Empirical characterization of faults in complex software systems, and strategies to mitigate them

D. Cotroneo, R. Natella, R. Pietrantuono
Università degli Studi di Napoli Federico II

1 Introduction

With software systems becoming increasingly large and complex, coping with software defects ("bugs") is more and more difficult. Despite good development practices, thorough testing, and proper maintenance policies, software is still shipped with a significant number of bugs, which represent a severe threat for the reliability and security of critical infrastructures.

The practice of defect analysis is recognized as an essential task for software process and product improvement. Understanding the type of residual bugs is fundamental for adopting proper countermeasures in current and future software releases. Researchers, across years, have analysed bugs from different viewpoints in order to improve the knowledge about their characteristics. Regardless of the semantics of the error committed by a developer, a fundamental aspect in bug comprehension is related to the process by which a bug manifests itself as a failure. In fact, while static properties of a bug (e.g., its type, or origin) are related to how a bug is introduced in the code, there are different causes for a bug provoking a failure.

Expectedly, many bugs systematically cause the same failure on a given (sequence of) input(s). On the other hand, there is a non-negligible set of bugs that cause a failure depending on the state of the execution environment, and hence may appear as non-deterministic or transient failure, as the failure does not occur unless the environment is in a certain state. The latter category contains bugs that have likely escaped testing, since their exposure may be a rare event. Specific strategies are required to cope with these bugs. Examples are fault tolerance strategies that mask faults, for instance by reinitializing the software state and retrying the failed operation, and verification techniques that do not need to actually reproduce the fault trigger during execution, such as code reviews and model checking. Instead, faults that can easily be triggered require more thorough testing in order to improve reliability.

Therefore, knowing how bugs manifest themselves (namely, the *fault triggering* conditions that make the bug to surface as a failure, and to impact on the rest of the infrastructure) is strictly tied to the effectiveness of in-process fault detection/correction activities, and of runtime mechanisms against resid-

ual bugs. We hereafter analyze the characteristics of bugs reported from the perspective of their fault triggering condition, in a set of large open-source software (OSS) projects, with the goal of understanding the common features of the bugs from the triggering perspective, as this point of view is directly related to the effectiveness of V&V and fault-tolerance techniques.

2 Approach

Past research studies attempted to classify bugs according to the point of view of fault triggering, by adopting different terminologies with slightly different meanings, such as *hard* vs. *soft faults* [1], or *transient* vs. *non-transient bugs* [2]; other studies focused on *concurrency bugs* [3, 4, 5, 6] as the class of faults causing failures that are difficult to reproduce. To examine fault triggers in a comprehensive way, Grottke and Trivedi [7] developed the following definitions concerning the conditions related to the *fault activation* and the *error propagation*, that we adopt and extend in our analysis:

- **Bohrbug**: a bug which can easily be isolated and which manifests consistently under a well-defined set of conditions, because its activation and error propagation lack "complexity".
- **Mandelbug**: a bug whose activation and/or error propagation are "complex", where *complexity* can be caused by the possibility of a time lag between the fault activation and the failure occurrence, or by the possible influence of indirect factors, such as the interactions of the software application with its system-internal environment (hardware, operating system, or other applications), the timing of inputs and operations (relative to each other), and the sequencing of inputs and operations.

There is a further subtype of Mandelbugs, that is responsible for a phenomenon increasingly being studied, known as *process aging* [8] or *software aging* [9]. Software aging is a typical problem of long-running software systems in which an increasing failure rate and/or degraded performance is observed. This Mandelbug subtype is defined as follows.

- **Aging-related bug**: a Mandelbug that is capable of causing an increasing failure rate and/or degraded performance, because the rate at which it is activated and/or the rate at which errors caused by it are propagated into (partial) failures increases with the total time the system has been running. Often, such an increasing error propagation rate is caused by the accumulation of internal error states. Since aging-related bugs are a subtype of Mandelbugs, each Mandelbug is either an *aging-related bug* or a Mandelbug that does not cause software aging, called a *non-aging-related Mandelbug*.

Note that these definitions do not focus on the circumstances of one specific manifestation of the bug (e.g., the one that made the testers notice its presence,

or that helped them locate it in the code), but rather on its *potential* manifestation characteristics and its *inherent* features [10]. For example, even if a developer is able to reproduce a failure in a well-controlled environment, the underlying fault is classified as a Mandelbug if its manifestation can result in a transient failure at the user site because one of the criteria of complex fault activation or error propagation (as laid out above) applies. Similarly, even if an aging-related bug is detected before it has had the chance to actually lead to a decreasing performance (e.g., during a code inspection), the fault is still considered aging-related.

Adopting the above classification, we performed an extensive analysis of fault triggers in four large OSS projects [11]: the Linux kernel, the MySQL database management system, the Apache HTTPD server, and the Apache AXIS Web services framework. The analysis of these OSS projects, which are often adopted in mission- and business-critical scenarios, provides us insights about Mandelbugs in complex software systems. These projects have different nature (e.g., in terms of type of system, size, and programming language), which allows us to relate the type of system with the type of bugs. Furthermore, dealing with OSS projects enables us to publicly release our data to the research community (`http://goo.gl/aeKoGR`), allowing other researchers to adopt the classification more easily, and to carry out further analyses based on our data.

From publicly-available bug repositories, we inspected problem reports describing the failure occurrences observed, the underlying bugs, and their fix. Based on our classification procedure, we classified each unique fault as Bohrbug (**BOH**), non-aging-related Mandelbug (**NAM**), or aging-related bug (**ARB**), and then analyzed its features. Moreover, we extended the classification to provide additional insights about subclasses of the NAM and ARB categories. Our analysis reveals the following main findings:

- The proportion of Mandelbugs significantly varies among systems, and can be close to the proportion of Bohrbugs as in the case of the Linux kernel; the proportions can be related to both the size and the nature of the system.

