Have Things Changed Now?

– An Empirical Study of Bug Characteristics in Modern Open Source Software

Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou and Chengxiang Zhai
Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801
{zli4, lintan2, xwang20, shanlu, yyzhou, czhai}@cs.uiuc.edu

ABSTRACT

Software errors are a major cause for system failures. To effectively design tools and support for detecting and recovering from software failures requires a deep understanding of bug[1] characteristics. Recently, software and its development process have significantly changed in many ways, including more help from bug detection tools, shift towards multi-threading architecture, the open-source development paradigm and increasing concerns about security and user-friendly interface. Therefore, results from previous studies may not be applicable to present software. Furthermore, many new aspects such as security, concurrency and open-source-related characteristics have not well studied. Additionally, previous studies were based on a small number of bugs, which may lead to non-representative results.

To investigate the impacts of the new factors on software errors, we analyze bug characteristics by first sampling hundreds of real world bugs in two large, representative open-source projects. To validate the representativeness of our results, we use *natural language text classification* techniques and automatically analyze around ***29,000*** bugs from the Bugzilla databases of the software.

Our study has discovered several new interesting characteristics: (1) memory-related bugs have decreased because quite a few effective detection tools became available recently; (2) surprisingly, some simple memory-related bugs such as NULL pointer dereferences that should have been detected by existing tools in development are still a major component, which indicates that the tools have not been used with their full capacity; (3) semantic bugs are the dominant root causes, as they are application specific and difficult to fix, which suggests that more efforts should be put into detecting and fixing them; (4) security bugs are increasing, and the majority of them cause severe impacts.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Reliability, Security

Keywords

Bug characteristics, empirical study, security, open source, bug detection

[1]We use "bug" and "error" interchangeably.

ASID'06 October 21, 2006, San Jose, California, USA.

1. INTRODUCTION

1.1 Motivation

Software failures greatly reduce system dependability. As software becomes more and more complex, there is an urgent need to explore more effective software testing and debugging tools and software engineering methods to minimize the number of bugs that escape into production runs. Since some bugs still slip through even the strictest testing, system designers also need to provide fault tolerant mechanisms to recover from these inevitable software failures.

To design effective tools for improving software quality requires a good understanding of software error characteristics in representative software. Such characteristics include bug distribution, impact, resolution time, and correlations. For example, if many bugs are caused by simple typos or copy-pastes [23], software development tools can provide more support to help detect these automatically. In testing, developers can focus on bugs based on the severity of the impact so that resource can be utilized more effectively [34].

Many empirical studies, including a few classic ones [9, 14, 17, 28, 35, 36], have been performed ten years ago to understand the characteristics of software errors. For example, several researchers (e.g. [14, 17]) have studied software errors occurring during software development, testing and validation phases. Sullivan and Chillarege [35, 36] analyzed error type, defect type and error trigger distribution for shipped code of two IBM database management products and one IBM operating system. 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. These studies provide useful insights and guidelines for software engineering tool designers and reliable system builders.

Over the last ten years, however, many factors in software development have significantly changed. As most previous studies were conducted using old software that was developed under the old development environment (e.g. without modern debugging tools) with traditional development paradigms and software architectures, it is unclear whether their results still apply. In addition, rising issues such as security concerns and multithread-related problems are not well studied in previous work.

Specifically, the following changes in software and development motivate a new study with modern software to answer those newly rising issues as well as to provide new answers to old questions:

- **More Effective Modern Debugging Tools:** Quite a few bug detection tools such as Purify [20] and Valgrind [27] have been recently proposed and widely used. An interesting question is whether they are helpful in minimizing the number of memory bugs in released code. To answer this question requires a new empirical study of bugs from some modern software developed after those debugging tools became available around 1997. If these tools become a standard practice in software testing and validation, there should be fewer memory bugs in modern software than that reported before for traditional software, and the diagnosis time for these types of bugs should be much shorter than that for other types of bugs. If the empirical results with modern software indicate otherwise, it may imply that these tools are not effectively used by programmers due to either too many false positives or to other reasons.

- **Software Architecture Shift:** Due to the recent advance in hardware, many modern computer systems, especially server systems, are all configured with multi-processors. As a result, many software architectures are multi-threaded or multi-processed to exploit the parallelism provided by hardware. These trends will continue especially since, with the continuous shrinking of transistor size and increasing chip area density, multithreaded/multicore (multiple processors on one chip or CMP) architecture is becoming a mainstream technology. For example, the Intel Pentium processor has already provided hyper-threading and dual-core capabilities. Multi-core technology is also embraced by other chip vendors including AMD, IBM and Sun. To exploit these technologies, increasingly more software is becoming multi-threaded. An interesting question is whether modern software has more concurrency bugs due to this software architecture shift? Do these bugs cause severe impacts on systems?

- **Emphasis on User-Friendly Interface:** In order to provide friendly user interface, graphical user interfaces (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 [26]. Therefore, GUI-related bugs may have become more pervasive and dominant. Further, since GUI modules and their development process are quite different from other modules, do they have different root causes?

