An Empirical Study of Fault Triggers in Linux Operating System: An Evolution Perspective

Guanping Xiao, Zheng Zheng, *Member, IEEE,* Beibei Yin, Kishor S. Trivedi, *Life Fellow, IEEE,* Xiaoting Du, and Kaiyuan Cai

***Abstract*—This paper is an empirical study of 5741 bug reports for the Linux kernel from an evolution perspective with the aim of obtaining a deep understanding of bug characteristics in the Linux operating system. A bug classification is performed based on fault triggering conditions, followed by analysis of proportions and evolution of bug types, together with their comparisons among versions, products and repair locations. In addition, an analysis of regression bugs and the relationship between bug types and the time needed to fix them are presented. Moreover, the analysis procedure of bug type characteristics based on complex network metrics is proposed, and four network metrics, i.e., degree, clustering coefficient, betweenness and closeness, are utilized to further investigate the relationship between bug types and software metrics. In this paper, 22 interesting findings based on the empirical results are revealed, and guidance based on these findings is provided for developers and users.**

***Index Terms*—bug classification, fault trigger, Linux operating system, Mandelbug, regression bug, evolution, complex network.**

NOMENCLATURE

Acronyms

OS	Operating system
BOH	Bohrbug
MAN	Mandelbug
NAM	Non-aging-related Mandelbug
ARB	Aging-related bug

Notations

k	Degree
k^{in}	In-degree
k^{out}	Out-degree
C	Clustering coefficient
C_B	Betweenness
C_C	Closeness
$lift(a_i, b_j)$	Correlation between categories a_i and b_j
nm	Network metric
$bug_{nm}^{(\cdot)}$	Network metric of a bug
$version_{nm}^{(\cdot)}$	Average network metric of bugs for a version

I. INTRODUCTION

OVER the past 25 years, the Linux operating system (OS) has been ubiquitously deployed in various fields in society. Well-supported Linux distributions are available for a wide variety of hardware platforms ranging from embedded devices and personal computers to powerful supercomputers [1]. With the evolution of the Linux OS, its functionality is continuously enhanced. For example, Linux version 1.0 was released in 1994 with approximately 17,000 lines of code, and version 4.14 was released in 2017 with more than 20 million lines of code. As the Linux OS provides operating environments to the software systems that are executed on a computer, its reliability has a direct impact on the services that are provided by the running software systems.

However, failures will inevitably manifest after the deployment of the Linux OS, as it is not cost-effective to guarantee high reliability for the Linux OS through exhaustive testing during the development period. Therefore, the activity of resolving bug reports provided by bug tracking systems (e.g., Bugzilla [2]) or static analysis tools (e.g., Coverity [3]) is a major task in the maintenance phase. Having a deep understanding of fault characteristics in the Linux OS is essential and useful for improving its reliability and thus has attracted much attention during the evolution of the Linux OS [4]–[9].

It can be expected that comprehending the factors that trigger faults and/or propagate errors could provide valuable insights into Linux OS development and maintenance phases. In 1985, Jim Gray [10] considered bug types from the bug manifestation perspective. For example, software bugs that always fail on retry, are regarded as "hard" bugs and are denoted as Bohrbugs, named after the solid and easily detected Bohr atom. In addition, software bugs with transient manifestation are considered as "soft" bug and are called Heisenbugs due to their uncertainty characteristics. For clarifying the relationships between different bug types, Grottke and Trivedi [11], [12] proposed the definitions of software fault types, in which the Bohrbug (**BOH**) can be consistently reproduced under a well-defined set of conditions. In contrast to the Bohrbug, the Mandelbug (**MAN**), a bug whose activation and/or error propagation conditions are complex, is a complementary antonym of the Bohrbug. In addition, the Mandelbug can be further categorized as a non-aging-related Mandelbug (**NAM**) and an aging-related bug (**ARB**). An aging-related bug is a type of bug that can lead to the software aging phenomenon, i.e., to an increase in the failure rate and/or performance degradation [13], [14]. According to above classification, research presented in [8] extended

This work was supported by the National Natural Science Foundation of China (Grant Nos. 61772055, 61402027 and 61572150) and in part by US NSF (Grant No. CNS-1523994). (*Corresponding author: Zheng Zheng*)

G. Xiao, Z. Zheng, B. Yin, X. Du and K. Cai are with the School of Automation Science and Electrical Engineering, Beihang University, Beijing 100191, China e-mail: (gpxiao@buaa.edu.cn; zhengz@buaa.edu.cn; yinbeibei@buaa.edu.cn; xiaoting_2015@buaa.edu.cn; kycai@buaa.edu.cn).

K. S. Trivedi is with Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708, USA e-mail: (ktrivedi@duke.edu).

TABLE I
SUMMARY OF FINDINGS RELATED TO BUG TYPE CHARACTERISTICS ANALYSIS IN LINUX OS

Findings on bug types	
#1	Among the 5741 bug reports, actual bugs account for 76.26%, and non-bugs account for 23.74%. In addition, about 75.2% of non-bugs are compile-time issues, feature requests and documentation issues.
#2	Among the 4378 actual bugs, the proportions of BOHs and MANs are 55.82% and 36.34%, respectively.
#3	The major subtypes of NAMs are TIM (37.23%), ENV (36.51%) and LAG (19.12%).
#4	The major subtype of ARBs is MEM (68.78%).
#5	The proportion of BOHs tends to grow slowly both with the evolution of versions and time, whereas the proportion of NAMs tends to decrease slowly. The proportion of ARBs tends to decrease slightly over time. The proportions of all the three types stabilize around a constant value after approximately 4000 days.
#6	The proportions of BOHs and MANs, and their evolution trends are different among versions. For all selected versions, the proportions of bug types tend to stabilize around a constant value after approximately 600 days.
#7	Driver bugs, i.e., bugs related to the products *Drivers* and *ACPI*, account for 51.57% of all classified bugs. In addition, the growth rates of the number of bugs related to the products *Drivers* and *ACPI* are faster than those of other products.
#8	A bug related to the products *Drivers*, *ACPI* or *Platform* is more likely to be a BOH; a bug in the products *File System*, *IO/Storage* or *Core* (i.e., *Memory Management*, *Process Management* and *Timers*) is more prone to be a NAM or ARB; a *Networking* bug is more likely to be a NAM.
#9	The evolution trends of bug type proportions are different among products. For example, the proportions of NAMs related to products *File System*, *IO/Storage* and *Core* (i.e., *Memory Management*, *Process Management* and *Timers*) tend to grow slightly with time, whereas the proportions of BOHs in all products tend to increase slowly. For ARBs, the proportions are prone to stabilize around a constant value after approximately 3000 days.
#10	The repair locations of most bugs are related to the *drivers* directory.
#11	A bug whose repair location is related to the *drivers* or *arch* directory is more likely to be a BOH, whereas a bug whose repair location is related to the *fs* or *core* directory (i.e., *kernel*, *mm* and *include*) is more likely to be a NAM or ARB. A bug whose patch location is related to the *net* directory is more likely to be a NAM.
Findings on regression bugs	
#12	Regression bugs account for approximately half of the classified bugs.
#13	Regression bugs possess more BOHs than non-regression bugs. In addition, a regression bug is more prone to be a BOH, whereas a non-regression bug is more likely to be a NAM or ARB.
#14	The proportion of regression bugs tends to increase with the evolution of versions and time. Moreover, the proportions of regression and non-regression bugs tend to stabilize around a constant value 0.5 after approximately 3500 days.
#15	More than half of regression bugs are caused by feature changes, including the activities of code cleanup and simplification, code conversion and refactoring, feature improvement and feature implementation, and so on.
#16	Approximately one third of regression bugs are caused by bug fixes. In addition, it is found that there are regression bug chains, since the fix for a regression bug can lead to another regression bug.
Findings on time to fix	
#17	The average time needed to fix an MAN tends to be longer than that needed to fix a BOH.
#18	The average time required to fix a regression bug tends to be shorter than that to fix a non-regression bug.
Findings on software metrics	
#19	With the evolution of clustering coefficient, a Linux with a large clustering coefficient tends to possess a low proportion of BOHs. In contrast, a Linux with a large clustering coefficient tends to have a high proportion of MANs.
#20	The characteristics of BOHs and MANs are significantly different based on the network metric degree. The sum of degrees (i.e., k^{out}, k^{in} and k), the average or maximum degree (i.e., k^{out} and k) for an MAN is significantly larger than that for a BOH.
#21	The characteristics of BOHs and MANs are not significantly different based on the network metrics clustering coefficient and betweenness.
#22	The characteristics of BOHs and MANs are significantly different based on the network metric closeness. The average or minimum closeness for a BOH is significantly larger than that for an MAN.

a more specific bug type classification for non-aging-related Mandelbugs and aging-related bugs, respectively. In addition, it was the first paper to explore the bug characteristics based on fault triggering conditions in the Linux OS.

In this paper, we perform a study of fault trigger-based bug characteristics for 5741 bug reports from the Linux kernel. This is a significant extension compared to the work presented in [8], whose data set is 346 bug reports. In addition, a further investigation of bug type characteristics from several aspects is conducted, including the analysis of proportions and evolution of bug types, the analysis of regression bugs, the analysis of relationships between bug types and fixing time, and the analysis of bug type characteristics based on network metrics. For each bug report, we carefully examine the description, comments and the attached files. The contributions of our work are the answers to the following five research questions.

RQ1: What are the proportions of bug types and how do they evolve over versions or time?

Over the past 25 years, Linux has put out more than 1300 releases ranging from versions 1.0 to 4.14. In addition, the development model of Linux also changed with evolution. For example, releases before version 2.6 were divided into stable versions and development versions. Therefore, the proportions of bug types in Linux and how they change with the evolution of versions or time, as well as the bug type proportions among versions, are warranted be explored. Moreover, the comparisons of bug type proportions among products and repair locations is also conducted in this research.

RQ2: What is the proportion of regression bugs in Linux and how does it evolve over versions or time?

The maintenance of Linux would become a difficult task with its evolution [15]. For example, regression bugs would occur. A regression bug is a bug that leads to the failure of a feature that worked normally in previous versions, due to the activities of bug fixes and/or new functionality implementation in more recent versions [16]. Therefore, the proportion of regression bugs, how it evolves over versions or time and how it impacts the evolution of bug type proportions, as well as what the causes of regression bugs are, are interesting subjects to explore.

RQ3: What is the relationship between bug types and fixing time?

The bug management process consists of several states, such as new, assigned, resolved, verified and closed [6]. The fixing time of a bug can be regarded as one measure of bug complexity. A more complex bug usually requires more time to fix. For this research question, we investigate the time spent by developers on fixing bugs for understanding the impact of bug types on the bug management process.

RQ4: Is there any software metric that can reflect the evolution of bug type proportions?

A bug in a software system means that there are faulty codes in the source codes. Thus, the relationship between the evolution of bug type proportions and software structure information is examined. In this study, we utilize complex network metrics to measure the structure information of the Linux OS. Large-scale software systems are one of the most complex man-made systems, whose interactions of fundamental compositions, for example, call graphs or class diagrams, can be abstracted as networks [17]–[19]. In our previous studies [20]–[22], we analyzed the topological and functional structures of Linux OS from a complex network perspective. This provides a research foundation for the research questions 4 and 5.

RQ5: Is there a discrepancy in bug type characteristics based on network metrics?

In this research question, we investigate the discrepancy in bug type characteristics based on complex network metrics. We label a bug and its bug type on the affected functions, which are obtained through inspecting the fixing patch. The affected functions are nodes from the corresponding Linux OS network. Thus, the network metrics of these labeled nodes can be acquired and further utilized to represent the characteristics of the bug and its bug type. The analysis procedure is elaborated in the Study Methodology section.

The contributions of this paper are summarized as 22 findings, as shown in Table I. The detailed implications of the findings are illustrated in the relevant sections of the paper. These results provide valuable insight for the developers and users of the Linux OS.

This paper extends and improves our previous work [23]. Several new analyses are conducted. For example, 1) for bug type analysis, we present detailed types in non-bugs and investigate bug type proportions among repair locations; 2) for regression bug analysis, the causes of regression bugs are further examined and discussed; 3) for fixing time analysis, the relationship between regression bugs and fixing time is presented; 4) for software metric analysis, we propose the analysis procedure of bug type characteristics based on complex network metrics and the comparison results of bug type characteristics based on network metrics are presented and discussed.

The remainder of this paper is organized as follows. Section II describes the research data, including Linux OS and the Linux bug data. Section III presents the methodology that was utilized in this study. Sections IV through VI present the answers for research questions 1 through 3, respectively. The investigations related to research questions 4 and 5 are given in Section VII. Section VIII reports the threats to validity of this study, and Section IX introduces related work. Finally, the conclusions and future work are given in Section X. Appendix A lists bug examples and their classifications. Appendix B describes the network modeling of Linux OS and the definitions of the selected network metrics. Appendix C provides detailed information for the comparison results of bug type characteristics based on network metrics.

II. Research Data

To conduct the research questions illustrated in the Introduction, we collected two types of research data, including the source code of the Linux kernel and the bug reports. The description of these data and their collection procedure are described in detail as follows.

A. Linux Operating System

The source code of Linux is obtained from the official website [24]. Linux was originally developed by Linus Torvalds in 1991. The development of Linux has gone through three stages which are classified according to the change of development models [15], [22]. The first stage includes releases from versions 1.0 to 2.5, and the second stage contains the version 2.6 series. The third stage consists of releases beginning with version 3.0. In the first stage, the version numbering is denoted as “a.b.c”, in which the first digit “a” represents the kernel version number, whereas the major and minor version numbers are denoted by the second digit “b” and the third digit “c”, respectively. In addition, the odd major version numbers correspond to development versions, whereas the even major version numbers represent stable versions. Since there was a long lag time until new functionality was introduced into stable versions, the developers decided to change development models when releasing version 2.6. In this stage, 4 digits are used to denote the releases starting with 2.6.11 [25]. The third digit indicates major versions with new functionalities,

TABLE II
DETAILS OF DATA SET

Status	Resolution	Versions	Products	Hardware	Reports	Time frame
CLOSED	CODE_FIX	2.4 – 4.9	All	All	5741	Nov. 2002 – Nov. 2016

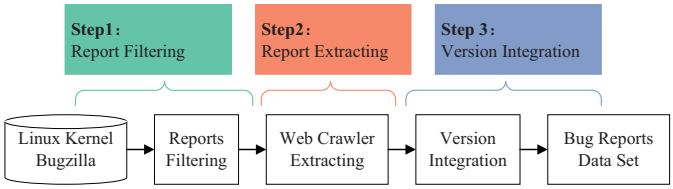


Fig. 1. Procedure for bug data collection and aggregation. Step 1: Report filtering. Step 2: Report extracting. Step 3: Version integration.

whereas the fourth number indicates minor versions with bug fixes and security patches. In 2011, for celebrating the 20th anniversary of Linux, developers stopped the old numbering method that was utilized in version 2.6 series and reused 3 digits to denote the releases since version 3.0. It is noted that starting from version 2.6.11, major versions would be released about every two or three months.

B. Linux Bug Data

With the evolution of Linux, an extensive repository of bugs has been accumulated and is publicly available. We collected the Linux bug data from Linux kernel's official bug reporting website [2]. As depicted in Fig. 1, the procedure for bug data collection and aggregation consists of three steps, i.e., report filtering, report extracting and version integration. These steps are described in detail below.