- In every project, the proportion of Mandelbugs seems to converge to a constant value during the lifecycle, although past studies hypothesized that the percentage of Mandelbugs should be predominant in the long term. This may be explained by ineffective quality assurance and testing activities, and by the possibility of introducing new bugs during the project lifecycle.

- The analysis of subtypes indicates that timing-related faults are the largest part of Mandelbugs, although other Mandelbugs, such as those involving interactions with other software and hardware, account for a remarkable share. Similarly, memory-related aging-related bugs predominate, although leaks related to system-dependent data structures are also frequent.

- The time to fix a bug is significantly affected by the bug type, and strategies specifically tailored for Mandelbugs would certainly help.

2.1 Extended bug type classification

To classify bugs more in detail, we define the following subtypes of a non-aging-related Mandelbug (NAM), based on the different kinds of complexity in fault triggering conditions:

- **LAG**: there can be a time lag between the activation of the fault and the occurrence of a failure;
- **ENV**: the activation and/or error propagation is influenced by the interactions of the software application with its system-internal environment;
- **TIM**: the activation and/or error propagation is influenced by the timing of inputs and operations;
- **SEQ**: the activation and/or error propagation is influenced by the sequencing (i.e., the relative order) of operations.

Of course, these subcategories could also be employed for ARBs. However, it can be expected that the LAG subclass would apply to almost all of them. It is thus more informative to distinguish ARBs according to the various underlying reasons for the software aging phenomenon. Based on our definition of an ARB as well as on the software aging literature [12, 13], we identify the following ARB subtypes:

- **MEM**: ARBs causing the accumulation of errors related to memory management (e.g., memory leaks, buffers not being flushed);
- **STO**: ARBs causing the accumulation of errors that affect storage space (e.g., the bug consumes disk space);
- **LOG**: ARBs causing leaks of "other logical resources", that is, system-dependent data structures (e.g., sockets or inodes that are not freed after usage);
- **NUM**: ARBs causing the accumulation of numerical errors (e.g., round-off errors, integer overflows);
- **TOT**: ARBs in which the increase of the fault activation/error propagation rate with the total system run time is not caused by the accumulation of internal error states (e.g., due to a bug in the Patriot missile defense system [12, 14], the system runtime was incorrectly processed, but the error produced was only propagated into a failure if the system had been running for more than eight hours; error states did not accumulate).

2.2 Bug sources

We considered four open-source software systems with public and actively-used bug repositories, which provided us with a large number of bugs for the analysis. The chosen software systems are widely adopted in business-critical contexts [15], and they cover different types of software: (i) the Linux kernel, a feature-rich OS used in several domains, from embedded systems to supercomputers; (ii) MySQL, one of the most-used database management systems, accounting for a significant market share among IT organizations; (iii) the Apache HTTPD server, and (iv) the Apache AXIS framework for Web services, adopted by many companies for running their Web applications.

Since these systems are very large and have been around for a long time, tens of thousands of problems have been reported by their users; hence it is unrealistic to analyze all of them. We therefore selected a subset of these components for each project, and focused the analysis on the problem reports related to them. The selected components/subsystems were: *Network Drivers*, *SCSI Drivers*, *EXT3 Filesystem*, and *Networking/IPV4* for Linux; *InnoDB Storage Engine*, *Replication*, and *Optimizer* for MySQL; *Apache httpd_core*, *Apache httpd_mod_proxy*, *Apache httpd_mod_cgi*, *Apache httpd_mod_ssl* for Apache HTTPD; for Apache AXIS, we inspected all reports but those related to the *Distribution*, *Documentation*, and *Samples* areas (i.e., only reports about problems that can affect the system during its execution). In the selection, we accounted for the relevance of subsystems/components in terms of usage and number of problem reports, as well as for the coverage of diverse functionalities of the system and of a significant share of the system code. Table 1 provides, for each project, its programming language, the total size in LoC (computed using the `sloccount` utility) of the whole project and of the considered components, the number of problem reports that have been marked as "fixed" and "closed" by the developers (i.e., a fix was found and included in the source code), and the time frame during which these reports were issued.

The bug repositories[1] provide a large amount of information. Although projects may slightly differ with respect to the gathered information (e.g., they use different versioning schemes or fault severity scales), all reports provide:

- the type of the problem report (e.g., if it is a bug report or a request for a new feature);
- the date it was opened, closed, and last modified;
- the severity of the problem (i.e., the perception of its effects by users and developers);
- the version(s) affected by the problem;
- the component or subsystem affected by the problem;

[1] Available at https://bugzilla.kernel.org (Linux 2.6), http://bugs.mysql.com (MySQL 5.1), https://issues.apache.org/bugzilla (Apache HTTPD 2), and https://issues.apache.org/jira/secure/IssueNavigator.jspax (Axis 1).

Table 1: Overview of the considered projects.

Project	**Language**	**LoC (selection)**	**LoC (project)**	**#reports (selection)**	**#reports (project)**	**Time frame**
Linux	C	1.31M	9.58M	346	3914	Jul 2003 - May 2011
MySQL	C/C++	453K	1.1M	244	894	Aug 2006 - Feb 2011
HTTPD	C	145K	195K	157	405	Mar 2002 - Oct 2007
AXIS	Java	80K	80K	216	226	Jul 2001 - Nov 2005

- some textual messages describing the effects of the problem, its diagnosis by developers, and information on whether and how it can be reproduced (e.g., the inputs for triggering the failure behavior at the user's site, or a test case accompanying the bug fix);
- the status (e.g., the problem has been assigned to a developer, it has been solved, etc.).

