, and the minimum number of instances per leaf (-m) from 1 to 3 in increment of 1. The detailed classification results are shown in Section 10.2.

  The Mozilla training set contains 318 fixed bug reports, 177 of which are fixed runtime bugs. In other words, for our first level classification, the training set contains 318 instance, 177 of which are positive instances (all of the fixed runtime bugs), and the rest are negative instances. Among the 177 fixed runtime bugs, 155 are semantic bugs, 19 are memory bugs, and three are concurrency bugs. Therefore, for the second-level memory bug classification, the training set contains 177 instance, with 19 memory bug instances, and 155 semantic bug instances. The concurrency bugs will received a negative label for the second level. The test set contain 317 bug reports, 162 of which are fixed runtime bugs.

Among these fixed runtime bugs, 21 are memory bugs, 140 are semantic bugs, and one is a concurrency bug. We apply multi-class classification to classify bugs into a memory bug, a semantic bug, or neither.

- **Evaluating Classification Performance.** To evaluate how good the classification models are, we measure the prediction performance. Four different types of prediction results are possible from a binary classifier: We use three metrics to evaluate the classification performance, which are *Precision* ($P = \frac{T_+}{T_+ + F_+}$), *Recall* ($R = \frac{T_+}{T_+ + F_-}$), and $F1$, which is an even combination of precision and recall ($F1 = \frac{2PR}{P+R}$). When precision and recall are equally important, $F1$ can be used. For bug classification, the goal is both high precision and high recall, so we use $F1$ as the tuning metric.

		Predicted label	
		Yes	No
Actual	Yes	True positive (T_+)	False negative (F_-)
Label	No	False positive (F_+)	True negative (T_-)

- **Applying Classification Model.** After we obtain classification models for each category, we apply them on the whole database to predict which categories a bug probably belongs to. The two level classification techniques described earlier are applied to automatically identify fixed runtime bugs, memory bugs, and semantic bugs. The detailed two-level classification results are shown in Section 10.2.

10.2 Automatic Classification Results

We presents our automatic classification results on Mozilla to show that it is feasible to automatically identify memory bugs and semantic bugs.

As described in Section 10.1, we apply four classification algorithms with different parameters on the training set to find classifiers with the highest $F1$. These classifiers are then applied on the Mozilla test set for performance evaluation. As the test set does not affect the selection of classification algorithms or parameter tuning, and the test set is a random sample of the entire bug database, the performance evaluated on the test set should be representative for the entire bug database. The first level classifier has a precision of 0.64, a recall of 0.79, and a F1 of 0.71 (Table 15) by Support Vector Machines using Sequential Minimal Optimization (SMO 2013). The kernel used (-K) is the normalized PolyKernel. The precision, recall, and F1 of the second-level classification on the test set by Bayes Net (2013) are shown in Table 15 as well. We use value 2 for the maximum number of parents (-P) parameter.

We then apply the classifiers on all 189,097 fixed bug reports in the whole Mozilla Bugzilla database; 109,014 are identified as fixed runtime bugs. Next, the multi-class classifier is applied on the 109,014 fixed runtime bugs to identify memory bugs

Table 15 Distribution of root causes based on automatic classification

	Percentage in Bugzilla	Classification on test set		
		Precision	Recall	F1
First-level: Bugs	57.6 %	0.64	0.79	0.71
Second-level: Memory	14.0 %	0.67	0.57	0.62
Second-level: Semantic	85.8 %	0.93	0.95	0.94

and semantic bugs. Table 15 column "Percentage" shows the distribution of bug root causes on the entire Mozilla Bugzilla database by our automatic classification. Figure 1a shows that in Mozilla's random sample, 11.8 % are memory bugs and 87.0 % are semantic bugs. Compared with these results using randomly sampled bug reports, the percentage of memory and semantic bugs from our automatic classification is similar, which indicates that the distribution based on sampled bugs and that based on a large data set are consistent for Mozilla. Similarly, Table 2 shows that 53.4 % (339/635) of the randomly sample bug reports in Mozilla are bugs. The percentage from our automatic classification is similar.

Although the performance of the automatic classification is reasonable, it is desirable to have higher precisions and recalls. The main challenge is to improve the performance of the first level classification, because it is hard for a classifier to tell whether a fixed bug report is a fixed runtime bug for the following main reason. Many types of bug reports are not considered fixed runtime bugs, e.g., new feature requests, compile-time errors, configuration errors, etc. These types of bug reports are difficult to be characterized in the bag-of-words level. It remains as our future work to further improve the first level classification.

11 Related Work

Much effort has been made to study fault related characteristics of large software systems (Aranda and Venolia 2009; Basili and Perricone 1984; Chillarege et al. 1991; Chou et al. 2001; Compton and Withrow 1990; Endres 1975; Fenton and Ohlsson 2000; Glass 1981; Gu et al. 2003; Kaâniche et al. 1994; Lu et al. 2008; Mockus et al. 2002; Moller and Paulish 1993; Ostrand and Weyuker 2002; Ostrand et al. 2005; Ostrand and Weyuker 1984; Pighin and Marzona 2003; Podgurski et al. 2003; Sahoo et al. 2010; Sullivan and Chillarege 1991, 1992; Yin et al. 2011). They show important results and have also identified some counter-intuitive findings. By analyzing the error type, defect type and error trigger distribution for shipped code of three IBM software systems, Sullivan and Chillarege (1992) found that memory bugs are a major type and have high impact. Ostrand and Weyuker (2002) found that the majority of post-release faults occurred in files that had no pre-release faults. This observation contradicts the conventional wisdom and suggests most testing effort for post-release software be put on previous fault-free or less-faulty parts instead of most faulty parts.