- **Step 1: Report filtering**. In this study, reports in Linux kernel Bugzilla are initially filtered based on conditions of "Status: CLOSED" and "Resolution: CODE_FIX".
- **Step 2: Report extracting**. The list of target reports is obtained after filtering, and then each report is downloaded to the local computer by a web crawler that we designed. Each report provides the following information: bug ID, summary, status, product, component, hardware, importance, kernel version, tree, regression, reported time, reporter, modified time, assignee, attachments (e.g., patch), description and comments.
- **Step 3: Version integration**. It is necessary to process the recorded versions of the collected reports for the following two reasons. First, some users used distribution versions that were based on the Linux kernel, but they also reported problems in Linux kernel Bugzilla. For example, recorded versions "2.6.6-1.414 (Fedora-devel Kernel)", "2.6.16-gentoo-r7" and "2.6.32-23-generic (ubuntu 10.04)" are not actually formal Linux released versions. In addition, some users compiled the latest source codes from Git, for example, recorded versions "2.6.21-rc5-git9", "2.6.22-rc5-git8" and "2.6.23-rc6-git2", but did not used the formal release versions. According to the version numbering method of Linux described in Section II.A, the recorded versions are integrated into the major versions. For example, recoded version "2.6.28.7" is regarded as 2.6.28, since version 2.6.28 is a major version, whereas version 2.6.28.7 is a minor version of 2.6.28.

After collection and aggregation of the bug data, we obtained 5741 bug reports, as shown in Table II. The collected data cover the mainstream tree for Linux range from versions 2.4 to 4.9 and include all targeted products and hardware platforms. The data range is for the period from November 2002 through November 2016.

III. STUDY METHODOLOGY

In this section, we first present the definitions of bug terminologies used in this paper, and describe the procedure that is conducted to classify bug types. Subsequently, we introduce the method used for correlation analysis. Lastly, an analysis procedure of bug characteristics based on network metrics is proposed.

A. Terminology

Before introducing the terminologies, it is noted that the terms **fault**, **bug** and **defect** are regarded as having the same meaning in this study. We adopt the bug type classification from [8], [11], [12]. A bug is categorized as a Bohrbug (BOH) or a Mandelbug (MAN) according to the complexity of fault triggering conditions. The definitions of the Bohrbug and Mandelbug are given as follows.

- **Bohrbug**: a bug can be consistently reproduced under a well-defined set of conditions since its activation and/or error propagation are simple.
- **Mandelbug**: a bug is difficult to reproduce since its activation and/or error propagation are complex. The complexity of the triggering conditions is attributed to the possible influence of the direct factor, for example, a time-lag between the fault activation and the failure occurrence. In addition, the complexity could also be due to the indirect factor, for example, the system-internal environment, the timing of inputs and operations, and the sequencing of inputs and operations.

Mandelbugs are separated into two subtypes, i.e., non-aging-related Mandelbugs (NAMs) and aging-related bugs (ARBs), according to whether a Mandelbug would lead to a software aging phenomenon. As depicted in Fig. 2, NAM and ARB also have subtypes. The definitions of NAM subtypes are presented as follows.

- **LAG**: there is a time lag between the activation of the bug and the manifestation of its failure;
- **ENV**: the interactions of the software application with its system-internal environment have impact on the activation and/or error propagation;

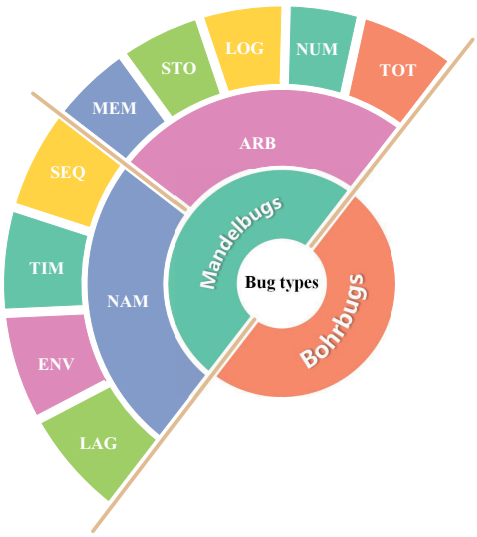


Fig. 2. Based on the complexity of fault triggering conditions, bugs are classified as Bohrbugs and Mandelbugs. Mandelbugs can be further categorized as non-aging-related Mandelbugs (NAMs) and aging-related bugs (ARBs). There are also subtypes in NAMs and ARBs.

- **TIM**: the timing of inputs and operations is the factor that impacts the fault activation and/or error propagation;
- **SEQ**: the sequencing (i.e., the relative order) of inputs and operations is the factor that impacts the activation and/or error propagation.

The definitions of ARB subtypes are as follows:

- **MEM**: the root cause of MEMs is due to the accumulation of errors because of improper memory management, such as memory leaks, buffers not being flushed;
- **STO**: the root cause of STOs is due to the accumulation of errors because of improper storage space management, such as disk space is consumed by the bug;
- **LOG**: the root cause of LOGs is a result of the leaks of other logical resources (system-dependent data structures, such as inodes or sockets that are not freed after usage);
- **NUM**: the root cause of NUMs is a result of the accumulation of numerical errors, such as integer overflows, round-off errors;
- **TOT**: the root cause of TOTs is that the fault activation or error propagation rate increases with total system runtime, but it is not induced by accumulation of internal error states.

In addition, the definitions of regression and non-regression bugs are listed below.

- **Regression bug**: a bug that causes a feature, which worked normally in previous versions, to stop working after a certain event;
- **Non-regression bug**: a bug that leads to the failure of a new feature in current versions.

B. Bug Taxonomies

The procedure of bug report classification and the determination of regression bugs are presented in the following. For a given bug report, the classification procedure is summarized in three steps, as shown in Fig. 3. Each step is described in detailed as follows.

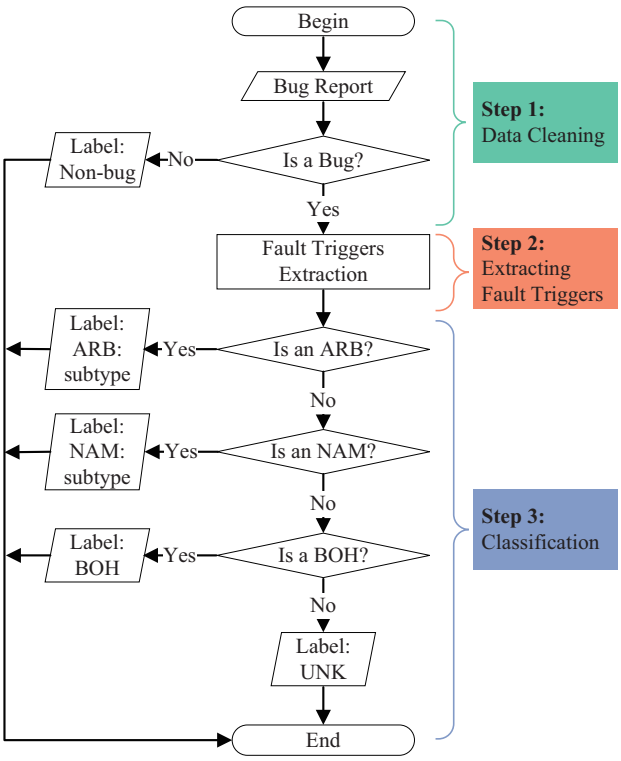


Fig. 3. Procedure of bug report classification. Step 1: Data cleaning. Step 2: Extracting fault triggers. Step 3: Classification.

- **Step 1: Data cleaning**. The bug report should be first inspected to confirm whether or not it was a bug. In this study, requests for new features or for enhancements, compile-time issues (e.g., make errors or linking errors), documentation issues (e.g., missing, outdated documentations, or harmless warning outputs), duplicates and operator errors are regarded as non-bugs and are removed from the analysis.
- **Step 2: Extracting fault triggers**. The description, discussion comments, patches, log files and other attached files of the bug report are carefully examined to determine: 1) the activation conditions, for example, the set of events and/or inputs needed to trigger errors; 2) the error propagation, for example, the parameters or states of the program that were changed by the bug and the manner that a changed parameter or state propagated; 3) the manifestation of the failure, for example, what phenomenon the users observed when failure occurred.
- **Step 3: Classification**. Finally, according to the extracted fault triggers and the bug manifestation phenomena, as well as the definitions of each subtype of ARB and NAM, the bug report is successively checked for whether it belongs to an ARB, NAM or BOH. It is noted that if a bug was marked as an ARB, but there was not enough information to determine its failure mechanics, its bug type would be labeled an **ARU**. Similarly, **NAU** is

labeled for the NAM bugs which lack the information for extracting the activation and error propagation conditions. At the end, if a report did not have sufficient information for classifying it as an ARB, NAM, or BOH, it would be labeled an unknown type (**UNK**).

In addition, the determination of a regression bug is processed according to two criteria. The first is to check the regression flag in a bug report. If a bug was reported as a regression, it would be tagged as Yes in the regression filed in its reporting page. However, it is not reliable to determine whether a bug report is a regression bug based only on its regression flag, since the regression flag was submitted by the reporter, who could misclassify it. In addition, the regression flags of some reports are blank, as the reporters did not record this information. Therefore, the second criterion is to examine the textual messages of a bug report (e.g., description and discussion comments) based on the definition of the regression bug.

All the work with respect to the bug type classification and regression bug classification is manually implemented by the authors, and when encountering suspicious classified cases, cross-checks and discussions occurred. To clarify the classification, examples of fault trigger-based bug type classification, and examples of regression bug classification, are depicted in Appendix A. This could be used as references for the classification. Furthermore, to enable other researchers to understand and implement the classification more easily, as well as for further analysis, our data have been released on our research website[1].

C. Correlation Analysis Among Bug Categories

According to the bug taxonomies, a bug can be categorized as a BOH or MAN. In addition, the bug could also be classified as a regression bug. Furthermore, through inspecting the product affected by a bug, the bug would be considered as a bug related to a specific product, for example, a driver bug. Therefore, to investigate the correlation among these categories, a statistical metric named $lift$ is utilized [26]. The $lift$ of category a_i and category b_j is defined as

$$lift(a_i, b_j) = \frac{P(a_i b_j)}{P(a_i) * P(b_j)} \tag{1}$$

where $P(a_i b_j)$ is the probability of a bug belonging to both category a_i and b_j. If $lift(a_i, b_j) > 1$, categories a_i and b_j are positively correlated; if a bug belongs to a_i, it is more possible that the bug also belongs to b_j. Symmetrically, if $lift(a_i, b_j) < 1$, if a bug belongs to a_i, it is less possible that the bug also belongs to b_j. In addition, if $lift(a_i, b_j) = 1$, the two categories a_i and b_j are not correlated.

For example, if the total number of bugs is 100, 40 of which are related to product *Drivers*, 50 of which are BOHs, and 25 of which are product *Drivers* bugs and BOHs, the correlation $lift$ between *Drivers* bugs and BOHs is calculated as follows. $P(a_i b_j)$, where category a_i represents *Drivers* bugs and category b_j denotes BOHs, is $25/100$. In addition, $P(a_i)$ is $40/100$ and $P(b_j)$ is $50/100$. Therefore, the value of the correlation $lift(a_i, b_j)$ is $(25/100)/((40/100) * (50/100)) = 1.25$. This case means that a bug belonging to product *Drivers* is more likely to also be a BOH.

[1]http://zhengzheng.buaa.edu.cn/en/pdf/linux.xlsx

D. Bug Analysis Based on Network Metrics

To measure bug characteristics using network metrics, we need to know the affected functions that are correlated to the bug. It should be noted that the affected functions are obtained from the fixing patch. For Linux bug reports, patches are usually provided as attachments or Git commit IDs. In the following, we elaborate the analysis procedure of bug type characteristics based on network metrics, i.e., **degree** k, **clustering coefficient** C, **betweenness** C_B and **closeness** C_C, and the integration methods of how we utilize these network metrics. The network modeling of the Linux OS and the definitions of selected network metrics are presented in Appendix B.

1) Analysis Procedure: The procedure for measuring characteristics of a bug based on network metrics consists of three steps. To clarify these steps, an example is given, as shown in Fig. 4.

- **Step 1: Affected function extracting**. This step is to extract the affected functions from the fixing patch of the bug. The changed statements are first inspected and then we determine the functions that contain the changed statements. These affected functions would be recorded in the table, in which the ID of the bug and its bug type, as well as the affected version, are also recorded. For example, as exhibited in Fig. 4, the changed statements of the fixing patch of bug "ID-1" are in functions func1 and func2, respectively. These functions are recorded in a table. It should be noted that a bug would be discarded if its fixing patch cannot identify which functions had been changed. For example, the fixing patch only modified data structures.
- **Step 2: Acquisition of network metrics**. Once all the affected functions are obtained, we examine these functions in the corresponding Linux OS network, and record the network metrics of the functions in a table. The network metrics considered in this work of an affected function include degree k, clustering coefficient C, betweenness C_B and closeness C_C. For example, the in-degree k^{in} of func1 is 1, whereas its out-degree k^{out} is 2, as shown in Fig. 4.
- **Step 3: Representation of bug characteristics**. After step 2, the network metrics of the affected functions of the bug have been obtained. To represent the characteristics of a bug, several integration methods (i.e., sum, average, maximum and minimum) of the network metrics are used. For example, we can use the sum of the out-degrees of the affected functions func1 and func2 as the network metric of bug "ID-1", as depicted in Fig. 4. The details of the integration methods are elaborated in the following section.