To work with reliable bug descriptions, we filtered the bug repository, focusing on problem reports that had been solved (i.e., marked as "fixed" and "closed"). Moreover, we restricted the analysis to reports related to the above-mentioned components as well as to stable and mature system versions; in particular, the considered versions were Linux 2.6, MySQL 5.1, Apache HTTPD 2, and Apache Axis 1.

2.3 Classification procedure

Given a problem report, to classify the related bug as a Bohrbug, a non-aging-related Mandelbug, or an aging-related bug, we conducted a manual analysis by examining the textual descriptions and, if available, the test case to reproduce the failure occurrence, the available patches, and additional information attached to the problem report. We defined a classification procedure consisting of the following steps to classify faults in a rigorous way:

1. The problem report was first examined to make sure that it was related to a unique bug; i.e., problems turning out to be operator errors, requests for software enhancements, and duplicates were removed from the analysis. A report was considered a duplicate if either a field in the report or the textual description indicated that the reported problem was caused by the same underlying bug as another report already included in our study.

2. The report was then searched for any information on the activation conditions of the bug (e.g., the set of events and/or inputs required to trigger errors), its error propagation (e.g., how the bug affected the program state and how an erroneous state propagated through the running system), and the failure behavior (e.g., the bug effects perceived by the users).

3. The bug was classified as an ARB if there were indications that the rate with which it is activated and/or the rate with which errors caused by it are propagated into (partial) failures can be an increasing function of the total time the system has been running (e.g., the report refers to leakage and/or gradual corruption of resources, or to the accumulation of numerical errors). Typically, the information in the failure report allowed us to determine the ARB subtype (MEM, STO, LOG, NUM, TOT) as well (e.g., because it was reported that memory expected to be freed had not been freed). Sometimes, it was merely known that the failure rate of a bug tended to increase over time (e.g., because it caused a failure only

after a certain function had been called multiple times), but there was not enough information about the exact failure mechanics, like the presence of error accumulation. In such a case, we classified the bug as an ARB of unknown subtype (**ARU**).

4. A bug that was not an ARB was classified as a NAM if we found indications that one of the types of "complexity" of the activation and/or error propagation, embodied in the four subtypes LAG, ENV, TIM, and SEQ, applied to it. Sometimes, we did not have sufficient information about the activation and error propagation conditions of a bug that was reported to sporadically cause failures that could not be reproduced. We then classified this bug as a NAM of unknown subtype (**NAU**).

5. If there was evidence that the bug was neither an ARB nor a NAM, we classified it as a Bohrbug (BOH).

6. Sometimes, a report did not contain sufficient details to classify the underlying bug as an ARB, NAM, or BOH. It was then labeled as a bug of unknown type (**UNK**).

During the manual analysis, further information was extracted for the purpose of our analyses: the time at which the report was opened and closed, and the severity stored in the bug repository. To clarify the classification, Table 2 shows examples of NAMs and ARBs, along with statements from the reports that provide information about the fault activation, the error propagation, and the failure occurrence.

3 Results of the bug analysis

Our inspection of problem reports, using the procedure previously described, provided a large set of bug data. We first examine the relative frequencies of the bug types in the considered projects, and relate the results with the features of the considered projects. Subsequently, we analyze bug types with respect to some relevant features, including the time to fix the bugs and their severity.

3.1 Bug type proportions

Table 3 summarizes the absolute numbers and the percentages of each bug type. Among the 963 problem reports, we identified 852 *actual bugs*: these reports include neither operator errors nor duplicates nor problems that do not affect the operational software, such as documentation and compile-time issues. A subset of 816 bugs was classified as BOH, NAM, or ARB. The remaining bugs, which we refer to as UNK, were lacking information for classifying them with certainty. Most of these bugs belong to the Linux project: for this system, some problem reports do not provide a precise diagnosis of the bug, since the related failure disappeared in newer versions of the system (e.g., the bug did not manifest itself anymore after a major rewrite of a module or subsystem).

Table 2: Examples of NAMs and ARBs.

Project	Bug ID	Type	Description
MySQL	54453	NAM/SEQ	"if you 'alter table .. rename to ..' on a table that has an active transaction open and UNIV_DEBUG is defined, mysqld crashes"
Linux	7207	NAM/LAG	"[The e1000 network driver at suspend/resume does not] explicitly free and allocate irq [...] Restarting the network solved the problem"
HTTPD	8184	NAM/ENV	"The error only occurs intermittently [...] It behaves as if requests are being distributed (via round-robin or the like) and handled sometimes by a worker thread that is not properly initialized"
AXIS	1270	ARB/MEM	"Strings and char[]s are being leaked"
Linux	32832	ARB/LOG	"In 2.6.35 and earlier, shutdown(2) will fully remove a socket. This does not appear to be true any more and is causing software to misbehave."
HTTPD	13511	ARB/STO	"Apache child processes will die trying to write logs which have reached 2GB in size."

Comparing the projects with respect to their relative percentages of BOH, NAM, and ARB, it is possible to notice significant differences. In the Linux project, there are more Mandelbugs than Bohrbugs (NAMs and ARBs together account for more than 50% of all bugs), while in the remaining projects the percentage of Bohrbugs is predominant: this percentage ranges between 56.6% and 92.5%. We believe that the first cause for this result is the different nature of the considered projects: Linux and operating systems in general are tightly related to hardware devices; this makes them more prone to incorrect interactions with the hardware and to bugs in event handling, which can lead to transient failures. Another reason for the high percentage of Mandelbugs is the presence of several complex and tightly-interacting subsystems in the Linux kernel. It seems that the proportion of NAMs decreases as we move up in the "software stack"; that is, the proportion is higher for "low-level" code, such as an operating system, and lower for "high-level" code, such as middleware for web applications. This can be expected since in "high-level" code there are fewer interactions with the hardware and less resource management burdens.