- **Rising Security Concerns:** 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 [11, 31]. Unfortunately, previous work has not addressed how software errors affect system security. It is unclear how many reported bugs are related to security, and what types of security-related bugs there are, and how fast these security-related bugs are fixed.

- **New Software Development Paradigm:** Recently more and more software is developed using the open source paradigm that allows programmers from the Internet to read, redistribute, and modify the source code. For quality control, most open source software (**OSS**) projects usually allow only a small set of experienced developers to check code changes into the main branch of the software. The OSS development paradigm enables software to evolve at a much faster pace. Some preliminary experiences seem to indicate that this process produces better software than the traditional closed model, in which only a few programmers can see the source and everybody else can only use the binary code. This is why the OSS paradigm is also endorsed by many industrial companies including IBM and Sun. An interesting question is whether OSS actually takes a much shorter time to fix bugs.

Furthermore, most previous studies were based on a small set (usually with 60–500) of software bugs for the entire software revolution, which may result in a large experimental error and misleading results. For example, some bugs may not be included in the sampled set but could account for a significant fraction in the whole bug database of the evaluated software. Moreover, small size datasets also make it difficult to study the bug trend with the software evolution.

1.2 Contributions

To understand the effects of the above new factors on software errors, we analyze bug characteristics in two large and popular OSS projects, Mozilla and Apache Web Server, each of which contains up to 4 million lines of code and about 90 release versions developed over the last 8–10 years. We first manually examine 362 randomly selected bugs from the bug databases and study the bug distribution in three dimensions, root causes, impacts, and software components. Furthermore, we also study the statistical correlations among these dimensions, which has never been systematically studied before (to the best of our knowledge). In addition, we also study 257 security related bugs and 90 concurrency bugs to understand the characteristics of these emerging types of bugs.

To validate that our analysis results from the sampled datasets are representative, we use *natural language text classification* and *information retrieval* techniques to **automatically** classify **a large number (around 29,000)** of bug reports. Such a large dataset enables us to provide more accurate results such as trends of bug types with software evolution which are usually difficult to draw representative and accurate results from small datasets.

Our study has discovered several interesting findings:

- Memory bugs only account for 12.2–16.3%, much less than the 28–38% reported in previous studies [35, 36], indicating memory bugs are becoming less pervasive due to the available techniques to automatically detect them. Our results also show bug detection tools can effectively reduce the diagnosis and resolution time of memory bugs. However, we also found that there still exist many simple memory bugs such as NULL pointer dereferences and uninitialized memory reads, indicating memory bug detection tools have not been used at their full capacity.

- Semantic bugs are the major root causes, accounting for 81.1–86.7%, and their percentages increases with the maturity of software. Moreover, they also have severe impacts on system availability, contributing to 42.9–44.2% of crashes. Additionally, it takes a longer time to diagnose and fix semantic bugs, almost twice as memory bugs. Our results suggest that more effort should be put into automatically detecting and diagnosing semantic bugs.

- Our results show that security bugs are increasing significantly over time in terms of number and relative percentage. Among different root causes of security vulnerabilities,

memory related bugs contribute for only 8.8–17.2% but are usually severe, while semantic bugs are the dominant cause, accounting for 71.9–83.9%.

- GUI bugs have become the major ones in graphical interface software, accounting for 52.7% of bugs in Mozilla, and resulting in 28.8% of all crashes. Furthermore, most GUI bugs are caused by semantic errors, which indicates that designing good GUI test cases with good coverages is probably the only way to significantly reduce the number of GUI bugs.

- Concurrency bugs account for a small portion of bug reports, probably because they are underreported. We found that 55.5% of them cause hangs or crashes, which means that most of them are benign faults (fail stop) so that most failures caused by them should be able to recover using simple generic techniques such as restart or rollback and reexecute.

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

2. BUG SOURCES

To discover what has changed now, we study bugs from two large widely-used *new* OSS projects, Mozilla and Apache HTTP Server.
Randomly Collecting Bugs In our study, we focus on the characteristics of software errors that manifest at **run time**, that is, excluding new feature requests, compile-time errors, configuration errors, environmental errors, and software maintenance. To ensure correct classification, we only study **fixed** runtime bugs whose root causes can be identified from reports because unfixed bugs may be invalid and root causes described in the reports can be wrong. In this way, we randomly select 548 fixed bug reports from the Mozilla Bugzilla database [4]. As not every report describes a runtime bug, we manually investigate them and narrow down to 264 runtime bugs, and then classify them manually. We study bug characteristics based on these manual labeled data, and further use them to train and evaluate automatic classifiers for the whole bug database as described in Section 3.2. Similarly, we randomly select 209 fixed bug reports from the Apache Bugzilla database [1], and then manually classify 98 runtime bugs. In the rest of this paper, we use the general name "bugs" to refer to fixed runtime bugs.
Collecting Security Bugs We collect all of the 193 security vulnerabilities in Mozilla and all of the 64 in Apache Web Server kept in National Vulnerability Database (NVD) [5].