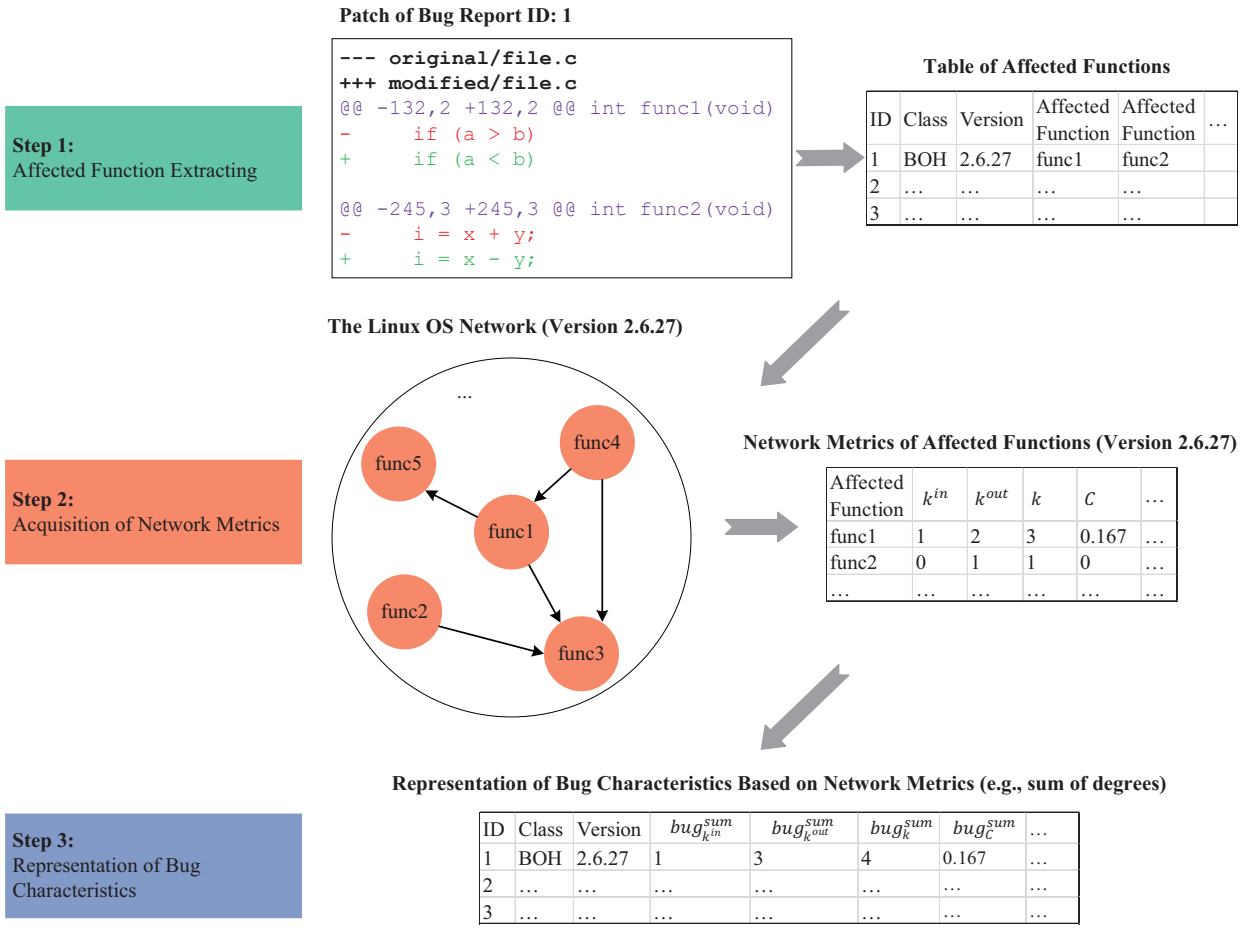


Fig. 4. Analysis procedure of bug characteristics based on network metrics. Step 1: Affected function extracting. Step 2: Acquisition of network metrics. Step 3: Representation of bug characteristics.

2) Integration Methods of Network Metrics: In this study, four integration methods of network metrics are utilized, including the **SUM**, **AVERAGE**, **MAXIMUM** and **MINIMUM**. The definitions of these integration methods are given as follows. Consider a bug, suppose that the number of affected functions, which are extracted from its patch, is q. Then, the network metric representations of the bug are denoted as:

- **SUM**: the sum of the affected functions' network metrics.

$$bug_{nm}^{sum} = \sum_{p=1}^{q} nm(p), 1 \leq p \leq q \tag{2}$$

where nm denotes a specific network metric. For example, when using clustering coefficient C, the notation of the network metric of the bug is expressed as bug_{C}^{sum}. Note that nm in the following expressions has the same meaning.

- **AVERAGE**: the average of the affected functions' network metrics.

$$bug_{nm}^{ave} = \frac{\sum_{p=1}^{q} nm(p)}{q}, 1 \leq p \leq q \tag{3}$$

- **MAXIMUM**: the maximum of the affected functions' network metrics.

$$bug_{nm}^{max} = max(nm(p)), 1 \leq p \leq q \tag{4}$$

- **MINIMUM**: the minimum of the affected functions' network metrics.

$$bug_{nm}^{min} = min(nm(p)), 1 \leq p \leq q \tag{5}$$

The average network metric of bugs for a version is further obtained by averaging the network metrics of all the bugs that belong to the version. Suppose that the number of bugs belonging to a given version is t. The average network metric of bugs for the version is calculated by the following expressions:

- **SUM**:

$$version_{nm}^{sum} = \frac{\sum_{s=1}^{t} bug_{nm}^{sum}(s)}{t}, 1 \leq s \leq t \tag{6}$$

- **AVERAGE**:

$$version_{nm}^{ave} = \frac{\sum_{s=1}^{t} bug_{nm}^{ave}(s)}{t}, 1 \leq s \leq t \tag{7}$$

- **MAXIMUM**:

$$version_{nm}^{max} = \frac{\sum_{s=1}^{t} bug_{nm}^{max}(s)}{t}, 1 \leq s \leq t \tag{8}$$

- **MINIMUM**:

$$version_{nm}^{min} = \frac{\sum_{s=1}^{t} bug_{nm}^{min}(s)}{t}, 1 \leq s \leq t \quad (9)$$

IV. Proportions and Evolution of Bug Types

In this section, we present the analytical results for **RQ1: What are the proportions of bug types and how do they evolve over versions or time?** The analysis is conducted from four aspects, including the overall proportions and evolution of bug types, comparisons of bug type proportions among versions, products and repair locations.

A. Overall Proportions and Evolution of Bug Types

Finding #1: *Among the 5741 bug reports, actual bugs account for 76.26%, and non-bugs account for 23.74%. In addition, about 75.2% of non-bugs are compile-time issues, feature requests and documentation issues.*

Fig. 5 illustrates the classification results of the collected bugs. After conducting the bug type classification, it can be observed from Fig. 5 (a) that actual bugs account for 76.26% of all collected bugs, whereas the percentage of non-bugs is 23.74%. It should be noted that in this study, as described in Section III.B, reports related to requests of features or enhancements, compile-time issues, documentation issues, duplicates and operator errors, are considered as non-bugs. Bug report triage is an important task which determines if a report is meaningful [27]. It can be observed from Fig. 5 (b) that about 75.2% of non-bugs are compile-time issues, feature requests and documentation issues. Although these reports could be unfriendly experiences to the users, it is not urgent to organize them for integration into the Linux development process, or these reports usually can be solved easily (e.g., compile-time issues and documentation issues.). A high quality of reported data could not only reduce the burdens of bug tracking system maintainers but could also benefit the studies of measurements and predictions based on the data. However, although the proportion of non-bugs is less than other open-source projects (e.g., 33.8% [28]), it still accounts for more than 20% of the collected bugs. This finding indicates that the quality of the bug data has the potential to be improved.

Implications: *To improve the quality of Linux bug reports, it is suggested that in the reporting page, there could be a custom drop-down field regarding the types of reported problems, for example, "Bug", "Feature Request", "Documentation Issue" and "Compile-time Issue". Or, the bug writing guidelines could suggest reporters to prefix the summary of the report with the words "Bug:", "Feature Request:", "Compile-time Issue:" or "Documentation Issue:". In addition, since approximately 40% of non-bugs are compile-time issues, developers should compile their source codes before releasing a version.*

Finding #2: *Among the 4378 actual bugs, the proportions of BOHs and MANs are 55.82% and 36.34%, respectively.*

The total numbers and percentages of each bug type, i.e., BOH, NAM, ARB, and UNK, are exhibited in Fig. 6 (a). The number of classified bugs is 4035, including BOHs, NAMs and ARBs, which account for 92.16% of all actual bugs. More than half of actual bugs in Linux are BOHs, as shown in Fig. 6 (a). This result indicates that in Linux, BOHs account for a large proportion, although they can be easily reproduced and debugged under a well-defined set of conditions. This phenomenon might occur due to the following two reasons. First, it is usually difficult to test a large operating system. In addition, the continuous development of Linux could lead to a high proportion of functional bugs.

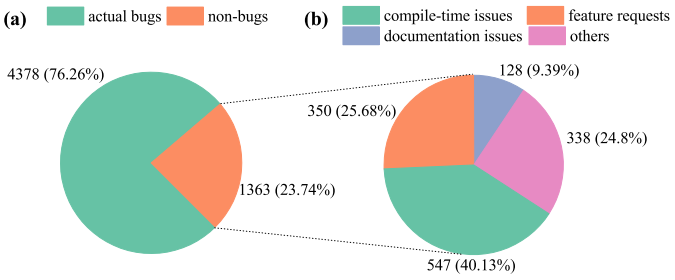


Fig. 5. Classification results of the collected bugs. (a) Total numbers and percentages for actual bugs and non-bugs. (b) Total numbers and percentages of detailed types of non-bugs.

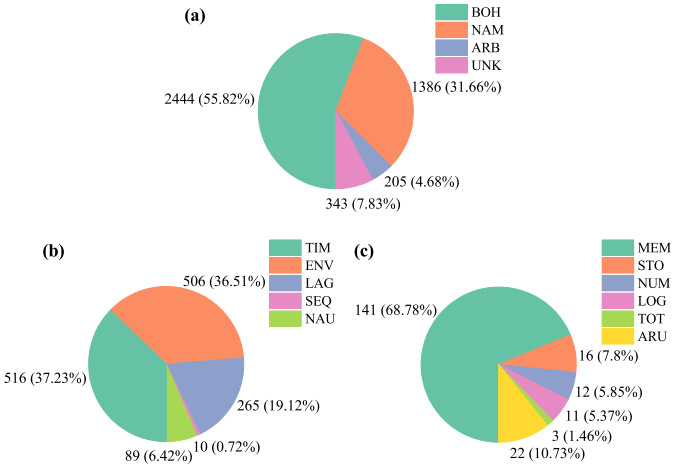


Fig. 6. Total numbers and percentages of bug types. (a) For each bug type among the 4378 actual bugs. (b) For each NAM subtype. (c) For each ARB subtype.

Moreover, MANs, including NAMs and ARBs, account for 36.34% of actual bugs. Obviously, they constitute a non-negligible portion of Linux bugs. Compared to other software systems, the proportion of MANs is close to those of MySQL (i.e., 38% [8]), space mission on-board software (i.e., 36.5% [29]) and Android OS (i.e., 31.4% [30]). Since the fault triggering conditions of MANs are more complex, specific testing methods and fault tolerance techniques should be developed to handle them.

Implications: *To mitigate BOHs, we suggest conducting sufficient testing before releasing, for example, LTP (Linux Testing Project) [31]. To mitigate MANs, a non-negligible fraction exists; we suggest developing specific testing methods, for example, combinatorial testing [32], and cost-effective fault tolerance techniques, for example, environment diversity*

[33], to conquer them.

Finding #3: *The major subtypes of NAMs are TIM (37.23%), ENV (36.51%) and LAG (19.12%).*

In addition, Fig. 6 (a) shows that the proportions of NAMs and ARBs account for 31.66% and 4.68% of the 4378 actual bugs, respectively. We further explore the proportions of subtypes of NAMs and ARBs, the two subcategories of MANs. As shown in Fig. 6 (b), TIM (37.23%), ENV (36.51%) and LAG (19.12%) are the major subtypes of NAMs. This result is in accordance with a previous study [8]. It is reasonable that TIM and ENV have high proportions due to the characteristics of an OS. The Linux OS inherently must handle concurrent activities, access shared resources and manage hardware, which would inevitably cause timing-related problems, for example, deadlock: "ID-26232: Multiple framebuffer oops and sysfs attribute deadlock" and race condition: "ID-77251: fanotify: race condition in case of error in fanotify_read", as well as environmental interaction problems, such as "ID-9111: kernel oops when unplugging usb mouse". For the subtype LAG, its root causes are usually data corruption problems or incorrect state changing problems. Once data is corrupted or a state value is incorrect, the failures would manifest after these errors propagate through the system. In Linux, the common faults of subtype LAG are null pointer dereference problems, such as "ID-10048: ipv4/fib_hash.c: fix NULL dereference".

Implications: *Since TIM, ENV and LAG are the major subtypes of NAMs, debugging, testing or fault tolerance for mitigating the impact of NAMs in Linux should focus on these bugs. More specifically, to handle TIMs, it is suggested that threads conflict or locking mechanisms of Linux should be paid more attention. To test ENVs, it should be focused on the hardware interfaces of Linux. While for LAGs, it should be examined carefully on the values of data variables or state variables, especially those which are passed in modules or subsystems.*

Finding #4: *The major subtype of ARBs is MEM (68.78%).*

The numbers and percentages of ARB subtypes are depicted in Fig. 6 (c). Subtype MEM accounts for more than two thirds (68.78%) of ARBs. The result is close to other software systems [8], [29], [30]. Linux is written in C language, in which memory management is conducted by the developers. This makes it more prone to software aging. In addition, the leaks are also associated with storage, numerical problems and other logical resources.

Implications: *We suggest that developers should pay special attention to the resource releases in Linux. Since MEM is the major subtype of ARBs, dynamic memory bug detection tools such as Kmemleak: kernel memory leak detector [34] and static code analysis tools such as Cppcheck [35], are suitable for kernel memory leaks debugging to handle the memory-related ARBs.*

In the following, we present the analysis results of bug type proportion evolution. The evolution analysis is conducted from two aspects, including evolution over versions and evolution over time. As described in Section II.B, the recorded versions were integrated into major versions. We calculate statistics of the classified bugs corresponding to their integrated versions. To ensure the validity of the analysis results, continuous

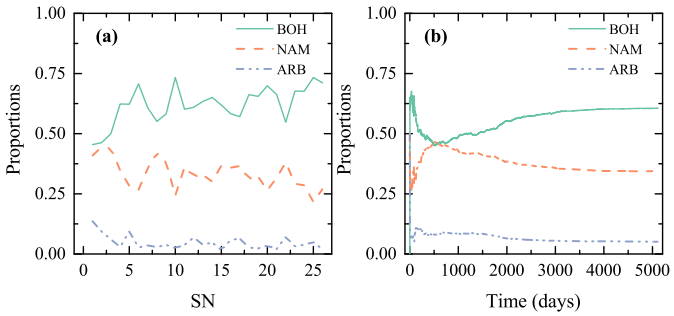


Fig. 7. Evolution of bug type proportions among classified bug reports. (a) Evolution over versions. Note that SN represents the sequential number that is assigned to each version according to their release dates, e.g., the sequence number of version 2.6.15 is 1 and that for version 3.0 is 26. The symbol will be used in the remaining parts of the paper. (b) Evolution over time.

adjacent versions (i.e., 2.6.15 to 3.0) with more than 50 bugs are chosen to analyze the evolution of bug type proportions over versions. In addition, the proportions of bug types that evolve with time are calculated. Since the life cycle of two major versions could overlap (for example, version 3.7 series was maintained from December 2012 to March 2013, whereas version 3.8 series was maintained from February 2013 to May 2013), all versions are considered in the temporal analysis. The evolution analysis results are depicted in Fig. 7.

Finding #5: *The proportion of BOHs tends to grow slowly both with the evolution of versions and time, whereas the proportion of NAMs tends to decrease slowly. The proportion of ARBs tends to decrease slightly over time. The proportions of all the three types stabilize around a constant value after approximately 4000 days.*

It is apparent in Fig. 7 that the proportion of BOHs tends to increase slowly with the evolution of versions and the time. In contrast, the proportion of NAMs tends to decrease. In addition, the proportion of ARBs tends to decrease slightly and tends to be more stable with the evolution of versions or time than BOHs and NAMs. Moreover, as shown in Fig. 7 (b), the proportions of all three types stabilize around a constant value after approximately 4000 days. It is noted that the evolution trends in Fig. 7 (a) are tested by means of Mann-Kendall trend test [36], [37]. The test results indicate that for a given criterion ($\alpha = 0.05$), the evolution trends of proportions for BOHs and NAMs are significant, whereas for ARBs it is not significant. The evolution trends of BOHs and NAMs can be explained as follows.