We observe that the percentage of ARBs is approximately the same for the Linux, MySQL, and HTTPD projects. Instead, for AXIS the percentage of ARBs is lower than for the other three projects. This can be explained by considering the kind of system and by the fact that AXIS has been developed using the Java language, which provides automated memory management through garbage collection. By contrast, the other three projects adopted the C and C++ languages, in which memory management is handled by developers, and which are therefore more prone to software aging issues. However, it is impor-

Table 3: Total numbers and percentages for each bug type.

Project	**#actual bugs**	**#classified bugs**	**#BOH**	**#NAM**	**#ARB**	**#UNK**	**%BOH**	**%NAM**	**%ARB**	**%UNK**
Linux	289	267	122	121	24	22	42.2	41.9	8.3	7.6
MySQL	221	209	125	67	17	12	56.6	30.3	7.7	5.4
HTTPD	143	141	116	15	10	2	81.1	10.5	7.0	1.4
AXIS	199	199	184	7	8	0	92.5	3.5	4.0	0.0

Table 4: Estimated fault densities for each bug type.

Project	**#bugs/kLoC**	**#BOH/kLoC**	**#NAM/kLoC**	**#ARB/kLoC**
Linux	0.3434	0.1569	0.1556	0.0309
MySQL	0.4939	0.2954	0.1583	0.0402
HTTPD	3.1054	2.5548	0.3304	0.2202
AXIS	26.1994	24.2245	0.9216	1.0532

tant to note that Java software is also subject to software aging, even in the presence of garbage collection: This happens in the case of objects that are no longer needed but still referenced, which prevents the garbage collector from reclaiming them [16].

Another perspective on the relative importance of the bug types is given by Table 4, which provides an estimated fault density, expressed in faults per kLoC, for each type of bug and each project. To obtain these fault density estimates, we divided the estimated total number of bugs of each type (calculated by multiplying the total number of bug reports, including reports that are still open and UNK reports, with the respective bug type proportion among all *classified* bugs) by the LoC of the considered components shown in Table 1. This computation necessarily makes the assumption that the bug type proportions among the reports classified are the same as among those bug reports not yet closed or not classified. Of course, the considered projects exhibit different *#bugs/kLoC* ratios, which is a result of the software development process and of quality assurance activities. Nevertheless, we can notice different trends for individual bug types. In fact, the ratio *#BOH/kLoC* decreases fast with an increase in the project size; instead, the decrease in the *#ARB/kLoC* and *#NAM/kLoC* ratios is slower. Therefore, when a large software project is considered, the fault densities for Bohrbugs and Mandelbugs may be similar, while for smaller projects the fault density for Bohrbugs tends to be higher. Note that in our sample of projects, there is a high dependency between code size and the kind of system (e.g., Linux is both a large and a "low-level" software); we therefore cannot separate these effects, and both are likely to have an influence on the fault densities.

Figure 1, Figure 2, and Figure 3 show the evolution of the BOH, NAM, and ARB proportions among the classified bugs during project life. For all the projects, the proportions stabilize around a constant value after about two years after project birth. Another interesting result is that the ARB proportions also settle to constant values, which are about the same for three of the four projects (as reported above).

It could have been expected that the proportion of Mandelbugs increases with time: according to Gray's conjecture [1], most of the software faults remaining after thorough testing and years of production are Mandelbugs due to their transient manifestation. However, there are other aspects that have to be

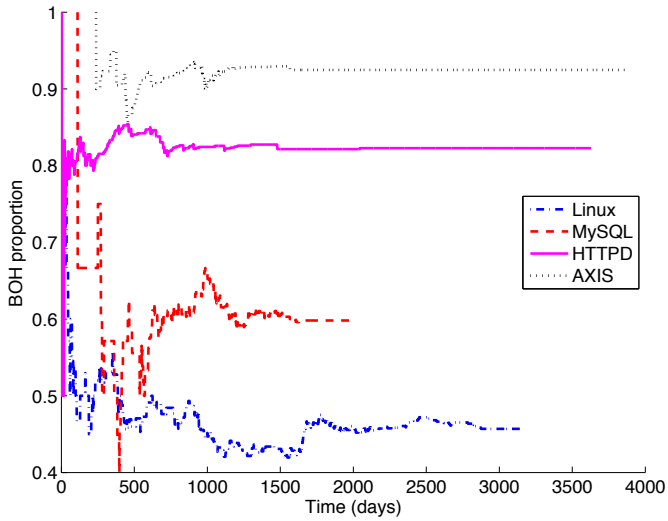

Figure 1: Proportions of BOH among classified bug reports.

taken into account to explain this result. The proportion of Bohrbugs (or Mandelbugs) is not necessarily equal to the percentage of software *failures* caused by Bohrbugs (or Mandelbugs, respectively): the failures actually experienced by the users also depend on the *operational profile*, that is, the kind of system usage that is made by its users. For this reason, our results do not contradict past empirical studies that were concerned with *failures* rather than *faults*, and that reported Mandelbugs as a major cause of software failures [17]. Note that even if Bohrbugs are easy to reproduce and to debug once detected, they are still difficult to detect in large and complex software systems. This may be due to ineffective quality assurance and testing activities, or simply due to the fact that it is impractical to extensively test such large systems. As a result, a significant number of Bohrbugs can still be found after several years. Another possible factor is that OSS in general are continuously evolving during their lifetime, since new features keep being introduced by developers. Therefore, even if Bohrbugs are detected and fixed, more Bohrbugs could be introduced when changing or extending the software.