3. BUG CLASSIFICATION AND ANALYSIS

3.1 Bug Categories

We classify bugs in three dimensions, **Root Cause**, **Impact** and **Software Component**. According to root causes, bugs can be classified into three disjoint categories, *Memory*, *Concurrency*, and *Semantic*, whose definition as well as the definition of impact categories, is shown in Table 1. Memory bug and semantic bugs are further classified into sub-categories as shown in table 2. Bugs can also be classified based on their impacts and software components. The definition of each category is shown in Table 1.

Classifying Security Vulnerabilities Security vulnerabilities are manually classified in three dimensions. The root cause dimension is the same as that of general bugs. Based on impact, vulnerabilities are classified into four categories, *confidentiality* (unauthorized disclosure of information), *integrity* (unauthorized modification), *availability* (disruption of service), and *access* (unauthorized access). The third dimension is the NVD severity, which contains three levels, *High*, *Medium*, and *Low*, as defined in [3].

3.2 Automatic Classification

The whole bug databases contain hundreds of thousands of bug reports, so it may result in large statistical variances in distribution analysis by sampling only hundreds of bugs.

To verify the analysis results from the sampled datasets, we propose a novel method to automatically classify a large number of bugs, and then study bug characteristics based on such a large dataset. Specifically, we apply text classification and information retrieval techniques on the bug reports to automatically classify 29,000 bugs. Such a large dataset also enables us to perform study on trend. Our method consists of the following steps:

- **Preprocessing** In Bugzilla databases, each bug report may contain the following information: bug ID, summary, time, status, reporter, assignee, severity, bug description, discussion comments, test cases, attachments, etc. We include most of the information except time and attachments because time is irrelevant for our classification, and the major contents in attachments are source code that is hard for current classifiers to use. We represent bug documents in 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 bugs as a training set to produce classification models for different categories of bugs. We use several different classifier learning methods, including support vector machine(SVM) [37], Winnow, Perceptron, and Naive Bayes [33]. We choose the best one based on their accuracy.

  We randomly divide the whole sampling dataset into two halves: *training set* for learning and tuning, and *test set* for accuracy evaluation. Applying the classification methods on the training set, we can get several models for different categories. Since each method has some parameters, we explore the entire parameter space and use n-fold ($n = 5$) cross validation [21] to find out the best method with the best parameter setting based on the accuracy metrics described in Section 3.2. Additionally, to avoid the errors of accuracy evaluation caused by tuning, we use only the training set for tuning and use the test set for accuracy evaluation. Therefore, the test set does not affect parameter tuning, and so accuracy evaluation based on the test set could be representative for the whole dataset.

- **Evaluating Accuracy** To evaluate how good the classification models are, we can measure the prediction accuracy. Four different types of prediction results are possible from a binary classifier:

		Predicted Class	
		Yes	No
Actual Class	Yes	True Positive (T_+)	False Negative (F_-)
	No	False Positive (F_+)	True Negative (T_-)

We use the following metrics to evaluate accuracy:

1. *Precision* measures the portion of class members predicted correctly over all instances classified as class members. It is calculated as $P = \frac{T_+}{T_+ + F_+}$.

Dimension	Category	Description	Abbr.
Root Cause	Memory	Bugs caused by improper handling of memory objects.	Mem
	Concurrency	Bugs that happen only in multi-threading (or multi-processes) environment, including data race, deadlock, and synchronization.	Con
	Semantic	Inconsistent with the original design requirements or the programmers' intention. We consider all bugs as Semantic bugs unless they are already classified as Memory bugs or Concurrency bugs.	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 behave as expected.	Func
	Unknown	The impact cannot be identified from the bug report.	Unknown
Software Component	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

Table 1: Categories of three dimensions. Some categories and definitions are borrowed from *BugBench* [25]. Impact dimension and Software Component dimension each has a category called "Others" which contains bugs that can not be classified into the categories above.

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 the buffer boundary.	Overflow
	Double Free	One memory location is freed twice.	2Free
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	Do 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

Table 2: Subcategories of memory and semantic bugs. Some definitions are borrowed from *BugBench* [25] and a book [7]. Both Memory Bugs and Semantic Bugs have a category called "Others" which contains bugs that can not be classified into the categories above.

2. *Recall* measures the portion of class members predicted correctly over all actual class members. It is calculated as $R = \frac{T_+}{T_+ + F_-}$.

3. $F1$ is an even combination of precision and recall. It is defined as $F1 = \frac{2PR}{P+R}$. When precision and recall are equally important, $F1$ can be used.

For bug classification, the accuracy goal is both of high precision and recall, so we use $F1$ as the accuracy metric in learning parameter tuning.

- **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. Specifically, we use the 548 sampled fixed bug reports as training and test data to learn a model, and apply it on the whole database to identify runtime bugs. Among all 75,519 fixed bugs in the whole database for Mozilla, 28,928 bugs are identified as runtime bugs with precision of 89.6% in the test set. Then the classification model for each bug category is applied on the 28,928 runtime bugs.