Approximately every two or three months, a major version of Linux is released. For example, version 4.1 was released on Jun 22, 2015, whereas version 4.2 was released on Aug 30, 2015. With the evolution of Linux, its complexity continuously grows, which is reflected by the increasing lines of code [15] and the increasing number of functions [22]. Meanwhile, a massive number of features is introduced, which might lead to more BOHs in newly released versions. Although code changes would also bring NAMs, the slow decreasing proportion of NAMs could be due to the fast growth rate of BOHs proportion.

Implications: *Due to the frequent-release characteristic of*

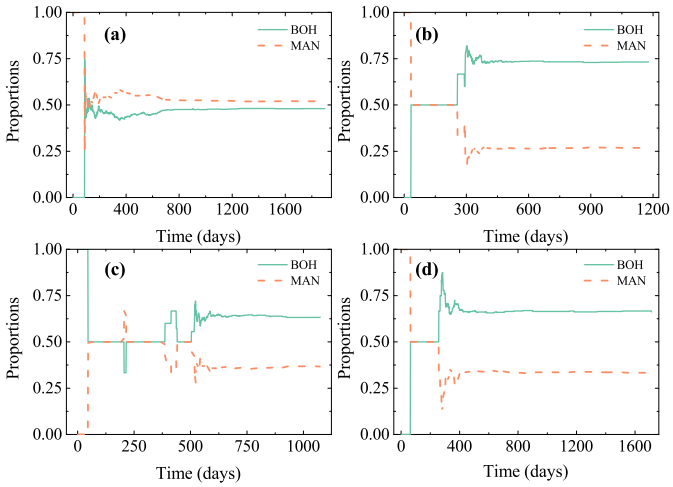


Fig. 8. Evolution of bug type proportions for four selected versions, including (a) 2.6.0, (b) 2.6.24, (c) 2.6.27 and (d) 2.6.32.

the Linux development paradigm, it is suggested that developers should pay greater attention to bugs introduced in new features and should conduct a continuous functional testing activity.

B. Comparison of Bug Types Among Versions

In this part, four versions with the most bugs are selected to explore the evolution of bug type proportions over the lifetime of a version and for comparison among versions. The analytical results for these versions, including 2.6.0, 2.6.24, 2.6.27 and 2.6.32, are exhibited in Fig. 8.

Finding #6: *The proportions of BOHs and MANs, and their evolution trends are different among versions. For all selected versions, the proportions of bug types tend to stabilize around a constant value after approximately 600 days.*

It can be observed from Fig. 8 (a) that the proportion of MANs is higher than that of BOHs in version 2.6.0, the first major version of the 2.6 series. The higher proportion of MANs in version 2.6.0 might be attributed to the implementation of a new CPU scheduler. In versions before 2.6.0, Linux used a single run-queue that represented a linked list of threads to manage all runnable tasks. However, to ensure better scalability on SMP systems, from version 2.6.0, Linux utilized a per CPU lock rather than the single run-queue lock for task management. Therefore, the kernel is preemptive from version 2.6.0, since it can respond to interactive processes immediately [38]. As developers need time to adapt to the new scheduler, this feature could lead to more NAMs, especially timing-related bugs, such as race conditions and deadlocks.

In addition, Fig. 8 (c) shows that the proportion of MANs tends to increase after about 750 days in version 2.6.27. This result may be because version 2.6.27 is one of the long-term support versions. These are special versions that are supported by the developers for a very long period. During maintenance, long-term support version could become more stable. Thus, the proportion of difficult-to-fix bugs (i.e., MANs) would increase, whereas the proportion of easily isolated and reproduced bugs (i.e., BOHs) would decrease.

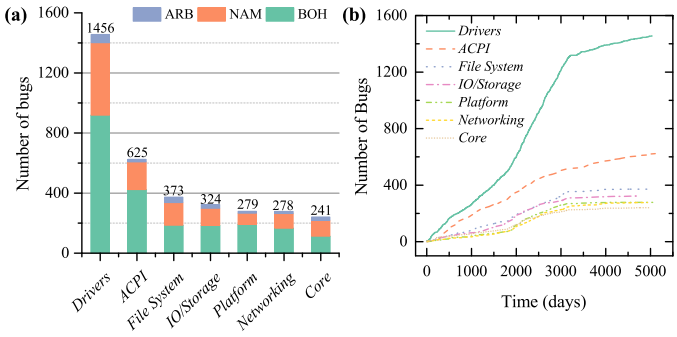


Fig. 9. Proportions and evolution of classified bugs among products. (a) Proportion of bugs, (b) evolution of bugs with time. *Core* includes bugs in 3 products: *Memory Management*, *Process Management* and *Timers*.

Implications: *The evolution trends of bug type proportions can be considered as indicators to determine which testing strategies should be mainly conducted for each version.*

C. Comparison of Bug Types Among Products

Linux consists of several functional products, such as drivers, file systems, memory management. We analyze the proportions of bug types and their temporal evolution among products to understand the impact of different products on bug types. In the Linux kernel Bugzilla, the first step for reporting a bug is to select a product (e.g., *Drivers*, *File System*, *Memory Management*, *Process Management*), to which the bug is related. We calculate statistics of bug type proportions according to the products and the evolution trends for the number of bugs, as depicted in Fig. 9. It is noted that the selected products possess the highest number of bugs. The numbers of BOHs, NAMs and ARBs in these products account for 89.32%, 87.37% and 88.78% of the total numbers of BOHs, NAMs and ARBs, respectively.

Finding #7: *Driver bugs, i.e., bugs related to the products Drivers and ACPI, account for 51.57% of all classified bugs. In addition, the growth rates of the number of bugs related to the products Drivers and ACPI are faster than those of other products.*

From Fig. 9 (a), we find that the numbers of bugs in the products *Drivers* (i.e., 1456) and *ACPI* (i.e., 625) account for 51.57% of the classified bugs (4035). In addition, the growth rates of the number of bugs related to these two products are faster than for other products, which can be observed in Fig. 9 (b). Linux supports a massive number of devices. For example, more than 100 types of devices are supported in version 4.1, and the number of functions of their source codes account for approximately 50% of the total number of functions [22]. It is noted that the product *ACPI* is short for advanced configuration and power interface, which indicates that the product is closely related to hardware devices. Since Linux supports a great diversity of devices, it is difficult to conduct compatibility testing for all of the device drivers.

Implications: *Since more than half of the classified bugs are related to Drivers, it is suggested that in Linux testing, developers should pay more attention to device drivers.*

TABLE III
CORRELATION BETWEEN BUG TYPES AND PRODUCTS

	BOH	NAM	ARB
Drivers	**1.03**	0.98	0.74
ACPI	**1.11**	0.87	0.53
File System	0.82	**1.20**	**1.90**
IO/Storage	0.94	**1.04**	**1.52**
Platform	**1.12**	0.79	0.92
Networking	0.98	**1.04**	0.99
Core	0.77	**1.30**	**1.79**

As shown in Fig. 9 (a), the proportions of bug types are different among products. To further investigate the correlation between bug types and products, we utilize metric $lift$, as described in Section III.C, to analyze the most likely product bug types. The analytical results are exhibited in Table III.

Finding #8: *A bug related to the products Drivers, ACPI or Platform is more likely to be a BOH; a bug in the products File System, IO/Storage or Core (i.e., Memory Management, Process Management and Timers) is more prone to be a NAM or ARB; a Networking bug is more likely to be a NAM.*

As presented in Table III, the $lift$ values of *Drivers*/*ACPI*/*Platform* and BOH are greater than 1, whereas the $lift$ values of *File System*/*IO Storage*/*Core* and NAM/ARB are greater than 1. In addition, the $lift$ value of *Networking* and NAM is greater than 1. The different bug type manifestations among products might be attributed to the inherent differences in products. With respect to the products *Drivers*, *ACPI* and *Platform Specific/Hardware*, although these products are considered to be bridges between an OS and devices, failures in these products would be observed by the users as more direct manifestations. If the driver of a specific device was not coded correctly, the device would not be functional. Therefore, bugs occurring in these products would be more inclined to be BOHs. Comparatively, the products *File System*, *IO/Storage*, *Networking* and *Core* (i.e., *Memory Management*, *Process Management* and *Timers*) are considered to be basic and core functions for the OS, which means that the interactions among these products tend to be more complex and tightly coupled [21]. Accordingly, bugs related to these products are more likely to be MANs.

Implications: *We suggest that different testing strategies should be selected to test different products. For example, more functional testing and compatibility testing should be taken to test product Drivers, whereas combinatorial testing [32] might be useful to test the products File System, Networking, IO/Storage, etc., since bugs related to these products are more prone to be MANs.*

Moreover, the evolution of bug type proportions among the selected products are explored, as exhibited in Fig. 10.

Finding #9: *The evolution trends of bug type proportions are different among products. For example, the proportions of NAMs related to products File System, IO/Storage and Core (i.e., Memory Management, Process Management and Timers) tend to grow slightly with time, whereas the proportions of BOHs in all products tend to increase slowly. For ARBs, the proportions are prone to stabilize around a constant value after approximately 3000 days.*

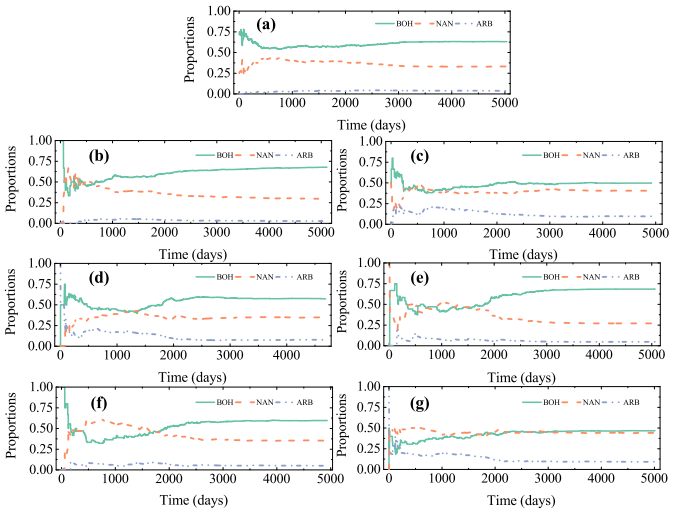


Fig. 10. Evolution of bug type proportions of selected products, including (a) *Drivers*, (b) *ACPI*, (c) *File System*, (d) *IO/Storage*, (e) *Platform*, (f) *Networking* and (g) *Core* (i.e., *Memory Management*, *Process Management* and *Timers*).

Implications: *Refer to implications for Findings #5 and #8.*

D. Comparison of Bug Types Among Repair Locations

The products mentioned in the bug reports are closely related to the directories of the Linux kernel source codes. For example, the source codes of the product *Drivers* are mainly located in the *drivers* directory. However, there could be differences between the recorded products in reports and the actual root causes of source codes, since the reporters might misjudge the products with problems. In the following, we calculate the statistics for the classified bugs that have patches. Furthermore, we examine the patches of the classified bugs to inspect and record their repair locations. For example, the code fix of "ID-18962: screen failes in kde" is located in "drivers/gpu/drm/i915/i915_gem.c", which can be obtained from the patch. Thus, the repair location of the bug is the *drivers* directory. If code fixes are related to several directories, in this circumstance, the directory with major changes is considered as the repair location.

Finding #10: *The repair locations of most bugs are related to the drivers directory.*

Finding #11: *A bug whose repair location is related to the drivers or arch directory is more likely to be a BOH, whereas a bug whose repair location is related to the fs or core directory (i.e., kernel, mm and include) is more likely to be a NAM or ARB. A bug whose patch location is related to the net directory is more likely to be a NAM.*

Statistics results for the classified bugs that have patches in their reports are depicted in Fig. 11 (a). It is notable that more than two thirds of the classified bugs have patches. It should be noted that the bugs whose patches cannot be found does not indicate that they have not been fixed, but indicates that their patches are not provided in the reports. Furthermore, we investigate the proportion of these bugs among five repair

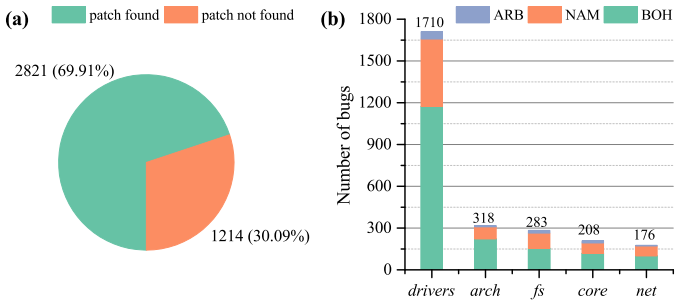


Fig. 11. Statistic results of the classified bugs that have patches in their reports. (a) Total numbers and percentages. (b) Proportion of bugs based on their repair locations. The location of *core* includes three directories, i.e., *kernel*, *mm* and *include*.

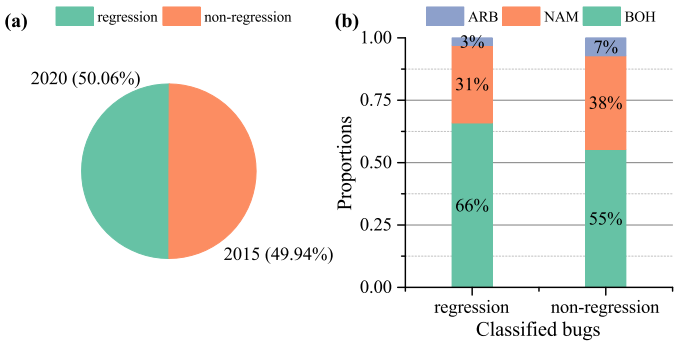


Fig. 12. Regression and non-regression bugs among classified bug reports. (a) Numbers and percentages for regression and non-regression bugs. (b) Comparison of bug types for regression and non-regression bugs.

TABLE IV
CORRELATION BETWEEN BUG TYPES AND REPAIR LOCATIONS

	BOH	NAM	ARB
drivers	**1.05**	0.92	0.82
arch	**1.06**	0.89	0.76
fs	0.83	**1.26**	**1.81**
core	0.86	**1.18**	**1.94**
net	0.87	**1.32**	0.77

TABLE V
CORRELATION BETWEEN BUG TYPES AND REGRESSION BUGS

	BOH	NAM	ARB
Regression bugs	**1.09**	0.91	0.58
Non-regression bugs	0.91	**1.09**	**1.42**

locations, i.e., *drivers*, *arch*, *fs*, *core* (*kernel*, *mm*, *include*) and *net*, as exhibited in Fig. 11 (b). It is noted that the numbers of BOHs, NAMs and ARBs in these repair locations account for 95.47%, 95.60% and 96.15%, of all BOHs, NAMs and ARBs, respectively in all repair locations. Fig. 11 (b) depicts that the *drivers* directory accounts for the most repair locations, since most bugs are related to the product *Drivers*. Similarly, a correlation between bug types and repair locations is conducted, as exhibited in Table IV. The $lift$ values of *drivers*/*arch* and BOH are greater than 1, and the $lift$ values of *fs*/*core* and NAM/ARB, are also greater than 1. In addition, the $lift$ value of *net* and NAM is greater than 1. Finding #11 provides evidence of the correlation between bug types and products from the other perspective.