To better understand which factor is most influential on the observed trends in bug type proportions, we analyzed the release dates of minor and major versions of the considered projects. Figure 4 shows the occurrences of minor and major releases for each project during the same time windows of Figure 1, Figure 2, and Figure 3. It can be seen that for all four projects several minor releases (e.g., Linux 2.6.31, 2.6.32, ...) occurred during the whole lifecycle. Instead, major releases occur rarely. Considering that after a major release (and even some time before it goes public) most development efforts are devoted to the new major release (e.g., new important functionalities are introduced in

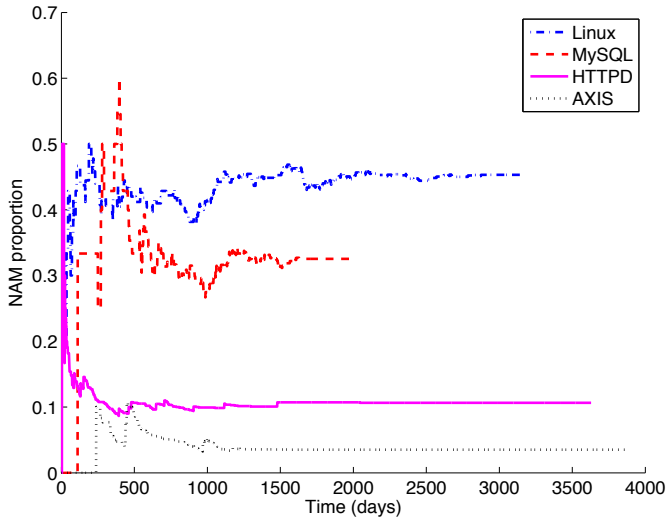

Figure 2: Proportions of NAM among classified bug reports.

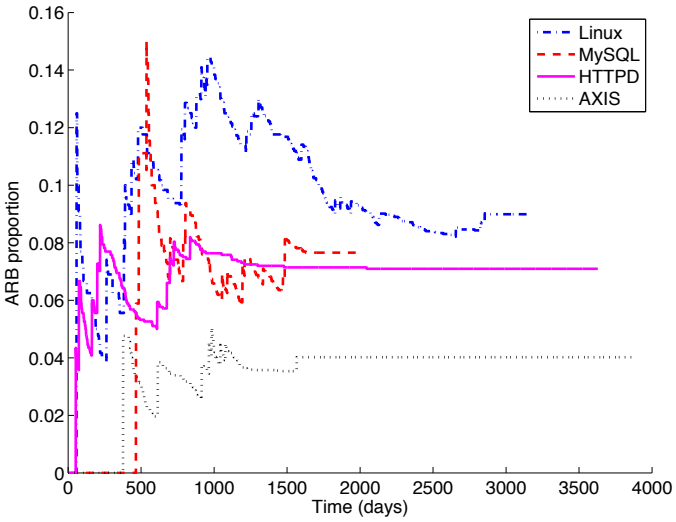

Figure 3: Proportions of ARB among classified bug reports.

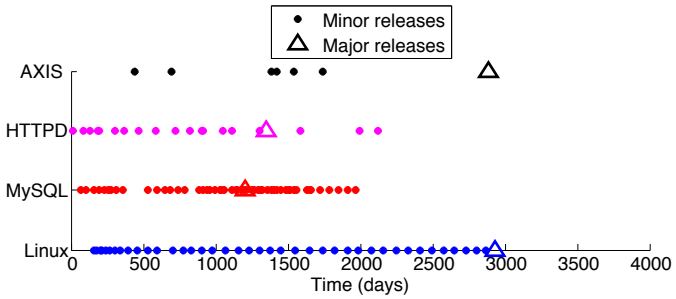

Figure 4: Minor and major release dates of considered projects.

MySQL 5.5 instead of MySQL 5.1), we can assume that the effects of code changes on bug type proportions are less significant from that time. In fact, when a major release occurs, minor releases are mostly focused on bug fixes and minor improvements, and are less likely to introduce new faults; therefore, the bug type proportions after a major release mainly reflect old bugs rather than new ones. In the case of MySQL and HTTPD, major releases occur in the middle of the lifecycle of a previous major version (e.g., the major version 5.5 of MySQL was released while MySQL 5.1 was still being updated and widespread among users), while for AXIS and Linux a major version is released after the end of the lifecycle of the previous major version (i.e., the lifecycles of two major releases do not overlap). For the MySQL and HTTPD projects, at the time of a new major release (at about 1250 days), a stabilization of bug type proportions has already occurred, and there does not seem to be an increasing trend in the proportion of NAMs; therefore, we attribute the stabilization of bug type proportions for these projects to old Bohrbugs that keep being discovered after some time rather than to new Bohrbugs introduced by late releases. For Linux and AXIS, significant changes may have occurred during their lifecycle, but the Bohrbug/Mandelbug proportions among the newly-introduced faults seem to be similar to those among the fixed faults.

In Figure 5 and Figure 6, we provide the proportions of NAM and ARB subtypes for all projects (omitting ARB/TOT, which never applied). As for Linux and MySQL, there is a predominance of timing-related bugs, which can be explained by the nature of these systems, where threads concur to access shared resources and have to be properly synchronized. Timing is a less significant problem in Apache HTTPD: although it is a multi-threaded software, there is a high degree of independence between threads, because they seldom have access to shared resources when handling HTTP requests, which renders synchronization problems much less frequent. Environment-related faults are also a significant share of NAMs: in Linux, they are often related to hardware management, while in Apache HTTPD and AXIS they are related to the network and the filesystem. Bugs exhibiting a time lag before a failure only affect Linux and MySQL, which have a tendency towards data corruption problems

that may cause failures only after these errors have propagated through the system. As for ARBs, there is a strong predominance of memory-related bugs (e.g., memory leaks). Leaks associated with storage and other logical resources were also found. Only few ARBs (a total of two bugs) were related to numerical problems, and in particular to integer overflows. This low number is probably due to the scarcity of floating point arithmetic in the considered projects, which is not used at all in the case of the Linux kernel [18].