Lu et al. (2008) conducted a detailed empirical study of concurrency bugs. While they focused on the bug root causes, bug manifestation, bug fix strategies, and bug avoidance of concurrency bugs, this paper studies the percentage of concurrency bugs, and the correlation between bug root causes and bug impact of concurrency bugs. A recent study (Sahoo et al. 2010) on bug reports in server applications shows that about 77 % of the failures are caused by bugs that can be reproduced with just one input request. Yin et al. (2011) studied how bug fixes become bugs. Other work studies different aspects of bug fixes including examining bug fixes that are cross-referenced between FreeBSD and OpenBSD (Canfora et al. 2011), and identifying bug-fixing commits (Tian et al. 2008). German studied and visualized fine-grained modifications (Germán 2006). Herraiz et al. (2008) showed that the bug priority/importance field is not a good indicator of the bug resolution time. This paper answers different research questions from the previous studies.

Ozment and Schechter (2006) measure the rate of code addition and the rate of reported security vulnerabilities in OpenBSD operating system to determine whether its security increases over time. Massacci et al. (2011) study reported security vulnerabilities in six major versions of Mozilla Firefox. Neuhaus and Zimmermann (2010) study the trend of security vulnerabilities without distinguishing different software projects. Zaman et al. (2011) use Firefox as a case study to understand how security bugs and performance bugs are different from other types of bugs. For example, they find that security bugs are fixed faster, but are more likely to be reopened. In addition to the trend of security vulnerabilities over time, this paper studies the correlation between the root cause and the severity and the correlation between the root cause and the impact of security vulnerabilities. On the other hand, this paper studies the reported security vulnerabilities in different software, i.e., Apache and the entire Mozilla suite (Firefox is only one of many projects in the Mozilla suite).

Many studies apply text mining and machine learning techniques to identify security bug reports (Gegick et al. 2010) and duplicate bug reports (Runeson et al. 2007; Sun et al. 2010; Wang et al. 2008). Many other studies predict the most appropriate developers for fixing a bug report (Anvik et al. 2006; Cubranic and Murphy 2004; Matter et al. 2009; Pamela and Iulian 2010; woo Park et al. 2011), the bug lifetime and bug-fix time (Bougie et al. 2010; Hooimeijer and Weimer 2007; Pamela and Iulian 2011; Panjer 2007), which bugs will be fixed (Guo et al. 2010) or re-opened (Emad et al. 2010), the quality of bug reports (Philipp et al. 2008), the severity of bug reports (Ahmed et al. 2011), source files that need to be changed (Syed and Franz 2010; Ying et al. 2004), and what files, classes, and modules are more fault-prone (D'Ambros et al. 2012; Graves et al. 2000; Guo et al. 2004; Ostrand et al. 2005; Kpodjedo et al. 2011; Shin et al. 2012; Weyuker et al. 2010). Podgurski et al cluster software failures based on automatically recorded function call profiles (Podgurski et al. 2003). Our classifiers address a different classification problem of identifying memory and semantic bugs automatically.

Our previous work (Li et al. 2006) studied the bug characteristics of two open source projects Mozilla and Apache. This paper makes several new contributions. First, we randomly sampled and studied additional 300 bug reports from another piece of software, the widely-used Linux kernel operating system, compared and contrasted our findings in OS and non-OS software (Sections 4–7). We also studied the trend of the Linux kernel bugs (Figs. 2b and d in Section 4), and the security bugs in the Linux kernel (Section 8). Many findings are new, e.g., regarding interrupt-related bugs, driver bugs, and the strong correlation between null pointer dereferences and bugs in file systems. Second, we updated our study with recent data and added new data. Specifically, we randomly sampled and studied additional 300 bug reports from the year 2005 to 2010 in Mozilla and Apache Bugzilla databases, while our previous work only sampled bug reports until 2005. We discuss how we combine the two data sets to maintain the randomness of sampling in Appendix A. The security vulnerability data is updated as well: We analyzed all 1,387 classified security vulnerabilities in NVD for Mozilla, Apache, and the Linux kernel until December 31st, 2011. Almost all figures and tables are either new or regenerated. Our results in this paper combined with the results in our previous paper (Li et al. 2006) show that the distributions of the data sets sampled by 2005 and 2010 are similar. This indicates that the variance in different data sets is small, which increases the confidence and reproducibility of our results. Third, we used a new multi-class classification to build classifiers on the new combined sample and applied the new classifiers on the entire Mozilla Bugzilla database. We now use the standard 10-fold cross-validation instead of 5-fold cross-validation. Fourth, we extended the related work and threats to validity sections, and added more details about how we collect bug samples, how we classify bug reports, etc. Lastly, we have added three appendices: Appendix A describes how we combine the two data sets to maintain the pure randomness of sampling. Appendix B presents bug examples and their classification. Appendix C gives the detailed numbers for the bar graphs whose bars show the break down of different categories.