Implications: *Refer to implications for Findings #7 and #8.*

V. CHARACTERISTICS OF REGRESSION BUGS

In this section, we present the results of **RQ2: What is the proportion of regression bugs in Linux and how does it evolve over versions or time?** The analytical results first present the proportion of regression bugs and their bug types, followed by the evolution analysis. Finally, the causes of regression bugs are examined.

A. Proportion of Regression Bugs

As described in Section III.A, a bug that could cause a normal feature which previously worked, to fail or misbehave completely in recent versions, is classified as a regression bug. In this section, we first calculate statistics for the numbers of regression bugs and non-regression bugs, as depicted in Fig. 12 (a). It is notable in Fig. 12 (a) that in Linux, approximately half of the classified bugs are regression bugs, i.e., existing normal feature broken problems. Compared to other software systems, it is found that the proportion of regression bugs is close to that of Google Chromium (e.g., 51.09% [39]).

Finding #12: *Regression bugs account for approximately half of the classified bugs.*

Finding #13: *Regression bugs possess more BOHs than non-regression bugs. In addition, a regression bug is more prone to be a BOH, whereas a non-regression bug is more likely to be a NAM or ARB.*

A comparison of bug type proportions between regression and non-regression bugs is performed, as shown in Fig. 12 (b). The proportion of BOHs in regression bugs is higher than that in non-regression bugs. In contrast, more MANs are observed in non-regression bugs. To further examine these correlations, we use a metric $lift$ (as described in Section III.C) to determine the bug types that regression bugs are more prone to be, as depicted in Table V. It is observed that the $lift$ value of regression bugs and BOH is greater than 1, and the $lift$ values of non-regression bugs and NAM/ARB are greater than 1. This result indicates that regression bugs would lead to more BOHs than MANs. Regression bugs are annoying to Linux OS users because when encountering serious regression bugs, users are unwilling to upgrade their OSs. As a result, although new versions could have more features or security enhancements, the systems with old OS versions would be more prone to suffer security problems.

Implications: *We suggest that developers implement more regression testing before releasing a new version to reduce the existing normal feature broken problems, since half of the bugs are related to regressions. While dealing with non-regression bugs, specific testing methods such as combinatorial testing [32] would be more effective, since a non-regression bug is more likely to be an MAN.*

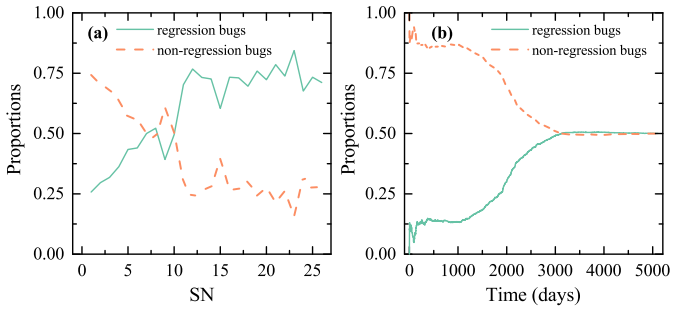


Fig. 13. Evolution of proportions of regression and non-regression bugs among classified bug reports. (a) Evolution over versions. (b) Evolution over time.

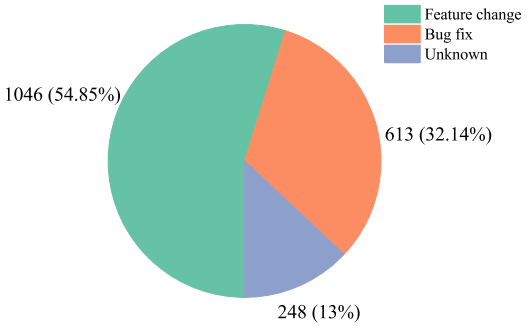


Fig. 14. Causes of regression bugs.

B. Evolution of Regression Bugs

In this part, we explore the evolution trends of proportions of regression and non-regression bugs. The evolution analysis is conducted from two aspects, i.e., evolution over versions and evolution over time, as exhibited in Fig. 13 (a) and (b), respectively.

Finding #14: *The proportion of regression bugs tends to increase with the evolution of versions and time. Moreover, the proportions of regression and non-regression bugs tend to stabilize around a constant value 0.5 after approximately 3500 days.*

With the evolution of Linux, more and more features are introduced. For example, the numbers of supported types of device drivers, architecture platforms, file systems and network protocols in version 2.4 are less than 40, 15, 40 and 30, respectively. However, in version 4.1, these numbers increase to more than 110, 25, 60 and 50, respectively. During the evolution, a massive number of code changes are implemented into Linux due to the introduction of a significant number of features. Consequently, this inevitably leads to regression bugs. In addition, the activity of bug fixes is another reason, for example, "ID-10679: pcspkr: fix dependancies breaks artsd" and "ID-11440: ipv4: sysctl fixes causes cannot open /proc/sys/net/ipv4/route/flush". Therefore, the results of Finding #14 are expected and they provide evidence for Findings #2 and #5.

Implications: *Since the proportion of regression bugs increases with evolution and they could produce more BOHs in Linux, we suggest that developers be more careful when implementing code changes, for example, feature introducing or bug fixing, and continuous regression testing should be done before releasing a new feature or fixing a bug [16].*

C. Causes of Regression Bugs

The causes of regression bugs are analyzed by manually inspecting the descriptions and comments in reports. For regression bug reports, the reporters would usually describe that the causes of bugs are due to the changes from some Git commits. In addition, since several affected versions are very close to previous versions (e.g., a feature worked normally in version 2.6.31.1, but fails in version 2.6.31.2), maintainers would ask the reporters to perform the git-bisect, a binary search, for finding the first bad commit (i.e., the first change that leading to this bug). Through examining the descriptions of these commits, we can obtain their purposes, such as bug fixing or feature changing. Since the Linux kernel used Git for tracking changes from version 2.6.12 [40], we only investigate the regression bugs with the version number starting with 2.6.12 to ensure the validity of the analytical results. As a result, there are 1907 selected regression bugs, which account for 94.4% of all regression bugs.

TABLE VI
DEVELOPMENT ACTIVITIES IN FEATURE CHANGES

Activity	Description	Example
Code optimization	Code cleanup and simplification	ID-32222
	Logic improvement	ID-12538
	Code conversion and refactoring	ID-10364
Feature improvement	Improvement of existing feature	ID-11875
Feature implementation	Implementation of new feature	ID-51881
Feature disable	Disable of existing feature	ID-14700

Finding #15: *More than half of regression bugs are caused by feature changes, including the activities of code cleanup and simplification, code conversion and refactoring, feature improvement and feature implementation, and so on.*

After performing manual inspection, we find that more than half of regression bugs are due to the activity of feature changes, as depicted in Fig. 14. It is noted that several reports lack information for determining their regression causes, and thus these bugs are labeled unknown. Moreover, with respect to the feature change, we identify four kinds of development activities as exhibited in Table VI. With the evolution of Linux, several "ancient" codes are necessary to update, i.e., code optimization, such as removing or implementing simplifications to obsolete codes. If these code optimization activities could not be handled properly, it would inevitably lead to regression bugs. In addition, the maintenance of existing features and the introduction of new features could also introduce regression bugs.

Finding #16: *Approximately one third of regression bugs are caused by bug fixes. In addition, it is found that there are regression bug chains, since the fix for a regression bug can lead to another regression bug.*

Furthermore, Fig. 14 shows that approximately one third of

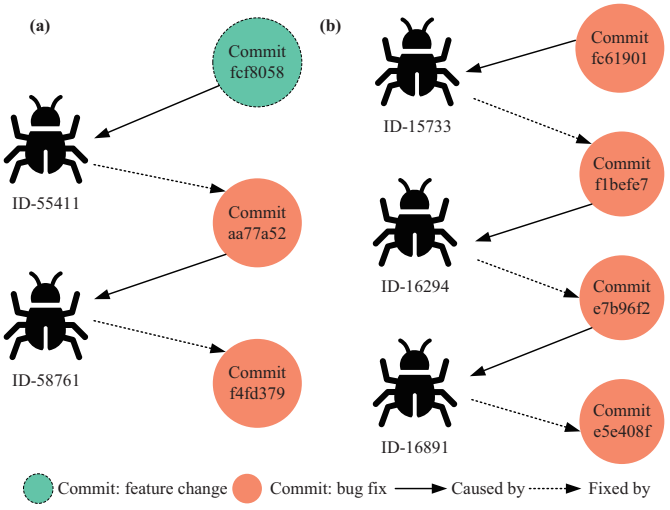


Fig. 15. Two typical examples of regression bug chains initially caused by (a) feature change or (b) bug fix.

regression bugs are introduced by bug fixes. When inspecting these regression bugs, an interesting phenomenon is observed: there are chains of regression bugs, which means that the fix for a regression bug could be a cause of another regression bug. Two typical regression bug chains are illustrated in Fig. 15. The first type of regression chain is initially caused by a feature change, as shown in Fig. 15 (a). The cause of regression bug "ID-55411" is due to "commit fcf8058: cpufreq: Simplify cpufreq_add_dev()". To fix this bug, developers provided "commit aa77a52: cpufreq: acpi-cpufreq: Don't set policy->related_cpus from .init()". However, this commit (i.e., aa77a52) led to bug "ID-58761". Finally, the bug was fixed by "commit f4fd379: acpi-cpufreq: Add new sysfs attribute freqdomain_cpus".

Comparatively, the second type of regression chain is initially attributed to a bug fix, as depicted in Fig. 15 (b). "commit fc61901: agp/intel-agp: Clear entire GTT on startup" was introduced in version 2.6.32.4 for fixing bugs, but it caused a problem reported in bug "ID-15733". Later, this bug was handled by "commit f1befe7: agp/intel: Restrict GTT mapping to valid range on i915 and i945". Unfortunately, since this fix was wrong, it resulted in a regression bug "ID-16294". Subsequently, the fix (i.e., "commit e7b96f2: agp/intel: Use the correct mask to detect i830 aperture size") also induced a regression bug "ID-16891", which was finally fixed by "commit e5e408f: intel-gtt: fix gtt_total_entries detection".

Implications: *Since more than half of regression bugs are caused by feature changes, the activities of code cleanup and simplification, code conversion and refactoring, feature improvement and feature implementation, and so on, should be carefully conducted. In addition, with respect to the regression bug chains, we suggest developing some techniques to construct the relationship between regression bugs and their causes and fixes, such as representing the relationship as networks, which could be further used to predict regression bugs. Moreover, as a developer commented in a regression bug reports: "I'd like to avoid a regression fix for a regression fix for a regression fix."*

TABLE VII
COMPARISON OF TIME TO FIX BETWEEN BOHS AND MANS

	Average fixing time (days)	Standard deviation
BOHs	218.63	360.72
MANs	254.22	374.22

VI. RELATIONSHIP BETWEEN BUG TYPE AND FIXING TIME

In this section, we present the analytical results of **RQ3: What is the relationship between bug types and fixing time?** This research question consists of two parts, i.e., the discrepancy of fixing time between BOHs and MANs and that between regression and non-regression bugs.

According to the definitions of BOH and MAN, the fault triggering conditions of an MAN are more complicated than those of a BOH. Therefore, it is expected that fixing an MAN needs a longer time. In the following, the time to fix a bug is estimated by the difference between the reported time and the last modified time, since there is no fixing time recorded in the bug reports.

Finding #17: *The average time needed to fix an MAN tends to be longer than that needed to fix a BOH.*

The average fixing time and standard deviation of BOH and MAN are presented in Table VII. It is notable that the average time taken to fix an MAN is 254.22 days, whereas fixing a BOH takes 218.63 days. We use the Wilcoxon-Mann-Whitney test [41] to further verify the result. The null hypothesis is that the fixing time for BOH and MAN is sampled according to the same distribution. For a given criteria ($\alpha = 0.05$), we obtained a p value less than 0.001 after performing the test. This result indicates that the null hypothesis can be rejected at 95% confidence. Accordingly, we conclude that the time taken to fix an MAN tends to be longer than that taken to fix a BOH, which is in accordance with previous studies regarding HTTPD [8], AXIS [8], and Android [30].

The significantly different fixing time between BOH and MAN might be attributed to the different time taken to handle them in the bug management process. The management process of a bug includes several states, such as Unconfirmed, New, Assigned, Resolve, Verified and Closed [6]. The major difference in time taken to fix a BOH and an MAN might be due to the different transition time between the states Assigned and Resolved. Due to the different complexity of fault triggering conditions between BOHs and MANs, developers usually require much time to obtain more information to detect the underlying root causes in the code to resolve an MAN. In addition, the non-deterministic characteristic of MANs could also result in taking more time to reproduce an MAN. Consequently, a longer time is necessary to fix an MAN.

Implications: *Due to the longer fixing time, specific testing methods and fault tolerance approaches would be helpful for handling MANs.*

TABLE VIII
COMPARISON OF TIME TO FIX BETWEEN REGRESSION AND NON-REGRESSION BUGS

	Average fixing time (days)	Standard deviation
Regression bugs	160.14	303.51
Non-regression bugs	305.36	407.55

Finding #18: *The average time required to fix a regression bug tends to be shorter than that to fix a non-regression bug.*

Moreover, we investigate the discrepancy of fixing time between regression and non-regression bugs, as exhibited in Table VIII. The average time taken to fix a regression bug (i.e., 160.14 days) is significantly shorter than that taken to fix a non-regression bugs (i.e., 305.36 days). Similarly, the result is verified by means of the Wilcoxon-Mann-Whitney test [41]. The null hypothesis is that the fixing time for regression and non-regression bugs is sampled from the same distribution. For a given criteria ($\alpha = 0.05$), after performing the test, we obtained a p value less than 0.001. This result indicates that the null hypothesis can be rejected at 95% confidence. Therefore, fixing a regression bug tends to require less time than that fixing a non-regression bug. The phenomenon is reasonable, since the causes of regression bugs can usually be found quickly. For most regression bugs, especially for recent regressions, reporters or developers can use git-bisect to search the first bad Git commit changes that cause the regressions. In some circumstances, solving a regression bug is done simply by reverting its initial bad changes.

Implications: *Although the average time taken to fix a regression bug tends to be shorter than that taken to fix a non-regression bug, fixing regression bugs should be done more carefully, since inappropriate regression fixes could lead to more regression bugs according to Finding #16.*

VII. BUG TYPE CHARACTERISTICS BASED ON NETWORK METRICS

In this section, we study the characteristics of bug types based on software metrics, and present the analytical results of **RQ4: Is there any software metric that can reflect the evolution of bug type proportions?** and **RQ5: Is there a discrepancy in bug type characteristics based on network metrics?**

A. Correlation Between Bug Type Proportions and Network Metrics

To determine a software metric that can be utilized to reflect the evolution of bug type proportions, we investigate the relationship between bug type proportions and a complex network metric clustering coefficient C. As described in Appendix B, the clustering coefficient is utilized to evaluate the tendency of a network to form tightly connected neighborhoods. To ensure the validity of the analytical results, we select the versions with more than 50 bugs and calculate their clustering coefficients of the entire Linux OS network [22]. Fig. 16 depicts the relationships between bug type proportions and clustering coefficients. In addition, to further analyze their

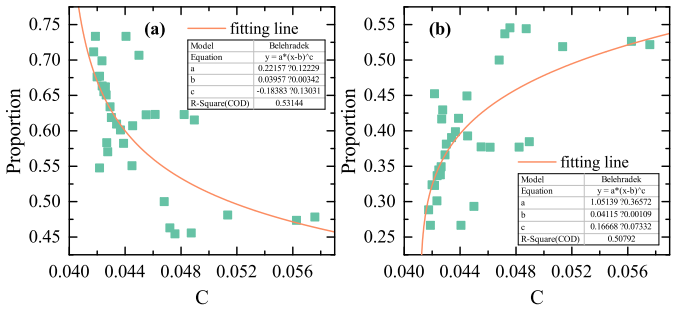


Fig. 16. The relationships between bug type proportions and clustering coefficients. (a) BOH, (b) MAN.