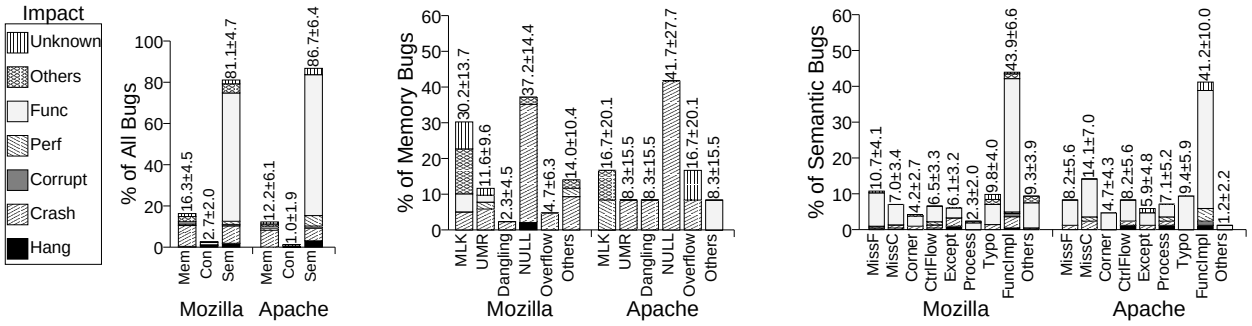
Some bug categories such as concurrency bugs only constitute a very small percentage of all reported bugs. Therefore, random sampling cannot provide enough data for classifier training. To solve this problem, we apply information retrieval techniques [22] on the bug database to retrieve the specific category of bugs.

3.3 Studying the Trend

To study the trend of different categories of bugs, including memory, concurrency, semantic, and security bugs, we calculate the number and relative percentage of bugs in each category reported in each year. Some previous work investigates the trend of bug density over releases [29], but it does not study the trend of different categories of bugs. Our analysis method on a large dataset enables us to study the change of different categories over time.

3.4 Measuring the Correlation

To study the correlation between two categories in different dimensions, we use a statistical metric called *lift*. The *lift* of category A_i in dimension A and category B_j in dimension B, $lift(A_i, B_j)$, is calculated as $\frac{P(A_i B_j)}{P(A_i) * P(B_j)}$, where $P(A_i B_j)$ is the probability that a bug belongs to both category A_i and B_j. If $lift(A_i, B_j)$ is equal to 1, it means $P(A_i B_j) = P(A_i)P(B_j)$, 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.



(a) All bugs (b) Memory bugs (c) Semantic bugs

Figure 1: Distribution of root causes with impacts. The numbers show 95% confidence level.

4. ROOT CAUSE ANALYSIS

We first present the analysis results based on sampled datasets in Section 4, 5 and 6, and then present the results based on automatic classification in Section 7 to confirm that the results from sampled datasets are representative.

In this section, we analyze the root causes of bugs in modern software and compare our results with those in the previous studies [35, 36], so that we can understand some important bug characteristic changes.

Figure 1 summarizes the distribution of bugs with different root causes and their corresponding impacts. From these figures, we can observe the following:

Memory Bugs Have Decreased. Figure 1(a) shows that memory bugs account for a relatively small fraction of all bugs, 16.3% in Mozilla and 12.2% in Apache. These percentages are much lower than the 28–38% reported in previous work [35, 36]. Since debugging tools are known to help detect/avoid memory bugs, this reduction probably benefits from using these tools in recent years.

Figure 1(b) shows that among memory bugs, NULL pointer dereference is a major cause, accounting for 37.2–41.7% in the memory categories, and most of them resulting in a system crash. Memory leak is another major cause, accounting for 16.7–30.2% of memory bugs, which is much more than the 8% reported previously [36]. This may be because memory leaks are relatively more difficult to detect without tools since their impacts may be "silent" within some time. Since most memory bugs can be detected by the existing tools such as Purify, Valgrind and Coverity [2], our results indicate that these debugging tools have not been used in *development* with their full capacity yet.

Semantic Bugs Are Dominant Root Causes. Figure 1(a) shows that the dominant root causes are semantic errors in both applications, accounting for 81.1% in Mozilla and 86.7% in Apache. These results are much more than the 55–66% reported in a previous study [36] (bugs excluding memory bugs and concurrency bugs). Therefore, semantic bugs not only remain to be the dominant root causes, but also seem to have become more dominant.

One possible reason may be that most semantic bugs are application specific and are different from memory bugs which are general for any applications. 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 more application specific. The percentage increase of semantic bugs may be attributed to the decrease of memory bugs. Further, the previous work [36] is based on commercial software, while our study is based on OSS whose specifications are not well defined as commercial software, and hence resulting in more semantic bugs in OSS.

In order to further understand what are the major causes among semantic bugs, in Figure 1(c), we show the breakdown of semantic bugs into subcategories. We see that 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 [9]. 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, similar to Daikon [15] and our previous work [24].

Interestingly, there are quite a few *simple* semantic bugs. For example, typo errors account for 9.4–9.8% of the semantic bugs. It indicates that careless programming is still causing many bugs, suggesting that the development environment should provide some tools for programmers to check for simple errors such as typos and copy-and-paste related bugs (as our previous work [23] does).