12 Conclusions and Future Work

This paper studies the bug characteristics in three large open-source software projects to guide the design of effective tools for detecting and recovering from software failures. We manually study 2,060 randomly-sampled real world bugs in three dimensions—root causes, impacts, and components. We further study the correlation between categories in different dimensions, and the trend of different types of bugs. Our findings and their implications include: (1) semantic bugs are the dominant root cause. As software evolves, semantic bugs increase, while memory-related bugs decrease, calling for more research effort to address semantic bugs; (2) the Linux kernel operating system (OS) has more concurrency bugs than its non-OS counterparts, suggesting more effort into detecting concurrency bugs in operating system code; and (3) reported security bugs are increasing, and the majority of them are caused by semantic bugs, suggesting more support to help developers diagnose and fix security bugs, especially semantic security bugs.

To reduce the manual effort in building bug benchmarks for evaluating bug detection and diagnosis tools, we use machine learning techniques to automatically classify hundreds of thousands of bugs. In the future, we would like to study bug characteristics of software written in other languages such as Java so that we can learn the language impact on bug characteristics.

Acknowledgements We thank Luyang Wang and Yaoqiang Li for classifying some bug reports. We thank Shan Lu for the early discussion and feedback. The work is partially supported by the National Science and Engineering Research Council of Canada, the United States National Science Foundation, the United States Department of Energy, a Google gift grant, and an Intel gift grant.

Appendices

A Combining Two Data Sets

To leverage the randomly sampled bug reports studied in our prior work in 2005 (Li et al. 2006), each of the two bug report samples from Mozilla and Apache Bugzilla databases is combined from two random samples. The combination is performed in the following way to maintain the pure randomness of sampling. The goal is to ensure that the combined set of fixed bug reports is no different from a random sample of fixed bug reports on the entire Bugzilla databases now.

Figure 9 illustrates the combination approach. We randomly sampled $2X$ % of fixed bug reports in one Bugzilla database by the cutoff date of our prior work, referred to as $Date1$. Now we randomly select half of the $2X$ % of fixed bug reports, referred to as $Set1$; the other half is discarded. Note that $Set1$ is a random sample of X% of bug reports fixed by $Date1$. On our new sampling date (Table 1), denoted

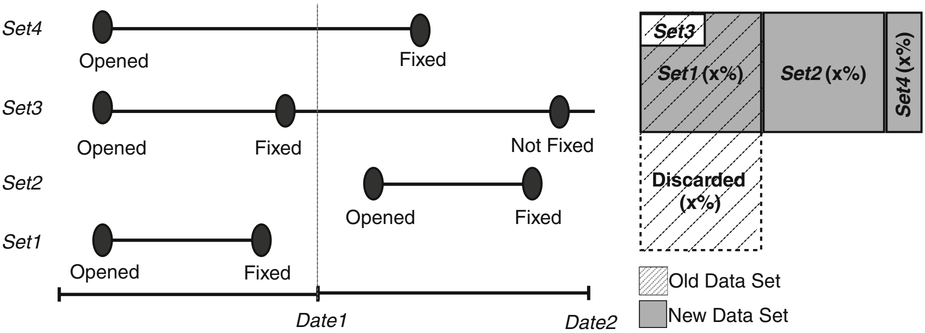


Fig. 9 Combining Two Data Sets. *Not Fixed* denotes any status other than *Fixed*, e.g., reopened, invalid, etc. *Old Data Set* is the data set used in our previous work Li et al. (2006) and *New Data Set* is the data set used in this paper

as *Date*2, we sample another X % of the fixed bug reports that were opened after *Date*1 and before *Date*2, denoted as *Set*2. We keep only half of the bug reports fixed by *Date*1 so that the sampled bug reports before *Date*1 and sampled bug reports after *Date*1 are in proportion to the bug reports belong to the two time ranges.

The status of a bug report may have changed since *Date*1 in the following two ways: (1) a fixed bug report by *Date*1 is no longer fixed by *Date*2; or (2) a unfixed bug report by *Date*1 is fixed by *Date*2. To compensate for these two scenarios, we identify all the fixed bug reports in *Set*1 that are no longer marked as fixed on *Date*2, denoted as *Set*3. Bug reports in *Set*3 should not be included in our sample, because if we take a random sample of fixed bug reports on *Date*2, those bug reports would not be sampled as they are not fixed. From the bug reports that are unfixed by *Date*1 but are fixed by *Date*2, we randomly sample X % of them, denoted as *Set*4. Our final random sample is the union of *Set*1, *Set*2, and *Set*4 with *Set*3 excluded.

Table 16 lists sizes of the four data sets for the three software projects. No combining is needed for the Linux kernel since it was sampled in 2010.

There is no difference between our combined sample and a sample randomly picked from the fixed bug reports on *Date*2, as either is a random sample of X% on the population. Therefore, the combined sample is representative of fixed bug reports in the Bugzilla database. In addition, our results show the distributions of these sets are similar, meaning that the variance in different data sets is small, which increases the confidence and reproducibility of our results.