TABLE IX
PEARSON AND SPEARMAN CORRELATIONS BETWEEN BUG TYPE PROPORTIONS AND CLUSTERING COEFFICIENT C

	BOH - C	MAN - C	p value
Pearson corr.	-0.676	0.676	0.00002
Spearman corr.	-0.659	0.659	0.00004

relationships, the Pearson and Spearman correlation analysis techniques are conducted, as exhibited in Table IX.

Finding #19: *With the evolution of clustering coefficient, a Linux with a large clustering coefficient tends to possess a low proportion of BOHs. In contrast, a Linux with a large clustering coefficient tends to have a high proportion of MANs.*

As shown in Table IX, both Pearson and Spearman correlations are significant at the 0.01 level. It is noted that the sample size is 32. According to the results of the evolution of the Linux OS network, the clustering coefficient decreases with the evolution of versions [22]. Therefore, the strong negative correlation between the proportion of BOHs and clustering coefficient indicates that with the evolution of clustering coefficient, the proportion of BOHs tends to become higher. In contrast, due to a strong positive correlation with the clustering coefficient, the proportion of MANs tends to be lower. This phenomenon might be attributed to the fact that a large clustering coefficient indicates that there is a tight local connection of functions, which could lead to more interactions among internal functions. As a result, more complex bug manifestation (i.e., MANs) could occur. Comparatively, a small clustering coefficient refers to a loose local connection of functions, which could produce a less complex bug manifestation (i.e., BOHs).

Implications: *The clustering coefficient can be utilized as an indicator to measure the bug type proportions in future releases of a Linux.*

B. Bug Type Characteristic Analyzing Based on Network Metrics

In this part, we further analyze the characteristics of bug types based on network metrics. The affected functions of a bug are extracted from its fixing patch. According to Section IV.D, there are 2821 classified bugs that have fixing patches. After performing the analysis procedure described in Section III.D, to ensure the validity of the results, we

TABLE X
COMPARISONS OF BUG TYPE CHARACTERISTICS BASED ON NETWORK METRICS

k^{out}	$version_{k^{out}}^{sum}$	$version_{k^{out}}^{ave}$	$version_{k^{out}}^{max}$	$version_{k^{out}}^{min}$
BOH	16.24	9.29	11.28	7.70
MAN	**22.04**	**10.28**	**13.42**	7.81
p value	**0.0000**	**0.0371**	**0.0023**	0.5628

k^{in}	$version_{k^{in}}^{sum}$	$version_{k^{in}}^{ave}$	$version_{k^{in}}^{max}$	$version_{k^{in}}^{min}$
BOH	3.55	1.88	2.59	1.45
MAN	**4.96**	2.04	3.41	1.24
p value	**0.0096**	0.5841	0.0736	0.1916

k	$version_{k}^{sum}$	$version_{k}^{ave}$	$version_{k}^{max}$	$version_{k}^{min}$
BOH	19.79	11.17	13.48	9.41
MAN	**27.00**	**12.32**	**16.08**	9.47
p value	**0.0000**	**0.0402**	**0.0018**	0.6058

C	$version_{C}^{sum}$	$version_{C}^{ave}$	$version_{C}^{max}$	$version_{C}^{min}$
BOH	0.0761	0.0401	0.0548	0.0299
MAN	0.0849	0.0360	0.0562	0.0240
p value	0.3717	0.2788	0.9378	0.0685

C_B	$version_{C_B}^{sum}$	$version_{C_B}^{ave}$	$version_{C_B}^{max}$	$version_{C_B}^{min}$
BOH	0.0013	0.0006	0.0012	0.0003
MAN	0.0057	0.0022	0.0053	0.0006
p value	0.6058	0.3888	0.6279	0.6730

C_C	$version_{C_C}^{sum}$	$version_{C_C}^{ave}$	$version_{C_C}^{max}$	$version_{C_C}^{min}$
BOH	0.3435	**0.1878**	0.2376	**0.1534**
MAN	0.3681	0.1519	0.2236	0.1119
p value	0.6279	**0.0071**	0.5628	**0.0042**

identify 1359 valid bugs covering 18 versions with each version containing at least 50 bugs. In the following, we explore the discrepancy in bug type characteristics based on network metrics, including degree k, clustering coefficient C, betweenness C_B and closeness C_C. We calculate the average network metrics of bug types for each version based on four integration methods defined in Section III.D.2 (i.e., $version_{nm}^{sum}$, $version_{nm}^{ave}$, $version_{nm}^{max}$ and $version_{nm}^{min}$). The detailed numbers of network metrics of bug types for each version are provided in Appendix C. The null hypothesis is that the network metrics for BOH and MAN are sampled from the same distribution. For a given criteria ($\alpha = 0.05$), the results are tested using the Wilcoxon-Mann-Whitney test [41]. Comparisons of bug type characteristics based on network metrics are presented in Table X.

Finding #20: *The characteristics of BOHs and MANs are significantly different based on the network metric degree. The sum of degrees (i.e., k^{out}, k^{in} and k), the average or maximum degree (i.e., k^{out} and k) for an MAN is significantly larger than that for a BOH.*

Finding #21: *The characteristics of BOHs and MANs are not significantly different based on the network metrics clustering coefficient and betweenness.*

Finding #22: *The characteristics of BOHs and MANs are significantly different based on the network metric closeness. The average or minimum closeness for a BOH is significantly larger than that for an MAN.*

It is apparent in Table X that, the sums of k^{out}, k^{in} and k for an MAN are significantly larger than those for a BOH. This phenomenon indicates that the affected functions of an MAN tend to be implemented more function calls, as well as to be called by more other functions. Additionally, a comparison of k^{out} and k^{in} shows that the characteristic difference of BOHs and MANs seems to be more influenced by the k^{out}. The average or the maximum k^{out} of the affected function for an MAN is significantly larger than those for a BOH, which illustrates that the affected functions on average or the one with the most function call implementations for an MAN is more complex than that for a BOH. The discrepancy in bug type characteristics based on network metric degree is well reflected in the complexity difference between BOHs and MANs.

In addition, it is found from Table X that the characteristics of bug types based on network metrics clustering coefficient and betweenness are not significantly different. Although using the same metric, i.e., C, the analysis in this part is different from Section VII.A. The analysis in Section VII.A

is performed by considering the entire Linux OS network, whereas the statistical results of C in this part concentrate on the affected functions of a bug. The bug manifestation process is not only influenced by the affected functions, but could also be impacted by the error propagation functions. For C_B, the result indicates that we cannot state that the characteristics of bug types are significantly different based on this metric.

Furthermore, we obtain from Table X that the characteristics of bug types based on network metric closeness are significantly different. The average closeness for a BOH is significantly larger than the average for an MAN. A larger closeness of a node means that it is closer to the other nodes. Therefore, the result of closeness analysis could explain why the manifestation of BOHs is more direct and consistent than MANs.

Implications: *The characteristics of bug types can be distinguished and explained by complex network metrics, i.e., degree and closeness. These metrics can be further utilized to predict bug types or as a feature utilized for the automatic classification of bug types.*

VIII. Threats to Validity

The validity of empirical studies is naturally subject to limitations. Since the examined bugs are concentrated on the Linux kernel, we do not intend to provide any general implications about bug characteristics in all software systems. Additionally, we identify the following threats.

Selection of bug reports: In this study, the bug data exclusively concentrate on closed and fixed reports. The reason is that the bugs which have not yet been fixed may contain incomplete and inaccurate information. Bug type proportions could be influenced by considering these future closed and fixed bug reports. In addition, the bug characteristic analysis is based on the bug data from the Bugzilla of the official Linux kernel. There are several bug sources of Linux distributions, for example, Arch Linux, Gentoo Linux and Ubuntu Linux. Similar analyses conducted on these bug sources could have different bug characteristics compared to the analysis results that are based on official Linux bug data.

Manual inspection: In this study, 1) the classification of fault trigger-based bug types, 2) the determination of repair locations, 3) the classification of regression bugs, 4) the determination of the causes for regression bugs and 5) the extraction of affected functions from the corresponding fixing patches, are manually conducted by examining several details of a bug report, including report descriptions, forum comments, attached files (e.g., patches applied for correcting the bug) and external links that were attached for providing further information (e.g., Git commit IDs). The results were carefully cross-checked by two or three of the authors to reduce possible misclassification and inspection mistakes. However, as in empirical study where manual inspection is needed, possible classification mistakes and manual inspection mistakes could not be avoided in this study.

Bug type definitions: The bug types are defined based on the bug manifestation properties in terms of fault triggering conditions. The fault triggering conditions could be different for different types of software systems, for example, the environment in subtype ENV of NAM. With respect to the Linux OS, we consider external hardware devices, mountable filesystems, running applications and so on, as the environment. However, with respect to non-OS software systems, the OS would be the environment.

Evolution analysis: Several factors may influence the evolution of bugs. For version evolution analysis, a new release can propel users to migrate to the new version. Thus, bugs of previous versions would be less reported. In addition, for temporal evolution analysis, bug reporting might decrease due to fixes of the version taking more time.

Fixing time: The time to fix a bug is computed as the difference between the reported time and the closed time. In this study, we consider that the process of verification of the provided fixing patches is also a part of the fixing time. In some circumstances, the fixing time of a bug does not reflect the actual time taken to fix the bug if reporters misused the bug tracking tool. For example, a bug may already have been fixed, but the reporter forgot to confirm and close the report. Thus, the results assume an average low impact of potential misuses of the tracking tool.

Network metric analysis of bug types: The analysis of bug type characteristics based on complex network metrics relies heavily on the fixing patches, i.e., the affected functions were extracted from the fixing patches. The correctness of the fixing patches could have impact on the analysis results. In addition, the discrepancy in bug type characteristics based on network metrics could be different in other software systems. In this sense, the analysis procedure and findings should be considered as a framework to utilize complex network metrics for analyzing the manifestation characteristics of bugs, to be confirmed or rejected by further studies on other software systems or on other bug type classifications, rather than as general findings.

IX. Related Work

In this section, we first highlight the most related work about the bug characteristic analysis from the bug manifestation perspective. Then, we present several other works regarding regression bug analysis. Finally, we introduce several studies that analyze the Linux OS from the complex network perspective.

Several existing papers on defining the general characteristics of bugs, for example the IEEE Std. 1044 scheme [42], the Hewlett-Packard (HP) scheme [43], and the Orthogonal Defect Classification (ODC) [44]. ODC categorize bugs using several attributes, of which the most important is the bug type that captures the semantic of the fix conducted by the programmers and the bug trigger. The classification utilized in this study is based on the bug manifestation perspective. In 1985, Jim Gray [10] proposed a systematic abstract about the manifestation of a bug. With respect to easily reproducible bugs, he used solid or hard faults to describe them and denoted them as Bohrbugs. For the transient reproducible bugs, he described them as soft or elusive bugs, and denoted them as Heisenbugs. After that, Mandelbug is used to represent

a type of bug whose underlying causes are so complex and whose manifestation is chaotic and non-deterministic [45]. For clarifying the relationship between different definitions of bug types, Grottke and Trivedi [11], [12] proposed the detailed definitions of Bohrbugs and Mandelbugs. It was illustrated that Mandelbugs are the complementary antonyms of Bohrbugs, whereas Heisenbugs are a subset of Mandelbugs. Moreover, according to whether a Mandelbug could cause a software aging phenomenon, Mandelbugs can be further classified as aging-related bugs and non-aging-related Mandelbugs. In 2013, researchers in [8] provided a more detailed subtype classification for ARBs and NAMs according to the different kinds of complexity in fault triggering conditions.

By adopting the above classification, several studies have concentrated on bug classification and related bug characteristic analysis in different software systems [8], [29], [30]. Grottke et al. [29] explored the faults found in the on-board software for 18 JPL/NASA space missions. In this paper, 61.4% of faults were identified as Bohrbugs and 36.5% of faults were classified as Mandelbugs among the 520 faults detected in all 18 missions. Researchers in [8] investigated bugs in four open-source software systems, including Linux, MySQL, HTTPD and AXIS. They found that the proportion of Mandelbugs tends to stabilize around a constant value during the lifecycles of the four projects. Moreover, Qin et al [30] performed a fault trigger-based bug classification on the Android operating system by examining 513 bug reports. In this work, it was found that 31.4% of bugs are Mandelbugs. Other studies related to bug manifestation perspective are summarized as follows. Chandra et al. [46] investigated the faults that occurred in the Apache web server, GNOME desktop environment and MySQL database. They found that 5–14% of faults were triggered by transient conditions, such as timing and synchronization. These faults naturally fixed themselves during recovery. Researchers in [47] studied the characteristics of the bug manifestation process by defining a set of failure-exposing conditions, such as workload-dependent triggers and environment-dependent triggers. In addition, several studies concentrated on specific bugs, including ARBs [48], concurrency bugs [49].

Here, we summarize several studies on regression bugs. Nir et al. [50] found that regression bugs were usually caused by encompassed bug fixes that were included in patches. Shihab et al. [51] performed an industrial study on the risk of software changes. In their work, they found that the number of bug reports and the developer experience could be considered as the best indicators of change risk. Khattar et al. [39] conducted an in-depth characterization study of regression bugs on the Google Chromium project. One interesting finding was that 51.09% of bugs in Google Chromium are regression bugs.

Regarding the software complex network analysis, several studies have been performed. Large-scale software systems can be considered to be one of the most sophisticated man-made systems, which can be abstracted as networks [17]. An operating system is a typical software which provides execution environments to the software that runs on the system. In 2008, Zheng et al. [52] proposed two new network growth models to better describe Gentoo Linux. Gao et al. [20] modeled the kernel directory of the Linux kernel as a complex network. In their work, it was found that the robustness of the kernel network under intentional attacks, large in-degree nodes providing basic services would cause more damage to the whole system. Wang et al. [21] investigated the coupling relationships among components in the Linux OS from the perspectives of topology and function, and they further studied the impact of failures on networks. Recently, Xiao et al. [22] performed an evolution analysis of 62 major releases of the Linux OS, including versions 1.0 to 4.1, from a complex network perspective. The characteristics of the topological and functional structure evolution of the Linux OS network were revealed.

X. Conclusions and Future Work

In this paper, a comprehensive empirical study of bug characteristics within the Linux OS was performed based on 5741 bug reports from an evolution perspective. First, we present the definition of bug class, and then the steps performed for the manual inspection. The analysis was conducted from the following four aspects: bug types, regression bugs, time to fix and software metrics, along with several findings and implications that can be useful to the developers and users of the Linux OS. For example, consider Finding #6: it reveals that driver bugs account for a large proportion of the Linux bugs. In the testing of Linux, more efforts should be taken to the device drivers. In addition, findings on software metrics demonstrate that complex network metrics are useful for evaluating the characteristics of bug types.