In order to further understand concurrency bugs, instead of randomly sampling, we use information retrieval technique to obtain possible concurrency bugs; the results are presented in Section 9.

5. IMPACT ANALYSIS

In this section, we study the distribution of impacts and the correlation between impacts and root causes.

Dominant Impacts. Figure 2 summarizes the distribution of different impacts with the corresponding root causes. It shows that incorrect functionality is the dominant impact; indeed, the percentage of incorrect functionality is 64.3–69.4%, much larger than the 35% reported in the previous work [35]. This is because the percentage of semantic bugs has significantly increased in modern software

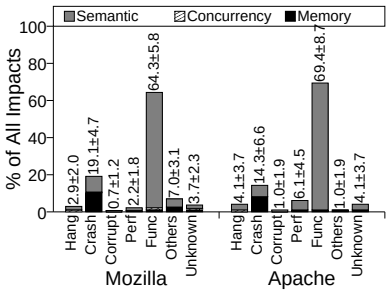


Figure 2: Distribution of impacts

Impact	Memory	Concurrency	Semantic	Memory Subcategories			Semantic Subcategories							
				MLK	UMR	NULL	MissF	MissC	CornerC	CtrlFlow	Except	Process	Typo	FuncImpl
Hang	0.77	9.43	0.77	0.00	0.00	2.06	0.00	2.20	0.00	2.36	5.08	0.00	0.00	0.35
Crash	3.31	0.73	0.55	0.78	3.05	5.08	0.22	0.68	1.13	0.73	1.95	0.00	0.73	0.38
Func.	0.11	0.65	1.19	0.23	0.30	0.00	1.31	1.31	1.01	1.08	0.70	1.21	0.86	1.33

Table 3: Correlation between root causes and impacts in Mozilla. Categories with too few examples are not shown due to statistical insignificance.

Component	Memory	Concurrency	Semantic	MLK	UMR	NULL	MissF	MissC	CornerC	CtrlFlow	Except	Process	Typo	FuncImpl
Core	1.84	1.06	0.83	1.71	1.97	1.85	0.75	0.99	0.82	1.06	1.14	0.99	0.35	0.81
GUI	0.31	1.09	1.14	0.29	0.38	0.36	1.32	1.01	1.06	0.81	0.73	1.14	1.45	1.15

Table 4: Correlation ($lift$) between root causes and software components in Mozilla.

and most of them cause incorrect functionality as shown in Figure 1(a).

In contrast, because memory bugs have decreased substantially, the severe impacts on availability, including crashes and hangs, have also been reduced to 18.4–22.0%. However, although the percentage of these impacts is reduced, they still account for a considerable portion and can significantly compromise availability. Thus, recovery techniques are still needed to provide highly available services.

Correlations Between Causes and Impacts. Figure 2 also shows that the major cause of crashes is memory bugs, accounting for 53.8–57.1%, which is similar to what has been found in the previous work [35]. Semantic bugs are clearly more likely to cause incorrect functionality, accounting for 96.6–98.5%, which is also consistent with the previous studies. However, among the crashes, 42.9–44.2% are contributed from semantic bugs, which is higher than that reported in the previous work [35]. It indicates that although most semantic bugs result in incorrect functionality, they are also one of the important factors of unavailability.

In order to further understand the correlation between causes and impacts, we show the correlation metric $lift$ in Table 3. Not surprisingly, here we see that hanging has an extremely strong correlation with concurrency bugs, while crashing has a strong correlation with memory bugs. The (incorrect) functionality impact has a relatively strong correlation with semantic bugs, though no specific subcategory of semantic bug has an exceptionally high correlation. Interestingly, although semantic bugs are overall negatively correlated with crashing and hanging, some specific semantic bugs are positively correlated with them.

6. BUGS IN DIFFERENT COMPONENTS

In this section, we study where the dominant bugs are located in software components, including core, GUI, network, I/O, and others. To the best of our knowledge, there is no previous work on studying bug characteristics from this perspective. Since the modern software tends to emphasize friendly user interfaces, we are particularly interested in studying characteristics of GUI bugs. Due to the lags of GUI testing techniques, we believe that characterization of GUI bugs is very important.

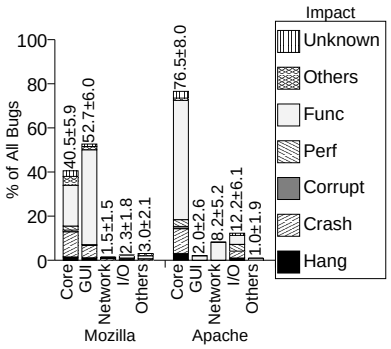


Figure 3: Distribution of bugs in software components

Bug distribution in software components Figure 3 shows the distribution of bugs within different components and their impacts. As we can see, GUI modules are critical for software reliability in modern graphical interface software, such as Mozilla. GUI modules account for more than half of bugs in Mozilla and also cause around 30% of Mozilla crashes. Unfortunately, GUI testing techniques still lag far behind now. Future research should pay more attention to GUI related testing and debugging.