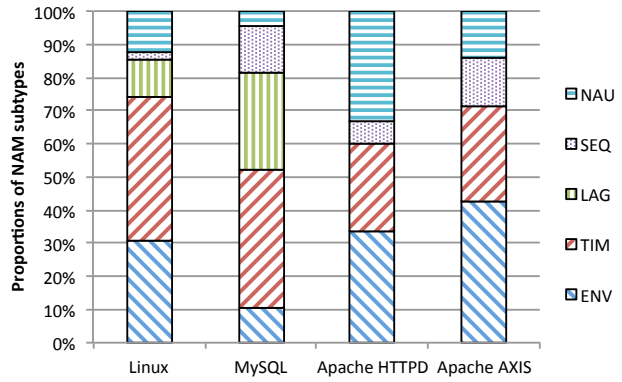

Figure 5: Proportions of NAM subtypes.

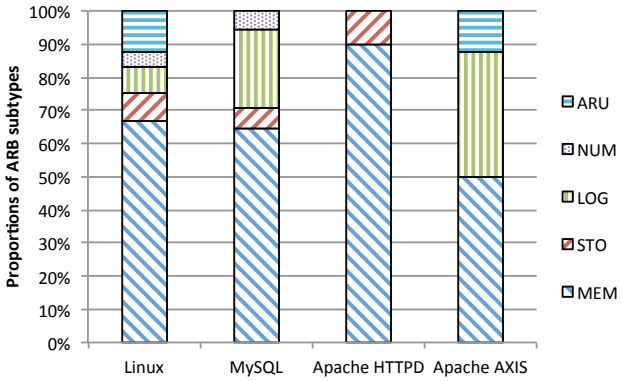

Figure 6: Proportions of ARB subtypes.

3.2 Time to fix

In order to understand the impact of bug types on the defect management process, we analyzed the time spent by developers on fixing bugs. We hypothesized

Table 5: Comparison of time to fix for bug types.

Project	Time to fix: avg. (std. dev.) BOH	NAM+ARB	Test result (adj. p-value) BOH vs. NAM+ARB
Linux	157.34 (226.21)	229.84 (304.24)	**reject** (0.0560)
MySQL	107.92 (176.76)	89.45 (109.60)	do not reject (0.2820)
HTTPD	99.09 (199.44)	116.65 (133.84)	**reject** (0.0973)
AXIS	111.19 (254.99)	186.50 (256.18)	**reject** (0.0973)

that the type of a bug has a significant impact on the time to fix, since Mandelbugs tend to be more difficult to reproduce and to diagnose than Bohrbugs. We collected from each bug report the date at which it was issued, as well as the date at which the bug was considered definitively solved by developers, and computed the difference between these two dates. This time period includes the time spent by developers on reproducing the failure reported by the users, diagnosing its root cause, developing a fix for the bug, and validating the effectiveness of the fix through testing and user feedback. It does not include the time required for users to reproduce the failure *on-site* before a bug report is filed; however, we expect that the delay for this activity is no longer for Bohrbugs than for Mandelbugs, given their transient nature, and therefore the time to fix obtained from the bug reports should not bias the comparison in favor of Mandelbugs.

We compared the time to fix for Bohrbugs and Mandelbugs by means of the *Wilcoxon rank-sum test* [19], which assesses whether one of two independent samples tends to attain larger values. Table 5 provides the average and the standard deviation of the time to fix for each class of bugs. It also shows whether the null hypothesis that for both types of bugs the time to fix is sampled from the same distribution can be rejected at a type I error level of 10%, which is the case for a p-value below 0.1. Since multiple comparisons (one per project) are being performed, using unadjusted p-values for making a test decision would lead to a probability of *at least one false rejection* that is larger than the type I error level chosen. We therefore adopted the Benjamini-Hochberg procedure [20], controlling the false rejection probability and retaining a higher power of the tests, to derive adjusted p-values. Despite this correction, which makes the tests more conservative, the null hypothesis can be rejected for three of the four projects (Linux, HTTPD, and AXIS); in each of these cases, the time to fix tends to be greater for Mandelbugs (including both NAM and ARB classes) than for Bohrbugs. One possible cause for this finding is that upon a failure caused by a Mandelbug the developer often requires additional information to understand its nature and to detect the underlying bug in the code. Moreover, Mandelbugs may be located in components that are more difficult to maintain (e.g., a problematic code area that can be dealt with only by few developers in the team). As Mandelbugs seem to be more difficult and time-consuming to cope with than Bohrbugs, strategies specifically tailored for Mandelbugs should

be useful to improve the reliability of software systems in a cost-effective way, both by means of fault tolerance mechanisms and by specific testing methods.