Developer may update bug reports after the bugs are fixed. Therefore, we check all bugs in *Set*1 to find out whether the later activities affect our classifications of the bugs. Fortunately, only 5 of those bug reports in Mozilla and none in Apache have activities after *Date*1. We manually read these 5 bug reports again; and find that those activities change the product, the QA contact, or the component of the bug reports, and do not change our original classifications. The component field

Table 16 Number of bug reports in *Set*1–4

Software	Set1	Set2	Set3	Set4
Mozilla	274	336	0	25
Apache	101	82	1	18
Linux	300			

used in the bug reports is finer-grained than the definition of component in Table 3. Therefore, the finer-grained component change in bug reports does not affect the higher-level component used in this paper.

B Bug Examples

Table 17 Bug examples for each category

Dimension	Category	Software	Bug ID	Relevant title & description
Root cause	Memory	Linux	11364	Memory Leak: ... The 'uccf' variable is not deallocated before 'return—ENOMEM' is called
	Concurrency	Apache	8124	mod_ssl fails to get and release semaphore mutex
	Semantic	Mozilla	267365	wrong homonym ... Actual Results: Text reads "Over-written ...." Expected Results: Text should read "Overridden ..."
Impact	Hang	Apache	29901	If the size of the file test.html reaches 65321 bytes, apache hangs and the page is never returned.
	Crash	Linux	12591	my box crashes during startup when hddtemp tries to start.
	Data corruption	Mozilla	327907	Ending process firefox.exe can lead to database corruption
	Performance degradation	Linux	6417	it results in ... severe slowdowns.
	Incorrect functionality	Mozilla	4593	style changes (bold,italic,underline) aren't transparent thru highlighted selection
Software component (Mozilla & Apache)	Core	Mozilla	169296	race condition in PK11SDR_Encrypt
	GUI	Mozilla	4593	style changes (bold,italic,underline) aren't transparent thru high-lighted selection
	Network	Apache	37911	The fix is: (in Secure Sockets layer (SSL))—ssl_engine_init.c
	I/O	Apache	29964	I'm seeing a non-terminating loop in ssl_io_input_getline().
OS component (Linux)	Drivers	Linux	715	Product: SCSI Drivers
	Core	Linux	8476	kernel BUG at include/linux/slub_def.h
	Network	Linux	780	Timing related bug in the RPC client code
	File system	Linux	6831	after io_getevents reports that write/appen was done, the data in file is still unaccessible
	Architecture	Linux	8870	Platform Specific/Hardware ... Problem disappears with i386-kernel or if noapic or acpi=off kernel option is used.

C Detailed Numbers for the Figures

Table 18 Distribution of root causes with impacts (BugSet1)

Software	Category	Hang	Crash	Corrupt	Perf	Func	Others	Unknown
Mozilla	Mem	0.00	6.04	0.00	0.28	1.43	2.01	2.01
	Con	0.59	0.29	0.00	0.00	0.00	0.00	0.29
	Sem	1.43	9.19	1.43	0.86	66.91	4.59	2.58
Apache	Mem	0.00	8.24	0.00	2.06	1.03	1.03	0.00
	Con	0.00	1.03	0.00	0.00	3.09	1.03	0.00
	Sem	3.09	2.06	1.03	9.27	61.85	1.03	4.12
Linux	Mem	1.81	7.25	1.36	2.26	2.72	0.00	0.90
	Con	5.44	2.17	1.63	1.63	2.72	0.00	0.00
	Sem	10.73	12.62	1.26	5.04	38.50	0.63	1.26

The table shows data for Fig. 1a

Table 19 Distribution of root causes with impacts (BugSet1)

Software	Category	Hang	Crash	Corrupt	Perf	Func	Others	Unknown
Mozilla	MLK	0.00	2.50	0.00	0.00	10.00	12.50	15.00
	UMR	0.00	7.50	0.00	0.00	0.00	0.00	2.50
	Danling	0.00	5.00	0.00	0.00	0.00	0.00	0.00
	NULL	0.00	25.38	0.00	0.00	2.30	2.30	0.00
	Overflow	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Others	0.00	10.00	0.00	2.50	0.00	2.50	0.00
Apache	MLK	0.00	0.00	0.00	8.33	0.00	8.33	0.00
	UMR	0.00	25.00	0.00	0.00	0.00	0.00	0.00
	Danling	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	NULL	0.00	33.33	0.00	0.00	0.00	0.00	0.00
	Overflow	0.00	8.33	0.00	8.33	8.33	0.00	0.00
	Others	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Linux	MLK	6.25	8.33	0.00	10.41	0.00	0.00	0.00
	UMR	0.00	0.00	0.00	0.00	0.00	0.00	4.16
	Danling	0.00	4.16	0.00	0.00	0.00	0.00	0.00
	NULL	0.00	20.83	0.00	0.00	0.00	0.00	0.00
	Overflow	1.89	7.57	5.68	0.00	5.68	0.00	0.00
	Others	0.00	8.33	0.00	0.00	12.50	0.00	4.16

The table shows data for Fig. 1b

Table 20 Distribution of root causes with impacts (BugSet1)