Root causes of GUI bugs The correlations between software components and root causes for Mozilla are shown in Table 4. Interestingly, the results indicate that bugs within 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 and concurrency bugs. Such difference is likely because the GUI modules and their development process are quite different from other modules. Further, GUI bugs are correlated to the subcategories of missing features and other wrong functionality implementation that are application specific. The results indicate that the existing debugging tools aiming at memory bugs are unsuitable for GUI bugs, while study on application-specific semantic bugs can be helpful.

7. AUTOMATIC CLASSIFICATION

This section presents our preliminary results with automatic classification on Mozilla so that we can confirm that the results of distribution based on the sampled bugs in previous sections are representative. Currently we are still improving the classification accuracy by exploiting higher level information to represent documents and optimizing the queries in information retrieval for small categories as described in Section 3.2.

Distribution of Root Causes. Table 5 shows the distribution of bugs with different root causes. Compared with the results using sampled bugs in Section 4, the percentage of memory and semantic bugs are similar, which indicates that the distribution results based on sampled bugs and a large dataset are consistent.

Trend of Root Causes. In order to understand the trend of bug distribution along software evolution, we plot the relative percent-

Cause	Percentage	Classification Accuracy		
		Precision	Recall	F1
Memory	13.2%	0.736	0.929	0.821
Semantic	78.0%	0.874	0.954	0.912

Table 5: Distribution of root causes based on automatic classification. Precision, recall, and F1 are defined in Section 3.2.

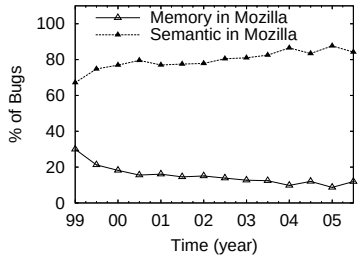


Figure 4: Trend of bugs with different root causes

Impact	Percentage	Classification Accuracy		
		Precision	Recall	F1
Hang	2.5%	0.714	0.833	0.769
Crash	13.0%	0.852	0.852	0.852
Data corruption	1.8%	0.400	1.00	0.571
Performance degradation	2.2%	0.500	0.500	0.500
Incorrect Functionality	75.2%	0.860	0.909	0.884

Table 6: Distribution of impacts based on automatic classification.

ages of memory and semantic bugs in Figure 4. Although software becomes mature and stable, semantic bugs that are specific to applications still remain dominant. Semantic bugs increase gradually with the maturity of software, while memory bugs decrease gradually. However, both types have not changed too much, compared with concurrency bugs that are decreasing significantly (Section 9). Bug detection tools such as Purify only affect the relative percentage of memory bugs a little during this period, because they had become available before 1999 and are used during these years.

Distribution of Impacts. Table 6 shows the distribution of bugs with different impacts. Compared with the results using sampled bugs in Section 4, the distribution is similar.

8. SECURITY RELATED BUGS

Security is becoming increasingly important recently. However, previous work has not addressed how software errors affect security. To study the security related bugs, we collect 193 security vulnerabilities in Mozilla and 64 in Apache HTTP Server from NVD [5]. In this section we would like to understand (1) whether security bugs are increasing, and (2) what types of bugs cause vulnerability.

Are Security Related Bugs Increasing? Figure 5(a) shows the numbers of vulnerabilities along time. We also normalize the numbers to relative percentage by comparing with the total fixed bugs reported in Bugzilla during the corresponding year. The trend shows that the numbers of vulnerabilities are increasing for both Mozilla and Apache. Although there are some exceptional points in some years, such as Mozilla in 2003 [2], the normalized percentages are increasing significantly over the recent years. It demonstrates that security issue is becoming *increasingly important* for both client and server software.

What Types of Bugs Lead to Vulnerabilities? Figure 5(b) shows the distribution of root causes and severity for security bugs. Surprisingly, memory bugs account for only 8.8–17.2%, while semantic bugs cause 71.9–83.9% of vulnerabilities. This finding is against the belief that buffer overflows are the most common form of security vulnerability [13]. The reason may be that most buffer overflows have been detected and fixed before release due to the available debugging tools recently.

The different distribution of impacts for Mozilla and Apache shown in Figure 5(c) indicates that vulnerabilities have different impacts on client and server systems. For client systems, most security bugs result in unauthorized accesses, while for Apache Web Server, both availability and unauthorized accesses are the major vulnerabilities.

[2]Data in 2005 only contain bug reports until July.

Cause	Severity			Impact			
	High	Medium	Low	Confidentiality	Integrity	Availability	Access
Mem	1.80/1.62	0.55/2.91	0.00/0.31	0.00/0.00	0.00/0.65	3.01/1.29	1.31/2.22
Con	0.00/0.00	1.65/8.00	2.51/0.00	6.40/4.92	0.00/7.11	0.00/0.00	0.00/0.00
Sem	0.87/0.85	1.09/0.52	1.14/1.17	1.11/1.18	1.19/1.08	0.85/0.93	0.92/0.73
Others	0.58/1.19	1.27/0.00	1.94/1.12	1.65/0.82	1.82/0.00	0.00/1.19	0.59/0.00

Table 7: Correlation ($lift$) between root causes and severity/impacts for security related bugs. The first number is for Mozilla, and the second one is for Apache Web Server.