There is abundant future work to be done in this field. We present the following directions that deserve to be pursued: 1) the automatic classification of fault trigger-based bug types. In our previous study [53], we utilized a deep learning method to automatically classify bugs with an accuracy of 0.691. We plan to improve this result by considering the characteristics of bug types based on network metrics. 2) Automatic construction of the relationships between regression bug reports and their causes and fixes. It would be interesting to develop a technique to formalize the regression relationship for further studying regression bugs. 3) Use of the network metric analysis procedure to examine the characteristics of bug types in other software systems.

Appendix A
Bug Type Examples

Examples of fault trigger-based bug type classification and examples of regression bug classification are depicted in the Table A1 and Table A2, respectively.

Appendix B
Linux OS Network

Here, we illustrate the modeling of the Linux OS network, and then provide the definitions of the selected complex network metrics which are utilized to measure the characteristics of bug types from a network perspective.

TABLE A1
SOME EXAMPLES OF CLASSIFIED BUGS

ID	Type	Description
6045	NAM/TIM	"Using the aic94xx/sas_class driver...,intermittent panic/hang on boot... due to a race condition between device discovery of the root disk and an attempt to mount the root file system"
7968	BOH	"After booting (and during booting) the keyboard LEDs (NumLock, CapsLock and ScrollLock) don't work (they're always off)."
11805	NAM/ENV	"mounting XFS produces a segfault...When there is no memory left in the system, xfs_buf_get_noaddr() can fail."
12684	NAM/LAG	"After a suspend/resume, and a second suspend, the machine refuses to resume... this could be rectified by forcibly saving and restoring the ACPI non-volatile state"
50181	ARB/MEM	"After 20 hours of uptime, memory usage starts going up..."

TABLE A2
SOME EXAMPLES OF REGRESSION BUGS AND THEIR BUG TYPES

ID	Type	Description
8736	NAM/TIM	"Here is another scenario I bumped onto - qdisc_ watchdog_cancel() and qdisc_restart() deadlock... Please try reverting commit 1936502d0... This one is a regression in 2.6.22"
11329	BOH	"in git1 and previous, cpu0_vid is reported as 1475 (which is correct). Since git2, it is reported as 725"
15699	BOH	"In 2.6.32 (and earlier), I was able to use the rt2500usb driver with my D-Link DWL-G122 wireless NIC... Now, with 2.6.34-rc3-00191-gdb217de, once I get an IP address via dhcpcd, it immediately looses the connection"
16572	NAM/LAG	"This did not occur with 2.6.33...I have not found a reliable way to trigger the panics...The entry point on NF_FORWARD did not meet the requirements of the IP stack, therefore leading to potential crashes/ panics... Reset IPCB when entering IP stack on NF_ FORWARD"
84381	BOH	"Starting with at least kernel 3.17.0rc4, the thinkpad extra buttons...are no longer functioning and remain unresponsive...The same configuration (same user space) worked flawless on a vanilla 3.15.0."

A. Network Modeling

The Linux kernel is implemented mainly based on C programming language. The functionality realization of a C language developed software system mainly depends on the function calls, which can be commonly considered as a call graph, as exhibited in Fig. A1. In this study, we define the static call graph of the Linux kernel as a directed network $G(N, E)$, where $N = \{v_1, v_2, , v_n\}$ is the set of n nodes, which represent functions in the source code of Linux kernel, and $E = \{e_1, e_2, ..., e_m\}$ is the set of m edges, each of which, $e_i = (v_s, v_t)(i = 1, 2, ..., m)$, denote the call between

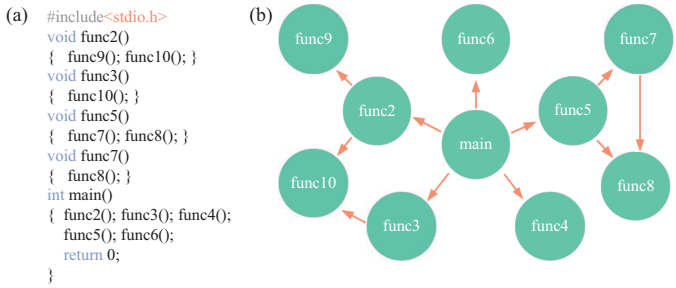


Fig. A1. A depiction for abstracting an example C language program as a directed network. (a) is an illustration of the source code of the C language program, whose static call graph that is depicted in (b) can be modeled as a directed network.

each pair of functions, i.e., nodes v_s and v_t $(v_s, v_t \in N)$. In the study, we model the selected Linux OS as directed networks and focus on the largest weakly connected part of each network.

B. Network Metrics

We choose four representative complex network metrics, in which two metrics are local properties, i.e., **degree** k and **clustering coefficient** C, and the other metrics are global properties, i.e., **betweenness** C_B and **closeness** C_C. The definitions of these network metrics are provided as follows:

- **Degree**: The degree of a node in a network is the number of edges connected to it. For a directed network, the degree of a node has two types: in-degree and out-degree. The in-degree of a node in the Linux OS network denotes the number of functions calling it, while the out-degree represents the number of functions that it calls. For example, as shown in Fig. A1, the in-degree of func5 is 1, whereas its out-degree is 2. The in-degree and out-degree of a given node i are usually denoted as k_i^{in} and k_i^{out}, respectively. In addition, the denotation of the undirected degree of the node i is k_i, which is expressed as

$$k_i = k_i^{in} + k_i^{out} \tag{A1}$$

- **Clustering Coefficient**: The clustering coefficient is utilized to measure the average probability that two neighbors of a given node are themselves neighbors. In a Linux OS network, a larger clustering coefficient of a node indicates that there are more tightly connected interactions of neighboring functions. To a directed network, the clustering coefficient of a node i is calculated as [54]

$$C_i = \frac{1}{2} \frac{\sum_j \sum_k (a_{ij} + a_{ji})(a_{ik} + a_{ki})(a_{jk} + a_{kj})}{(k_i^{in} + k_i^{out})(k_i^{in} + k_i^{out} - 1) - 2\sum_j a_{ij} a_{ji}} \tag{A2}$$

where the value of a_{ij} is 1 if there exists an edge from node i to j; otherwise, $a_{ij} = 0$. For example, as depicted in Fig. A1, the clustering coefficient of func5 is 0.167. The clustering coefficient of the entire network is denoted as

TABLE A3
DETAILED NUMBERS FOR TABLE X: k^{out}

version	$version^{sum}_{k^{out}}$		$version^{ave}_{k^{out}}$		$version^{max}_{k^{out}}$		$version^{min}_{k^{out}}$	
	BOH	MAN	BOH	MAN	BOH	MAN	BOH	MAN
2.6.0	15.25	21.87	9.25	10.96	10.69	14.33	7.94	7.97
2.6.20	20.63	27.44	9.38	8.66	11.83	14.06	7.00	4.88
2.6.23	13.09	22.82	8.92	9.44	10.55	12.18	7.45	6.88
2.6.24	18.04	20.21	9.58	12.02	11.83	14.30	7.85	9.70
2.6.25	11.40	22.94	7.26	10.20	8.60	14.85	6.25	7.74
2.6.26	15.86	26.11	9.77	13.14	11.14	16.31	8.75	10.97
2.6.27	20.91	27.03	10.59	12.32	15.56	17.86	6.38	8.10
2.6.28	13.46	20.04	7.45	10.27	9.35	12.11	6.06	8.75
2.6.29	16.08	22.17	10.49	11.35	12.31	14.50	9.16	8.58
2.6.30	18.67	24.86	10.57	9.02	12.62	12.34	8.67	6.34
2.6.31	15.98	20.51	8.99	11.96	10.78	15.88	7.55	8.90
2.6.32	13.81	25.13	8.21	8.93	10.13	12.88	6.55	5.50
2.6.33	22.31	17.72	11.65	8.78	14.81	10.00	9.89	7.67
2.6.34	11.78	20.59	8.49	11.00	9.13	13.95	8.03	9.41
2.6.35	15.23	17.50	9.80	6.05	11.57	7.82	8.77	4.50
2.6.36	16.53	17.86	8.98	10.41	11.00	12.21	7.25	8.83
2.6.37	18.48	18.95	9.34	11.10	11.20	14.38	7.78	8.48
2.6.38	14.75	23.06	8.53	9.46	9.90	11.67	7.25	7.39

TABLE A4
DETAILED NUMBERS FOR TABLE X: k^{in}

version	$version^{sum}_{k^{in}}$		$version^{ave}_{k^{in}}$		$version^{max}_{k^{in}}$		$version^{min}_{k^{in}}$	
	BOH	MAN	BOH	MAN	BOH	MAN	BOH	MAN
2.6.0	3.31	4.47	1.87	1.80	2.42	3.17	1.47	0.90
2.6.20	1.86	4.44	0.81	1.46	1.29	3.13	0.51	0.06
2.6.23	2.15	3.65	1.34	1.51	1.85	2.24	0.97	1.06
2.6.24	3.97	2.03	1.47	1.48	2.57	1.73	0.95	1.30
2.6.25	4.13	4.77	2.64	1.52	2.94	2.98	2.38	1.09
2.6.26	5.21	7.25	3.30	2.90	3.96	5.11	2.82	1.61
2.6.27	4.09	4.66	2.04	1.95	3.36	2.79	1.55	1.45
2.6.28	5.00	2.64	1.73	1.19	3.73	1.79	1.00	0.82
2.6.29	3.51	3.96	1.24	1.72	2.27	2.71	0.67	0.92
2.6.30	3.17	7.24	1.97	2.54	2.38	4.45	1.62	0.76
2.6.31	2.55	4.71	1.39	1.95	2.25	3.83	0.65	1.00
2.6.32	2.30	3.63	1.46	1.29	1.74	1.75	1.25	0.92
2.6.33	4.50	4.00	2.53	2.00	2.97	2.83	2.22	1.72
2.6.34	1.73	4.14	1.10	1.54	1.33	2.41	0.95	0.86
2.6.35	3.37	6.95	1.63	4.20	2.17	5.23	1.20	3.82
2.6.36	3.56	4.69	2.17	2.93	2.72	3.66	1.78	2.41
2.6.37	6.16	10.33	2.76	2.95	3.92	7.29	2.02	0.81
2.6.38	3.42	5.72	2.36	1.78	2.69	4.33	2.15	0.78

$$C = \frac{1}{n}\sum_{i=1}^{n} C_i \tag{A3}$$

- **Betweenness**: Betweenness is a shortest path-based metric of the centrality in a network. It measures the number of shortest paths that pass through a node. The expression of the betweenness for a node i is given as [55]

$$C_B(i) = \sum_{s\neq i\neq t} \frac{\sigma_{st}(i)}{\sigma_{st}} \tag{A4}$$

where σ_{st} is the total number of shortest paths from node s to t, while $\sigma_{st}(i)$ is the number of those paths that through node i. To a large network, the normalization of C_B can be performed as in Eq. A5. For example, to the network exhibited in Fig. A1, the betweenness of func5 is 1.

TABLE A5
DETAILED NUMBERS FOR TABLE X: k

version	$version_k^{sum}$		$version_k^{ave}$		$version_k^{max}$		$version_k^{min}$	
	BOH	MAN	BOH	MAN	BOH	MAN	BOH	MAN
2.6.0	18.56	26.33	11.11	12.77	12.83	16.20	9.64	9.47
2.6.20	22.49	31.88	10.18	10.12	12.60	16.50	7.89	5.25
2.6.23	15.24	26.47	10.26	10.96	12.21	13.41	8.55	8.71
2.6.24	22.01	22.24	11.05	13.49	14.03	15.70	9.09	11.24
2.6.25	15.52	27.70	9.90	11.72	11.37	17.11	8.79	9.09
2.6.26	21.07	33.36	13.07	16.04	14.71	20.61	11.68	12.81
2.6.27	25.00	31.69	12.63	14.27	18.15	19.72	8.25	10.03
2.6.28	18.46	22.68	9.18	11.46	12.57	13.18	7.54	9.96
2.6.29	19.59	26.13	11.73	13.06	14.20	16.58	10.00	10.13
2.6.30	21.83	32.10	12.54	11.56	14.38	15.38	10.88	7.66
2.6.31	18.53	25.22	10.37	13.91	12.75	19.24	8.37	10.12
2.6.32	16.11	28.75	9.67	10.22	11.55	14.17	8.06	6.75
2.6.33	26.81	21.72	14.18	10.79	17.31	12.00	12.39	9.61
2.6.34	13.50	24.73	9.59	12.54	10.33	15.77	9.05	10.68
2.6.35	18.60	24.45	11.43	10.25	13.57	12.77	10.09	8.55
2.6.36	20.09	22.55	11.15	13.34	13.28	14.93	9.41	11.90
2.6.37	24.64	29.29	12.10	14.04	14.58	20.67	10.16	9.81
2.6.38	18.17	28.78	10.89	11.24	12.29	15.56	9.56	8.67

TABLE A6
DETAILED NUMBERS FOR TABLE X: C

version	$version_C^{sum}$		$version_C^{ave}$		$version_C^{max}$		$version_C^{min}$	
	BOH	MAN	BOH	MAN	BOH	MAN	BOH	MAN
2.6.0	0.1308	0.1134	0.0692	0.0563	0.0888	0.0777	0.0525	0.0388
2.6.20	0.1001	0.1315	0.0452	0.0393	0.0604	0.0817	0.0320	0.0224
2.6.23	0.0565	0.0630	0.0391	0.0203	0.0471	0.0357	0.0319	0.0107
2.6.24	0.0784	0.0707	0.0419	0.0393	0.0546	0.0566	0.0333	0.0260
2.6.25	0.0662	0.0746	0.0377	0.0289	0.0507	0.0509	0.0290	0.0203
2.6.26	0.0844	0.0599	0.0358	0.0219	0.0546	0.0334	0.0223	0.0134
2.6.27	0.0854	0.1408	0.0427	0.0569	0.0612	0.0941	0.0317	0.0371
2.6.28	0.0651	0.0606	0.0354	0.0264	0.0538	0.0371	0.0236	0.0170
2.6.29	0.0541	0.1038	0.0326	0.0370	0.0422	0.0728	0.0256	0.0160
2.6.30	0.0729	0.0879	0.0387	0.0357	0.0541	0.0568	0.0277	0.0252
2.6.31	0.0524	0.0437	0.0330	0.0232	0.0412	0.0340	0.0264	0.0167
2.6.32	0.0615	0.1258	0.0371	0.0494	0.0477	0.0752	0.0293	0.0371
2.6.33	0.0748	0.0665	0.0276	0.0238	0.0498	0.0314	0.0144	0.0165
2.6.34	0.0516	0.1003	0.0351	0.0474	0.0415	0.0686	0.0307	0.0336
2.6.35	0.0638	0.0845	0.0427	0.0301	0.0521	0.0543	0.0363	0.0176
2.6.36	0.0733	0.0634	0.0384	0.0435	0.0547	0.0511	0.0256	0.0384
2.6.37	0.0994	0.0643	0.0303	0.0335	0.0513	0.0487	0.0202	0.0219
2.6.38	0.0989	0.0734	0.0601	0.0354	0.0797	0.0518	0.0456	0.0236

$$C_B^{'}(i) = \frac{C_B(i) - min(C_B)}{max(C_B) - min(C_B)} \tag{A5}$$

- **Closeness**: Closeness is another metric of centrality. It is calculated as the sum of the length of the shortest paths between a node and all the other nodes in the network. A larger closeness of a node means that it locates more central in the network and is closer to all other nodes. The definition of closeness for a given node i is [55]

$$C_C(i) = \frac{1}{\sum_{j \neq i} d(i,j)} \tag{A6}$$

where $d(i,j)$ is the distance between nodes i and j. In addition, the normalization of C_C is represented as in Eq. A7. For example, to the network depicted in Fig. A1, the closeness of func5 is 1.