3.3 Bug severity

Finally, we compared the severity of bugs as perceived by users and developers, who can assign through the bug tracker system an indication of the "importance" of the bug in terms of consequences caused by it. We limited this analysis to severity because it is the only indicator of bug importance available for all the four considered projects. In every project, the severity is expressed using a severity *scale*. Since the scale is different for each project (in terms of the number and names of severity levels), the severities of the bugs of two different projects cannot be compared. We thus focus on analyzing the severity of bugs within the same project. Table 6 provides the contingency tables for bug type and bug severity, which we analyzed in order to understand whether there is a bug type that is perceived to be more severe than the other one. We adopted *Fisher's exact test of independence* [19], assessing the null hypothesis that two variables are independent. Again, p-values were adjusted using the Benjamini-Hochberg procedure [20]. The null hypothesis cannot be rejected at a reasonable type I error level for any of the projects; the percentage of bugs across severity levels does not seem to be influenced by the bug type. Therefore, we conclude that, although Bohrbugs and Mandelbugs exhibit a different behavior, there is no evidence from the considered projects that their effects in terms of failure severity are perceived to be different. This can be explained by the fact that the distinction between Bohrbugs and Mandelbugs is concerned with fault triggering (e.g., the sequence of inputs or events that make the fault affect the system state), rather than the way in which a bug manifests itself to external users as failures. Therefore, different strategies are needed for dealing with each of them.

3.4 Limitations

Although the analysis is comforted by the extensiveness of the study, accounting for more than 900 problem reports, care must be taken when interpreting the results and drawing conclusions. While the classification procedure can easily be generalized, results are limited by the chosen applications and components, by the bugs considered, and by the quality of bug reports, like for any empirical study in this field. However, using the outlined criteria for bug selection, we selected a well-defined set of bugs (i.e., the entire set of fixed bugs) from the repositories, avoiding biases related to sampling (e.g., keyword-based sampling, random sampling), which was instead adopted in past empirical studies to deal with the huge number of bug reports [2, 3, 5, 6]. Moreover, we focused our attention on widely-used and diverse projects and components (e.g, we considered both "low-level" code, such as device drivers and a storage engine, and "high-level" code, such as a web framework).

Table 6: Contingency tables for bug type and severity.

(a) Linux (outcome = do not reject, adj. p-value = 0.8276)

	BOH	**NAM+ ARB**
Blocking	9	11
High	18	30
Low	7	4
Normal	88	100

(b) MySQL (outcome = do not reject, adj. p-value = 0.8276)

	BOH	**NAM+ ARB**
Critical	28	17
Serious	41	29
Non-critical	55	36
Performance	1	2

(c) HTTPD (outcome = do not reject, adj. p-value = 0.8276)

	BOH	**NAM+ ARB**
Blocker	4	0
Critical	16	4
Major	24	7
Minor	8	0
Normal	62	14
Trivial	2	0

(d) AXIS (outcome = do not reject, adj. p-value = 0.4924)

	BOH	**NAM+ ARB**
Blocker	5	0
Closed	1	0
Critical	6	3
Major	78	6
Minor	15	0
Trivial	1	0

The analysis does not include bugs that have not yet been fixed, since their reports may contain inaccurate or incomplete information. This could bias our estimates, since unfixed bugs may have properties different from the fixed ones; for instance, Mandelbugs may tend to be fixed less frequently than Bohrbugs. However, the previous study of NASA systems [10], for which *all reports* were analyzed due to the availability of detailed failure data, showed trends similar to the ones in this study; it is thus possible that our focusing on fixed bugs has not biased the results. Also, the analysis of fixed bugs found a remarkably large (absolute) number of Bohrbugs, highlighting that Bohrbugs are a serious problem even for mature systems.

4 Discussion

The presented classification and analysis provide insights about how bugs manifest themselves during operation. These kinds of results are useful for (i) understanding the bug characteristics that make failures difficult to reproduce, and (ii) identifying the best countermeasures to cope with bugs during development, testing, and maintenance. The following findings help us on these issues.

The bug type proportions vary with the size and nature of systems. Although Mandelbugs (NAMs and ARBs) account for about 32.9% (25.7% and

7.2%, respectively) of all classified faults, which is in line with previous studies [10, 21], we noticed significant differences among systems. While the size of the software may have some influence, its kind seems to play an important role as well. In fact, non-reproducible behavior of bugs is often related to interactions of the systems with hardware and with low-level resource management. This observation is confirmed by the subsequent analysis of bug subtypes, which distinguishes between the causes of complexity; the *environment* and the *timing* of inputs and events (e.g., concurrency) represent the main subtypes of Mandelbugs, whereas the LAG class is a secondary cause. NAM/LAG bugs exhibit a long chain of events between fault activation and manifestation, which hinders systematic reproduction; they are related to coupling among system components and to code complexity. The predominance of ENV and TIM, along with the greater percentage of NAMs in Linux, suggests that Mandelbugs are more related to low-level interactions and resource management than to software size/complexity. Further analyses are needed for investigating the relationship between NAMs and software metrics. As for ARBs, results confirmed that memory-related problems are the main source of software aging, but the non-negligible percentage of ARBs connected with other system-dependent resources suggests pushing the research on software aging (today mainly focused on memory issues) to investigate other types of ARBs.

Within each project, the bug type proportions stabilize over the years. This finding contradicts the popular opinion that the prevalence of simple bugs (i.e., Bohrbugs) decreases with time, thus leading to an increase in the proportion of Mandelbugs; instead, an approximately constant proportion has been observed. The similarity of this behavior among projects (even in terms of the time to stabilization) suggests that both Mandelbugs and Bohrbugs keep being detected during the overall lifecycle of the product. This appears to be the consequence of ineffective verification activities, leaving many Bohrbugs in the code. It is, in fact, impractical to extensively test large systems. Analyzing the evolution of bug types during the lifecycle of large systems can provide feedback on the effectiveness of quality assurance and on the need for improvements. Another influential factor is the continuous evolution of open-source software, since maintenance actions, such as corrective actions and the introduction of new features, can introduce regression Bohrbugs, as well as Bohrbugs in new functionalities. This observation can be symptomatic of the need to improve maintenance activities: Bohrbugs represent a significant portion of faults and should not be neglected when operating on existing code. This means that more thorough analyses should be made to verify the effects of changes.