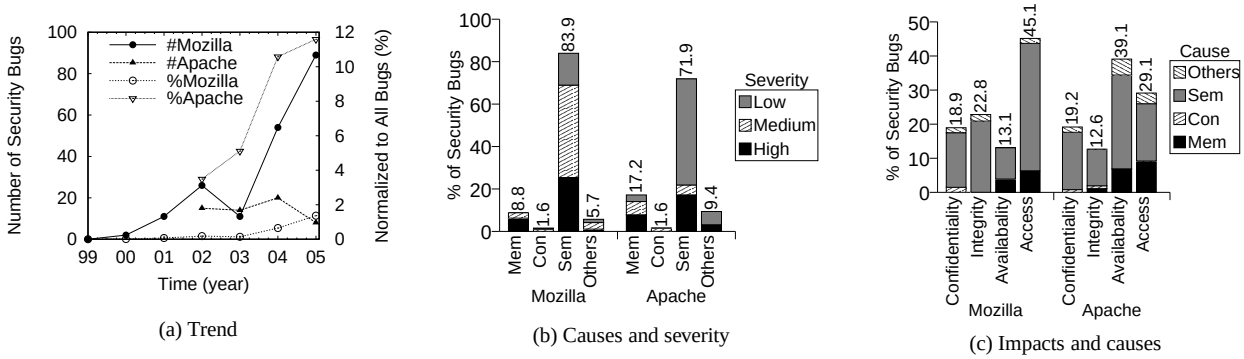
The correlation metric $lift$ shown in Table 7 indicates that memory bugs are more severe than the other types of bugs, and they are likely to cause unavailability and unauthorized accesses because systems can be intruded by exploiting buffer overflows [6]. Thus besides reducing the number of buffer overflows in source code using bug detection tools, it is also important to prevent attackers from exploiting them. StackGuard [12] is such a tool that can protect against stack smash attacks. In addition, semantic bugs are less severe than memory bugs, and are likely to cause confidentiality and integrity problems.

9. CONCURRENCY BUGS

Because concurrency bugs account for only a small portion in bug report databases, it is difficult to study their characteristics by randomly sampling hundreds of bug reports. For example, Chandra and Chen collected only 12 transient bugs in their study [8]. To this end, we use information retrieval techniques to obtain potential concurrency bugs, and then manually verify these potential concurrency bugs. In this way, we have collected 90 concurrency bugs from the top of retrieval results for Mozilla. The larger available dataset allows us to study the characteristics of concurrency bugs, which has never been done before.

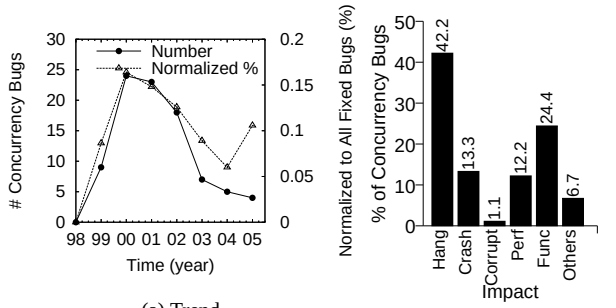
Are Concurrency Bugs Increasing? Figure 6(a) shows the changes of concurrency bugs over time in Mozilla. Unexpectedly, the number of concurrency bugs increased only in the first two years (1999–2000) but sharply decreased later. To show the relative percentage, we normalize the number by all fixed bugs in the corresponding year (which is not absolute percentage because we only collect part of concurrency bugs). The relative percentage has the same trend except in 2005. The number and relative percentage of concurrency bugs increased, and then decreased as the software became stable, which is different from the trend of memory and semantic bugs shown in Figure 4.

However, the previous studies [8, 18] indicated the perception that the relative percentage of Heisenbugs (transient bugs) should increase when the software becomes stable because the other non-transient bugs are fixed but the transient bugs are still remaining in the software. Our finding about the trend of concurrency bugs does not follow this perception. A possible explanation is that the hard-to-reproduce concurrency bugs are underreported. Before the software became stable (the first two years), most of the concurrency bugs that could be easily reproduced were reported and fixed, and the other concurrency bugs that were difficult to reproduce were not reported and remained in software. When we verified the concurrency bugs, we found that *most of them could not be reproduced by developers with an acceptable probability*, say more than once out of ten times. Therefore, when the software became stable, even though the software still contained many concurrency bugs, the



(a) Trend (b) Causes and severity (c) Impacts and causes

Figure 5: Security related bugs collected from NVD



(a) Trend (b) Impacts and components

Figure 6: Concurrency bugs in Mozilla. Data in 2005 only contain bug reports until July.

failures caused by them were unlikely to be reported by users or to be fixed by developers because they were difficult to reproduce.