Software	Category	Hang	Crash	Corrupt	Perf	Func	Others	Unknown
Mozilla	MissF	0.00	0.33	0.00	0.00	8.81	0.00	0.33
	MissC	0.30	0.61	0.00	0.00	6.19	0.00	0.00
	Corner	0.00	2.37	0.33	0.00	6.10	0.33	0.00
	CtrlFlow	0.32	0.32	0.00	0.32	5.48	0.32	0.00
	Except	0.67	1.69	0.00	0.00	1.35	0.00	0.00
	Process	0.00	0.30	0.30	0.00	2.77	0.00	0.00
	Typo	0.00	1.30	0.00	0.00	4.90	1.96	0.98
	FuncImpl	0.32	3.61	0.98	0.32	36.14	0.00	1.31
	Others	0.00	0.00	0.00	0.35	4.59	2.82	0.35

Table 20 (continued)

Software	Category	Hang	Crash	Corrupt	Perf	Func	Others	Unknown
Apache	MissF	0.00	0.00	0.00	0.00	7.50	0.00	0.00
	MissC	1.25	2.50	0.00	3.75	11.25	0.00	0.00
	Corner	1.25	0.00	0.00	0.00	6.25	0.00	0.00
	CtrlFlow	1.25	0.00	0.00	0.00	7.50	0.00	0.00
	Except	0.00	0.00	0.00	0.00	1.25	0.00	0.00
	Process	0.00	0.00	0.00	0.00	3.75	0.00	1.25
	Typo	0.00	0.00	0.00	2.50	11.25	1.25	0.00
	FuncImpl	0.00	0.00	1.25	5.00	26.25	0.00	3.75
	Others	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Linux	MissF	0.97	0.00	0.00	0.00	2.91	0.00	0.00
	MissC	3.72	5.59	1.86	0.93	10.25	0.00	0.93
	Corner	0.83	1.66	0.00	0.83	2.49	0.00	0.00
	CtrlFlow	0.00	0.97	0.00	0.00	2.91	0.00	0.00
	Except	0.97	0.97	0.00	0.97	0.00	0.00	0.00
	Process	0.00	0.00	0.00	0.00	3.88	0.00	0.00
	Typo	0.00	3.88	0.97	0.00	1.94	0.00	0.97
	FuncImpl	7.64	3.39	0.00	4.24	24.63	0.84	0.00
	Others	0.86	1.72	0.00	0.00	5.17	0.00	0.00

The table shows data for Fig. 1c

Table 21 Distribution of bug impacts (BugSet1)

Software	Category	Memory	Concurrency	Semantic
Mozilla	Hang	0.00	0.59	1.47
	Crash	6.19	0.29	9.43
	Corrupt	0.00	0.00	1.47
	Perf	0.29	0.00	0.88
	Func	1.47	0.00	68.73
	Others	2.06	0.00	4.71
	Unknown	2.06	0.29	2.65
Apache	Hang	0.00	0.00	3.09
	Crash	8.24	1.03	2.06
	Corrupt	0.00	0.00	1.03
	Perf	2.06	0.00	9.27
	Func	1.03	3.09	61.85
	Others	1.03	1.03	1.03
	Unknown	0.00	0.00	4.12
Linux	Hang	2.72	6.80	11.56
	Crash	10.88	2.72	13.60
	Corrupt	2.04	2.04	1.36
	Perf	3.40	2.04	5.44
	Func	4.08	3.40	41.49
	Others	0.00	0.00	0.68
	Unknown	1.36	0.00	1.36

The table shows data for Fig. 3

Table 22 Distribution of Mozilla and Apache bugs in software components (BugSet1)

Software	Category	Hang	Crash	Corrupt	Perf	Func	Others	Unknown
Mozilla	Core	0.23	7.32	0.94	1.41	17.25	3.07	3.07
	GUI	0.70	4.72	0.23	0.70	41.13	1.41	1.18
	Network	0.47	1.65	0.00	0.47	4.01	0.47	0.23
	I/O	0.23	1.18	0.23	0.00	2.83	0.00	0.23
	Others	0.00	0.47	0.00	0.00	2.12	1.89	0.00
Apache	Core	3.80	11.42	2.85	5.71	37.14	1.90	2.85
	GUI	0.00	0.00	0.00	0.95	2.85	0.00	0.00
	Network	0.00	0.00	0.00	1.90	20.00	0.00	0.00
	I/O	0.95	0.95	0.00	1.90	1.90	0.00	0.95
	Others	0.00	0.00	0.00	0.00	0.95	0.95	0.00

This table shows data for Fig. 4

Table 23 Distribution of Linux bugs in software components (BugSet1)

Software	Category	Hang	Crash	Corrupt	Perf	Func	Others	Unknown
Linux	Core	0.00	2.32	1.16	0.58	2.90	0.00	0.00
	Driver	11.04	12.20	0.58	5.81	21.51	0.00	1.74
	Network	4.06	1.16	0.00	0.58	4.06	0.00	0.00
	FS	0.58	3.48	1.74	0.00	2.90	0.00	0.00
	Arch	2.32	2.32	0.58	1.74	8.13	0.58	0.58
	Others	0.00	1.74	0.58	0.58	2.32	0.00	0.00