$$C_C^{'}(i) = (n-1)C_C(i) \tag{A7}$$

TABLE A7
DETAILED NUMBERS FOR TABLE X: C_B

version	$version_{C_B}^{sum}$		$version_{C_B}^{ave}$		$version_{C_B}^{max}$		$version_{C_B}^{min}$	
	BOH	MAN	BOH	MAN	BOH	MAN	BOH	MAN
2.6.0	0.0009	0.0079	0.0005	0.0062	0.0007	0.0077	0.0004	0.0055
2.6.20	0.0002	0.0006	0.0001	0.0003	0.0002	0.0005	0.0001	0.0000
2.6.23	0.0004	0.0002	0.0002	0.0002	0.0004	0.0002	0.0001	0.0001
2.6.24	0.0089	0.0001	0.0029	0.0001	0.0088	0.0001	0.0001	0.0001
2.6.25	0.0006	0.0056	0.0006	0.0015	0.0006	0.0038	0.0006	0.0003
2.6.26	0.0007	0.0462	0.0006	0.0150	0.0006	0.0457	0.0005	0.0002
2.6.27	0.0011	0.0296	0.0005	0.0099	0.0011	0.0287	0.0004	0.0003
2.6.28	0.0005	0.0002	0.0002	0.0002	0.0004	0.0002	0.0001	0.0002
2.6.29	0.0002	0.0009	0.0001	0.0003	0.0002	0.0008	0.0000	0.0000
2.6.30	0.0011	0.0030	0.0008	0.0010	0.0010	0.0015	0.0006	0.0004
2.6.31	0.0016	0.0013	0.0005	0.0007	0.0016	0.0012	0.0001	0.0002
2.6.32	0.0001	0.0000	0.0001	0.0000	0.0001	0.0000	0.0000	0.0000
2.6.33	0.0007	0.0004	0.0004	0.0004	0.0006	0.0004	0.0001	0.0003
2.6.34	0.0032	0.0045	0.0024	0.0022	0.0030	0.0028	0.0019	0.0017
2.6.35	0.0004	0.0011	0.0001	0.0010	0.0004	0.0011	0.0000	0.0010
2.6.36	0.0012	0.0012	0.0011	0.0010	0.0012	0.0011	0.0011	0.0010
2.6.37	0.0004	0.0002	0.0002	0.0001	0.0003	0.0002	0.0001	0.0000
2.6.38	0.0004	0.0002	0.0001	0.0001	0.0002	0.0001	0.0000	0.0001

TABLE A8
DETAILED NUMBERS FOR TABLE X: C_C

version	$version_{C_C}^{sum}$		$version_{C_C}^{ave}$		$version_{C_C}^{max}$		$version_{C_C}^{min}$	
	BOH	MAN	BOH	MAN	BOH	MAN	BOH	MAN
2.6.0	0.3123	0.4113	0.2232	0.1679	0.2608	0.2736	0.1901	0.0973
2.6.20	0.1883	0.2674	0.1075	0.0663	0.1382	0.1365	0.0869	0.0371
2.6.23	0.2248	0.5207	0.1477	0.1665	0.1799	0.3029	0.1209	0.0884
2.6.24	0.3432	0.2286	0.1439	0.1258	0.2053	0.1648	0.1030	0.0906
2.6.25	0.2904	0.3340	0.1692	0.1227	0.1954	0.2121	0.1469	0.0929
2.6.26	0.2908	0.3344	0.1687	0.1495	0.2132	0.2432	0.1523	0.1070
2.6.27	0.3974	0.3740	0.1835	0.1678	0.2438	0.2452	0.1306	0.1397
2.6.28	0.3709	0.2287	0.2052	0.1554	0.2706	0.1581	0.1718	0.1528
2.6.29	0.2483	0.2719	0.1133	0.1101	0.1704	0.1345	0.0820	0.0858
2.6.30	0.2733	0.2839	0.1687	0.1331	0.2031	0.1848	0.1417	0.1123
2.6.31	0.2813	0.4713	0.1941	0.1897	0.2246	0.2842	0.1640	0.1267
2.6.32	0.4401	0.5201	0.2731	0.1981	0.3338	0.2933	0.2287	0.1707
2.6.33	0.4291	0.4872	0.1671	0.1820	0.2397	0.2443	0.1314	0.1540
2.6.34	0.2742	0.2678	0.2230	0.1575	0.2444	0.2151	0.2019	0.1232
2.6.35	0.4080	0.4609	0.2316	0.1640	0.2765	0.2916	0.2004	0.0916
2.6.36	0.3866	0.3427	0.2186	0.1814	0.2913	0.2170	0.1761	0.1566
2.6.37	0.6831	0.2443	0.2382	0.1312	0.3248	0.1888	0.1676	0.1058
2.6.38	0.3412	0.5758	0.2043	0.1643	0.2605	0.2336	0.1642	0.0810

APPENDIX C
DETAILED NUMBERS

The detailed numbers of network metrics of bug types for Section VII.B are provided in Table A3 through Table A8.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Wen-Bo Du from Beihang University for kindly providing valuable technical advice on this manuscript.

REFERENCES

[1] (2017) LWN Distributions List. [Online]. Available: https://lwn.net/Distributions/
[2] (2017) Kernel.org Bugzilla Main Page. [Online]. Available: https://bugzilla.kernel.org/
[3] (2017) Coverity Scan - Static Analysis. [Online]. Available: https://scan.coverity.com/
[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proc. ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, 2001, pp. 73–88.
[5] P. J. Guo and D. R. Engler, "Linux kernel developer responses to static

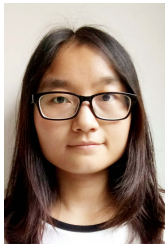
analysis bug reports." in *Proc. USENIX Annual Technical Conference*, 2009, pp. 285–292.
[6] M. F. Ahmed and S. S. Gokhale, "Linux bugs: Life cycle, resolution and architectural analysis," *Inf. Softw. Technol.*, vol. 51, no. 11, pp. 1618–1627, Nov. 2009.
[7] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: Ten years later," in *Proc. ACM SIGPLAN Notices*, vol. 46, no. 3, 2011, pp. 305–318.
[8] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE'13)*, Pasadena, USA, Nov. 2013, pp. 178–187.
[9] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empir. Softw. Eng.*, vol. 19, no. 6, pp. 1665–1705, Dec. 2014.
[10] J. Gray, "Why do computers stop and what can be done about it?" in *Proc. IEEE Symposium on Reliability in Distributed Software and Database Systems (SRDS'86)*, Los Angeles, USA, Jan. 1986, pp. 3–12.
[11] M. Grottke and K. Trivedi, "Software faults, software aging and software rejuvenation," *J. Rel. Eng. Assoc. Japan*, vol. 27, no. 7, pp. 425–438, Oct. 2005.
[12] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, no. 2, Feb. 2007.
[13] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Proc. IEEE International Symposium on Fault-Tolerant Computing (FTCS'95)*, Pasadena, USA, Jun. 1995, pp. 381–390.
[14] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *Proc. IEEE International Conference on Software Reliability Engineering Workshops (ISSREW'08)*, Seattle, USA, Nov. 2008, pp. 1–6.
[15] A. Israeli and D. G. Feitelson, "The linux kernel as a case study in software evolution," *J. Syst. Softw*, vol. 83, no. 3, pp. 485–501, Mar. 2010.
[16] J. Johnson, J. Kenefick, and P. Larson, "Hunting regressions in gcc and the linux kernel," 2004.
[17] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Phys. Rev. E*, vol. 68, no. 4, p. 046116, Oct. 2003.
[18] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "Power-laws in a large object-oriented software system," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 687–708, Oct. 2007.
[19] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 1, p. 2, Sep. 2008.
[20] Y. Gao, Z. Zheng, and F. Qin, "Analysis of linux kernel as a complex network," *Chaos Solitons Fractals*, vol. 69, pp. 246–252, Dec. 2014.
[21] H. Wang, Z. Chen, G. Xiao, and Z. Zheng, "Network of networks in linux operating system," *Physica A*, vol. 447, pp. 520–526, Apr. 2016.
[22] G. Xiao, Z. Zheng, and H. Wang, "Evolution of linux operating system network," *Physica A*, vol. 466, pp. 249–258, Jan. 2017.
[23] G. Xiao, Z. Zheng, B. Yin, K. S. Trivedi, X. Du, and K. Cai, "Experience report: Fault triggers in linux operating system: From evolution perspective," in *Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE'17)*, Toulouse, France, Oct. 2017, pp. 101–111.
[24] (2017) The Linux Kernel Archives. [Online]. Available: https://www.kernel.org/
[25] D. G. Feitelson, "Perpetual development: a model of the linux kernel life cycle," *J. Syst. Softw*, vol. 85, no. 4, pp. 859–875, Apr. 2012.
[26] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.
[27] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, p. 10, Aug. 2011.
[28] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proc. IEEE International Conference on Software Engineering (ICSE'13)*, San Francisco, USA, May 2013, pp. 392–401.
[29] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'10)*, Chicago, USA, Jul. 2010, pp. 447–456.
[30] F. Qin, Z. Zheng, X. Li, Y. Qiao, and K. S. Trivedi, "An empirical investigation of fault triggers in android operating system," in *Proc. IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'17)*, Christchurch, New Zealand, Jan. 2017, pp. 135–144.
[31] P. Larson, "Testing linux® with the linux test project," in *Proc. Ottawa Linux Symposium*, 2002, p. 265.
[32] Z. B. Ratliff, D. R. Kuhn, R. N. Kacker, Y. Lei, and K. S. Trivedi, "The relationship between software bug type and number of factors involved in failures," in *Proc. IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'16)*, Ottawa, Canada, Oct. 2016, pp. 119–124.
[33] K. S. Trivedi, M. Grottke, and E. Andrade, "Software fault mitigation and availability assurance techniques," *Int. J. Syst. Assur. Eng. Manag.*, vol. 1, no. 4, pp. 340–350, Dec. 2010.
[34] J. Corbet. (2006) Detecting kernel memory leaks [LWN.net]. [Online]. Available: https://lwn.net/Articles/187979/
[35] D. Marjamäki, "Cppcheck: a tool for static c/c++ code analysis," 2013.
[36] H. B. Mann, "Nonparametric tests against trend," *Econometrica J. Econometric Soc.*, pp. 245–259, Jul. 1945.
[37] M. G. Kendall, *Rank correlation methods.* Oxford, England: Griffin, 1948.
[38] L. A. Torrey, J. Coleman, and B. P. Miller, "A comparison of interactivity in the linux 2.6 scheduler and an mlfq scheduler," *Softw.-Pract. Exp.*, vol. 37, no. 4, pp. 347–364, Apr. 2007.
[39] M. Khattar, Y. Lamba, and A. Sureka, "Sarathi: Characterization study on regression bugs and identification of regression bug inducing changes: A case-study on google chromium project," in *Proc. ACM India Software Engineering Conference (ISEC'15)*, 2015, pp. 50–59.
[40] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the linux kernel variability model," *Software Product Lines: Going Beyond*, pp. 136–150, 2010.
[41] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.
[42] "I. S. 1044-2009, Standard Classification for Software Anomalies," 2010.
[43] R. B. Grady, *Practical software metrics for project management and process improvement*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1992.
[44] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, Nov. 1992.
[45] E. Raymond, *Ed, The New Hacker's Dictionary*. Cambridge, MA: MIT Press, 1991.
[46] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *Proc. IEEE International Conference on Dependable Systems and Networks (DSN'00)*, New York, USA, Jun. 2000, pp. 97–106.
[47] D. Cotroneo, R. Pietrantuono, S. Russo, and K. Trivedi, "How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation," *J. Syst. Softw*, vol. 113, pp. 27–43, Mar. 2016.
[48] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging analysis of the linux operating system," in *Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE'10)*, San Jose, USA, Nov. 2010, pp. 71–80.
[49] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proc. ACM Sigplan Notices*, vol. 43, no. 3, 2008, pp. 329–339.
[50] D. Nir, S. Tyszberowicz, and A. Yehudai, "Locating regression bugs," in *Proc. Springer Haifa Verification Conference*, 2007, pp. 218–234.
[51] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proc. ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'12)*, Cary, USA, Nov. 2012, p. 62.
[52] X. Zheng, D. Zeng, H. Li, and F. Wang, "Analyzing open-source software systems as complex networks," *Physica A*, vol. 387, no. 24, pp. 6190–6200, Oct. 2008.
[53] X. Du, Z. Zheng, G. Xiao, and B. Yin, "The automatic classification of fault trigger based bug report," in *Proc. IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'17)*, Toulouse, France, Oct. 2017, pp. 259–265.
[54] G. Fagiolo, "Clustering in complex directed networks," *Phys. Rev. E*, vol. 76, no. 2, p. 026107, Aug. 2007.
[55] L. C. Freeman, "Centrality in social networks conceptual clarification," *Soc. Networks*, vol. 1, no. 3, pp. 215–239, Jan. 1978.

Guanping Xiao received his B.Sc from Nanjing University of Aeronautics and Astronautics in 2012 and M.Sc. from Civil Aviation University of China in 2015. He is currently a Ph.D. candidate at Beihang University. His research interests include software reliability, and software complex networks.

Zheng Zheng received Ph.D. degree in computer software and theory in Chinese Academy of Science. In 2014 he was with Department of Electrical and Computer Engineering at Duke University, working as a research scholar. He is currently an Associate Professor at Beihang University of China. His research interests include software dependability modeling, and software fault localization.

Beibei Yin received the Ph.D. degree from Beihang University (Beijing University of Aeronautics and Astronautics), Beijing, China, in 2010. She has been a Lecturer with Beihang University since 2010. Her main research interests include software testing, software reliability, and software cybernetics.

Kishor S. Trivedi holds the Fitzgerald Hudson Chair in the Department of Electrical and Computer Engineering at Duke University. He is a Life Fellow of the Institute of Electrical and Electronics Engineers and a Golden Core Member of IEEE Computer Society. His research interests are in reliability, availability, performance and survivability of computer and communication systems and in software dependability.



Xiaoting Du received her B.Sc in Automation from Yantai University. She is currently a master student at Beihang University. Her research interests include software reliability, and software testing.

Kaiyuan Cai received the B.S., M.S., and Ph.D. degrees from Beihang University (Beijing University of Aeronautics and Astronautics), Beijing, China, in 1984, 1987, and 1991, respectively. He has been a Full Professor at Beihang University since 1995. His main research interests include software testing, software reliability, reliable flight control, and software cybernetics.