What Are the Impacts from Concurrency Bugs? Figure 6(b) shows the distribution of impacts from concurrency bugs. Compared with all bugs shown in Figure 2, concurrency bugs can cause much more severe impacts on systems, which is consistent with previous work [36]. Specifically, 42.2% of concurrency bugs can cause hanging due to synchronization errors and deadlocks, 10 times higher than all bugs. Further, 55.5% of concurrency bugs lead to fail-stop failures (crashes and hangs), which can be detected and recovered by generic recovery techniques.

10. RELATED WORK

Many efforts have been made to study fault related characteristics of large software systems [8, 10, 16, 19, 28, 29, 30, 32, 36]. 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 [36] found that memory bugs are a major type and can have high impact. Chandra and Chen [8] showed that only 5–14% of faults are triggered by transient conditions in release software, which is against the intuition that transient bugs are more difficult to reproduce and hence to fix so that most bugs left in release software should be transient. Ostrand and Weyuker [29] found that the majority of post-release faults occurred in files that had no pre-release faults. This observation contradicts the conventional wisdom and suggest most testing efforts for post-release software be put on previous fault-free or less-faulty parts instead of most faulty parts.

11. CONCLUSION

This paper investigates the impacts of new factors on software errors and studies the bug characteristics in two large modern OSS projects. We manually collect 709 bugs, including 362 randomly sampled bugs, 257 security related bugs from NVD, and 90 concurrency bugs. We then classify bugs from different dimensions (root causes, impacts and software components) and study the correlation between categories in different dimensions. Additionally, we use text classification and information retrieval techniques to automatically classify tens of thousands of bugs so that we can verify the analysis results from sampled datasets and perform more representative study on bug trend. Our study found several new interesting bug characteristics in modern OSS that can provide useful guidelines for related research.

12. REFERENCES

[1] ASF bugzilla. http://issues.apache.org/bugzilla, 2005.

[2] Coverity: Automated error prevention and source code analysis. http://www.coverity.com, 2005.

[3] Def. of NVD severity metric. http://nvd.nist.gov/faq.cfm#8, 2005.

[4] Mozilla.org Bugzilla. https://bugzilla.mozilla.org, 2005.

[5] National vulnerability database. http://nvd.nist.gov, 2005.

[6] Anonymous. Once upon a free(). http://www.phrack.org/phrack/57/p57-0x09.

[7] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[8] S. Chandra and P. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, 2000.

[9] R. Chillarege, W.-L. Kao, and R. G. Condit. Defect type and its impact on the growth curve. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, 1991.

[10] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, 2001.

[11] C. Cowan. Software security for open-source systems. *IEEE Security and Privacy*, 2003.

[12] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, Jan 1998.

[13] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX)*, Jan. 2000.

[14] A. Endres. An analysis of errors and their causes in system programs. In *Proc. of the Intl. Conf. on Reliable software*, 1975.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, 2001.

[16] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE TSE*, 2000.

[17] R. Glass. Persistent software errors. *IEEE TSE*, 7(2), 1981.

[18] J. Gray. Why do computer stop and what can be about it? In *the 5th Symposium on Reliablity in Dist. Softw. and Database Sys.*, 1985.

[19] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang. Characterization of linux kernel behavior under errors. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, 2003.

[20] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the USENIX Winter Technical Conf.*, 1992.

[21] T. Joachims. *Learning to classify text using support vector machines*. Kluwer Academic Publishers, 2002.

[22] K. S. Jones, S. Walker, and S. E. Robertson. A probabilistic model of information retrieval: development and comparative experiments part 2. *Inf. Process. Manage.*, 36(6):809–840, 2000.

[23] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Sixth Symposium on Operating Systems Design and Implementation*, 2004.

[24] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'05)*, 2005.

[25] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: A benchmark for evaluating bug detection tools. In *Bugs 2005 (Workshop on the Evaluation of Software Defect Detection Tools) on Programming Language Design and Implementation (PLDI) 2005*, 2005.

[26] A. M. Memon. GUI testing: Pitfalls and process. *Computer*, 2002.

[27] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *the 3rd Workshop on Runtime Verification*, 2003.

[28] T. Ostrand and E. Weyuker. Collecting and categorizing software error data in an industrial environment. *Journal of Sys. and Softw.*, 1984.

[29] T. Ostrand and E. Weyuker. The distribution of faults in a large industrial software system. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 2002.

[30] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE TSE*, 31(4), 2005.

[31] C. Payne. On the security of open source software. *Information Systems Journal*, 2002.

[32] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE '03: Proceedings of the 23th International Conference on Software Engineering*, pages 465–475, 2003.

[33] D. Roth and D. Zelenko. Part of speech tagging using a network of linear separators. In *COLING-ACL*, pages 1136–1142, 1998.

[34] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: An empirical study. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, 1999.

[35] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. In *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, 1991.

[36] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *FTCS '92: 22nd Annual International Symposium on Fault-Tolerant Computing*, 1992.

[37] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.