This table shows data for Fig. 5

Table 24 Distributions of causes and impacts of security related bugs in NVD (BugSet2)

Software	Category	High	Medium	Low
Mozilla	Mem	20.89	6.44	0.39
	Con	0.19	0.39	0.19
	Sem	20.31	47.26	3.90
Apache	Mem	11.11	16.66	0.00
	Con	0.00	1.85	0.00
	Sem	12.96	48.14	9.25
Linux	Mem	17.79	15.16	1.49
	Con	0.56	2.80	0.37
	Sem	16.66	26.77	18.35

This table shows data for Fig. 7a

Table 25 Distributions of causes and impacts of security related bugs in NVD (BugSet2)

Software	Category	Mem	Con	Sem
Mozilla	Confidentiality	11.08	0.29	22.87
	Integrity	9.39	0.29	21.67
	Availability	14.18	0.19	16.98
	Access	0.49	0.00	2.49
Apache	Confidentiality	2.81	1.40	21.12
	Integrity	0.00	0.00	30.98
	Availability	21.12	1.40	19.71
	Access	0.00	0.00	1.40
Linux	Confidentiality	9.84	0.99	25.00
	Integrity	8.40	0.77	11.72
	Availability	19.69	2.21	20.35
	Access	0.22	0.00	0.77

This table shows data for Fig. 7b

References

ASF bugzilla (2010) http://issues.apache.org/bugzilla. Accessed September 2010

Bayes Net (2013) http://www.cs.waikato.ac.nz/~remco/weka_bn/

Coverity (2013) Automated error prevention and source code analysis. http://www.coverity.com

Debugging Memory Leaks (2013) https://wiki.mozilla.org/Performance:Leak_Tools

Kernel Bug Tracker (2010) http://bugzilla.kernel.org/. Accessed February 2010

Mozilla.org Bugzilla (2010) https://bugzilla.mozilla.org. Accessed September 2010

National Vulnerability Database (2011) http://nvd.nist.gov. Accessed 31 December 2011

NVD Common Vulnerability Scoring System (2013) http://nvd.nist.gov/cvss.cfm?version=2

Support vector machines using sequential minimal optimization (2013) http://weka.sourceforge.net/doc.dev/weka/classif\iers/functions/SMO.html

Valgrind (2013) http://www.valgrind.org/

Ahmed L, Serge D, Quinten S, Tim V (2011) Comparing mining algorithms for predicting the severity of a reported bug. In: Proceedings of the 15th European conference on software maintenance and reengineering, pp 249–258

Amir M, Tao X (2005) Helping users avoid bugs in gui applications. In: Proceedings of the 27th international conference on software engineering, pp 107–116

Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: Proceedings of the 28th international conference on software engineering, pp 361–370

Aranda J, Venolia G (2009) The secret life of bugs: Going past the errors and omissions in software repositories. In: Proceedings of the 31st international conference on software engineering. IEEE, pp 298–308

Avizienis A, Laprie JC, Randell B, Landwehr CE (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secur Comput 1(1):11–33

Basili VR, Perricone BT (1984) Software errors and complexity: an empirical investigation. Commun. ACM 27(1):42–52

Beizer B (1990) Software testing techniques, 2nd edn. Van Nostrand Reinhold Co., New York, NY, USA

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. ESEC/FSE '09, pp 121–130

Bougie G, Treude C, German DM, Storey MA (2010) A comparative exploration of FreeBSD bug lifetimes. In: 2010 7th IEEE working conference on mining software repositories (MSR 2010). IEEE, pp 106–109

Canfora G, Cerulo L, Cimitile M, Di Penta M (2011) Social interactions around cross-system bug fixings: the case of freebsd and openbsd. In: Proceedings of the 8th working conference on mining software repositories, pp 143–152

Chang CC, Lin CJ (2001) Libsvm—a library for support vector machines. The Weka classifier works with version 2.82 of LIBSVM. http://www.csie.ntu.edu.tw/~cjlin/libsvm/

Chillarege R, Kao WL, Condit RG (1991) Defect type and its impact on the growth curve. In: Proceedings of the 13th international conference on software engineering, pp 246–255

Chou A, Yang J, Chelf B, Hallem S, Engler DR (2001) An empirical study of operating system errors. In: Proceedings of the 18th ACM symposium on operating systems principles, pp 73–88

Compton BT, Withrow C (1990) Prediction and control of ADA software defects. J Syst Softw 12(3):199–207

Cowan C (2003) Software security for open-source systems. IEEE Secur Priv 1(1):38–45

Cowan C, Wagle P, Pu C, Beattie S, Walpole J (2000) Buffer overflows: attacks and defenses for the vulnerability of the decade. In: Proceedings of the DARPA information survivability conference and exposition

Cubranic D, Murphy GC (2004) Automatic bug triage using text categorization. In: Proceedings of the 16th international conference on software engineering and knowledge engineering, pp 92–97

Dallmeier V, Zimmermann T (2007) Extraction of bug localization benchmarks from history. In: Proceedings of the 22nd IEEE/ACM international conference on automated software engineering, pp 433–436