Mandelbugs take longer to fix, and require specific strategies to be dealt with. We found a statistically significant difference in the times to fix of Bohrbugs and Mandelbugs, respectively, for three out of four projects. In each of these cases, Mandelbugs tend to have a greater time to fix than Bohrbugs. Adopting strategies and tools for improving the diagnosis of Mandelbugs would improve the fixing time of such bugs. This is the case for the MySQL project,

in which there is no statistically significant difference in the times to fix; we attribute this result in parts to the fact that MySQL developers make extensive use of the Valgrind debugging tool for tracking down NAMs and ARBs [22]. Moreover, Mandelbugs are by their nature difficult to detect by testing, and they require more specific techniques to be found during V&V. If the number of Mandelbugs found during operation is high, there are basically two alternatives. The first one is to employ additional V&V techniques for future releases, by introducing model checking, stress testing, code reviews. The second solution is to rely on runtime failure detection [23] and recovery mechanisms [24, 14], to compensate for the longer repair time of these bugs, and avoid system downtime while developers investigate the root cause of problems. Recovery mechanisms include: *restart* of a component or a service; *reconfiguration* of components (e.g., migration to a diverse environment); *retry* operations. These strategies can be adopted depending on the system and failure type (e.g., a retry can succeed in the case of a timing bug in the software application, while a complete reboot is needed for bugs in the OS). Moreover, software aging issues can be prevented by *software rejuvenation* [8], a technique that proactively restarts a system in order to avoid the occurrence of aging failures.

The severities of bug types are perceived to be similar. The analysis of bug severity highlights that, despite the higher complexity of Mandelbugs that might endow them with more severe consequences, the failure severity assigned by developers shows no significant differences between Bohrbugs and Mandelbugs. These bug types differ in their failure reproducibility, not in their impact on the system. The relative importance of Bohrbugs and Mandelbugs during design, testing, and maintenance activities is therefore determined by their relative proportions. If there is a significant proportion of Mandelbugs, additional testing and recovery strategies are recommended, while more regression and functional testing may be needed if the proportion of Bohrbugs is high.

References

[1] J. Gray, "Why do computers stop and what can be done about it?" Tandem Computers, Tech. Rep. 85.7, 1985.

[2] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? A fault study using open-source software," in *Proc. 2000 Intl. Conf. Depend. Sys. Netwks.*, 2000, pp. 97–106.

[3] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now? an empirical study of bug characteristics in modern open source software," in *Proc. Wksp. Arch. Sys. Supp. Improv. Softw. Depend.*, 2006, pp. 25–33.

[4] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics." in *Proc. Intl. Conf. Arch. Supp. Prog. Lang. Op. Sys.*, 2008, pp. 329–339.

[5] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *Proc. Intl. Conf. Depend. Sys. Netwks.*, 2010, pp. 221–230.

[6] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *Proc. Intl. Conf. Softw. Eng.*, 2010, pp. 485–494.

[7] M. Grottke and K. S. Trivedi, "Software faults, software aging and software rejuvenation," *J. Rel. Eng. Ass. Japan*, vol. 27, no. 7, pp. 425–438, 2005.

[8] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: Analysis, module and applications," in *Proc. Intl. Symp. Fault-Tolerant Computing*, 1995, pp. 381–390.

[9] A. Pfening, S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, "Optimal software rejuvenation for tolerating soft failures," *Perf. Eval.*, vol. 27-28, pp. 491–506, 1996.

[10] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *Proc. Intl. Conf. Depend. Sys. and Netwks.*, 2010, pp. 447–456.

[11] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on.* IEEE, 2013, pp. 178–187.

[12] M. Grottke, R. Matias, and K. Trivedi, "The fundamentals of software aging," in *Proc. Wksp. Softw. Aging Rejuv.*, 2008.

[13] S. Garg, A. Van Moorsel, K. Vaidyanathan, and K. Trivedi, "A methodology for detection and estimation of software aging," in *Proc. Intl. Symp. Softw. Rel. Eng.*, 1998, pp. 283–292.

[14] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *IEEE Comp.*, vol. 40, no. 2, pp. 107–109, 2007.

[15] Wikipedia, "LAMP (software bundle)," http://en.wikipedia.org/wiki/LAMP_(software_bundle), 2013.

[16] G. Xu and A. Rountev, "Precise memory leak detection for Java software using container profiling," in *Proc. Intl. Conf. Softw. Eng.*, 2008, pp. 151–160.

[17] I. Lee and R. Iyer, "Software dependability in the Tandem GUARDIAN system," *IEEE Trans. Softw. Eng.*, vol. 21, no. 5, pp. 455–467, 1995.

[18] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley, 2010.

[19] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*. CRC Press, 1997.

[20] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *J. Roy. Stat. Soc., Ser. B (Method.)*, pp. 289–300, 1995.

[21] R. Chillarege, "Understanding Bohr-Mandel bugs through ODC triggers and a case study with empirical estimations of their field proportion," in *Proc. Wksp. Softw. Aging Rejuv.*, 2011.

[22] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[23] M. Cinque, D. Cotroneo, and A. Pecchia, "Event logs for the analysis of software failures: A rule-based approach," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 806–821, 2013.

[24] K. S. Trivedi, R. Mansharamani, D. S. Kim, M. Grottke, and M. Nambiar, "Recovery from failures due to Mandelbugs in IT systems," in *Proc. Pacific Rim Intl. Symp. Depend. Comp.*, 2011, pp. 224–233.