D'Ambros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir Softw Eng 17(4–5):531–577

Edwards A, Tucker S, Demsky B (2010) Afid: an automated approach to collecting software faults. Autom Softw Eng 17(3):347–372

Emad S, Akinori I, Yasutaka K, Walid MI, Masao O, Bram A, Ahmed EH, Ken-ichi M (2010) Predicting re-opened bugs: a case study on the eclipse project. In: Proceedings of the 2010 17th working conference on reverse engineering, pp 249–258

Endres A (1975) An analysis of errors and their causes in system programs. In: Proceedings of the international conference on reliable software, pp 327–336

Engler D, Chen DY, Chou A (2001) Bugs as deviant behavior: a general approach to inferring errors in systems code. In: Proceedings of the 18th ACM symposium on operating systems principles, pp 57–72

Ernst MD, Cockrell J, Griswold WG, Notkin D (2001) Dynamically discovering likely program invariants to support program evolution. IEEE Trans Softw Eng 27(2):99–123

Fenton NE, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Trans Softw Eng 26(8):797–814

Gegick M, Rotella P, Xie T (2010) Identifying security bug reports via text mining: an industrial case study. In: Proceedings of the 7th working conference on mining software repositories, pp 11–20

Germán DM (2006) An empirical study of fine-grained software modifications. Empir Softw Eng 11(3):369–393

Glass R (1981) Persistent software errors. IEEE Trans Softw Eng 7(2):162–168

Graves T, Karr A, Marron J, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661

Gu W, Kalbarczyk Z, Iyer RK, Yang ZY (2003) Characterization of linux kernel behavior under errors. In: Proceedings of the 2003 international conference on dependable systems and networks, pp 459–468

Guo L, Ma Y, Cukic B, Singh H (2004) Robust prediction of fault-proneness by random forests. In: 15th international symposium on software reliability engineering, pp 417–428

Guo PJ, Zimmermann T, Nagappan N, Murphy B (2010) Characterizing and predicting which bugs get fixed. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, ICSE ’10, vol 1, pp 495–504

Hafiz M (2010) Security on demand. Ph.D. thesis

Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The WEKA data mining software: an update. SIGKDD Explorations, vol 11, issue 1

Han J, Kamber M (2001) Data mining: concepts and techniques. Morgan Kaufmann Publishers

Hastings R, Joyce B (1992) Purify: fast detection of memory leaks and access errors. In: Proceedings of the winter USENIX conference, pp 125–136

Herraiz I, German DM, Gonzalez-Barahona JM, Robles G (2008) Towards a simplification of the bug report form in eclipse. In: Proceedings of the 2008 international workshop on mining software repositories, MSR ’08. ACM Press, pp 145–148

Hooimeijer P, Weimer W (2007) Modeling bug report quality. In: Proceedings of the 22nd IEEE/ACM international conference on automated software engineering, pp 34–43

Joachims T (2002) Learning to classify text using support vector machines. Kluwer Academic Publishers

Kaâniche M, Kanoun K, Cukier M, de Bastos Martini MR (1994) Software reliability analysis of three successive generations of a switching system. In: Proceedings of the 1st European dependable computing conference on dependable compproceedings of the first European dependable computing conference on dependable computing, pp 473–490

Kim S, Zimmermann T, Pan K, Whitehead E Jr (2006) Automatic identification of bug-introducing changes. In: 21st IEEE/ACM international conference on automated software engineering (ASE’06), pp 81–90

Kpodjedo S, Ricca F, Galinier P, Guéhéneuc YG, Antoniol G (2011) Design evolution metrics for defect prediction in object oriented systems. Empir Softw Eng 16(1):141–175

Li Z, Lu S, Myagmar S, Zhou Y (2004) CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In: 6th symposium on operating systems design and implementation, pp 289–302

Li Z, Tan L, Wang X, Lu S, Zhou Y, Zhai C (2006) Have things changed now? An empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st workshop on architectural and system support for improving software dependability, ASID ’06

Lu S (2008) Understanding, detecting and exposing concurrency bugs. Ph.D. thesis

Lu S, Li Z, Qin F, Tan L, Zhou P, Zhou Y (2005) BugBench: a benchmark for evaluating bug detection tools. In: Workshop on the evaluation of software defect detection tools

Lu S, Park S, Seo E, Zhou Y (2008) Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th international conference on architectural support for programming languages and operating systems, pp 329–339

Massacci F, Neuhaus S, Nguyen VH (2011) After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In: Proceedings of the 3rd international conference on engineering secure software and systems, pp 195–208

Matter D, Kuhn A, Nierstrasz O (2009) Assigning bug reports using a vocabulary-based expertise model of developers. In: Proceedings of the 2009 6th IEEE international working conference on mining Software repositories, pp 131–140

Memon AM (2002) GUI testing: pitfalls and process. Computer 35(8):87–88

Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: Apache and Mozilla. ACM Trans Softw Eng Methodol (TOSEM) 11(3):309–346

Moller KH, Paulish DJ (1993) An empirical investigation of software fault distribution. In: Proceedings of the 1st international software metrics symposium, pp 82–90

Neuhaus S, Zimmermann T (2010) Security trend analysis with cve topic models. In: Proceedings of the 2010 IEEE 21st international symposium on software reliability engineering, pp 111–120

Ostrand T, Weyuker E (2002) The distribution of faults in a large industrial software system. In: Proceedings of the 2002 ACM SIGSOFT international symposium on software testing and analysis, pp 55–64

Ostrand TJ, Weyuker EJ (1984) Collecting and categorizing software error data in an industrial environment. J Syst Softw 4(4):289–300

Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 31(4):340–355

Ozment A, Schechter SE (2006) Milk or wine: does software security improve with age? In: Proceedings of the 15th conference on USENIX security symposium, vol 15

Pamela B, Iulian N (2010) Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In: Proceedings of the 2010 IEEE international conference on Software maintenance

Pamela B, Iulian N (2011) Bug-fix time prediction models: can we do better? In: Proceedings of the 8th workin conference on mining software repositories

Panjer LD (2007) Predicting Eclipse bug lifetimes. In: Proceedings of the Fourth International Workshop on mining software repositories, pp 29–32

woo Park J, woong Lee M, Kim J, won Hwang S, Kim S (2011) CosTriage: a cost-aware triage algorithm for bug reporting systems. In: In Proceedings of 25th conference on artificial intelligence

Park S, Lu S, Zhou Y (2009a) CTrigger: exposing atomicity violation bugs from their hiding places. In: Proceedings of the 14th international conference on architectural support for programming languages and operating systems, pp 25–36

Park S, Zhou Y, Xiong W, Yin Z, Kaushik R, Lee KH, Lu S (2009b) PRES: probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles, pp 177–192

Payne C (2002) On the security of open source software. Inf Syst J 12(1):61–78

Philipp S, Juergen R, Philippe C (2008) Mining bug repositories—a quality assessment. In: Proceedings of the 2008 international conference on computational intelligence for modelling control and automation

Pighin M, Marzona A (2003) An empirical analysis of fault persistence through software releases. In: Proceedings of the international symposium on empirical software engineering, p 206

Podgurski A, Leon D, Francis P, Masri W, Minch M, Sun J, Wang B (2003) Automated support for classifying software failure reports. In: Proceedings of the 23th international conference on software engineering, pp 465–475

Quinlan R (1993) C4.5: programs for machine learning. Morgan Kaufmann Publishers, San Mateo, CA

Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: Proceedings of the 29th international conference on software engineering, pp 499–510

Sahoo SK, Criswell J, Adve V (2010) An empirical study of reported bugs in server software with implications for automated bug diagnosis. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering—ICSE ’10, vol 1. ACM Press, p 485

Shin Y, Bell RM, Ostrand TJ, Weyuker EJ (2012) On the use of calling structure information to improve fault prediction. Empir Softw Eng 17(4–5):390–423

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: MSR ’05: Proceedings of the 2005 international workshop on mining software repositories, pp 1–5

Sullivan M, Chillarege R (1991) Software defects and their impact on system availability—a study of field failures in operating systems. In: 21st int. symp. on fault-tolerant computing, pp 2–9

Sullivan M, Chillarege R (1992) A comparison of software defects in database management systems and operating systems. In: 22nd annual international symposium on fault-tolerant computing, pp 475–484

Sun C, Lo D, Wang X, Jiang J, Khoo SC (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, pp 45–54

Swift MM, Bershad BN, Levy HM (2003) Improving the reliability of commodity operating systems. In: Proceedings of the ACM SIGOPS symposium on operating systems principles, pp 207–222

Syed A, Franz W (2010) Impact analysis of scrs using single and multi-label machine learning classification. In: Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement

Tan L, Yuan D, Krishna G, Zhou Y (2007) /* iComment: bugs or bad comments? */. In: Proceedings of the 21st ACM symposium on operating systems principles

Tan L, Zhou Y, Padioleau Y (2011) aComment: mining annotations from comments and code to detect interrupt-related concurrency bugs. In: Proceedings of the 33rd international conference on software engineering

Tang Y, Tang Y, Gao Q, Gao Q, Qin F, Qin F (2008) LeakSurvivor: towards safely tolerating memory leaks for garbage-collected languages. In: USENIX 2008 annual technical conference on annual technical conference, pp 307–320

Tian Y, Lawall J, Lo D (2008) Identifying linux bug fixing patches. In: Proceedings of the international conference on software engineering

Wang X, Zhang L, Xie T, Anvik J, Sun J (2008) An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the 30th international conference on software engineering, pp 461–470

Weyuker EJ, Ostrand TJ, Bell RM (2010) Comparing the effectiveness of several modeling methods for fault prediction. Empir Softw Eng 15(3):277–295

Wu R, Zhang H, Kim S, Cheung SC (2011) 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, pp 15–25

Yin Z, Yuan D, Zhou Y, Pasupathy S, Bairavasundaram L (2011) How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering, pp 26–36

Ying ATT, Murphy GC, Ng RT, Chu-Carroll M (2004) Predicting source code changes by mining change history. IEEE Trans Softw Eng 30(9):574–586

Zaman S, Adams B, Hassan AE (2011) Security versus performance bugs: a case study on firefox. In: Proceedings of the 8th working conference on mining software repositories, pp 93–102