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

Guanping Xiao, *Student Member, IEEE*, Zheng Zheng, *Senior Member, IEEE*, Beibei Yin, Kishor S. Trivedi, *Life Fellow, IEEE*, Xiaoting Du, and Kai-Yuan Cai

***Abstract*—This paper presents an empirical study of 5741 bug reports for the Linux kernel from an evolutionary perspective, with the aim of obtaining a deep understanding of bug characteristics in the Linux operating system. Bug classification is performed based on the fault triggering conditions, followed by an analysis of the proportions and evolution of the bug types as well as comparisons among versions, products, and repair locations. In addition, an analysis of regression bugs and the relationship between the types of bugs and the time needed to fix them are presented. Moreover, a procedure for the analysis 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, complex network, evolution, fault trigger, Linux operating system (OS), Mandelbug (MAN), regression bug.**

NOMENCLATURE

Acronyms

OS	Operating system.
BOH	Bohrbug.
MAN	Mandelbug.
NAM	Nonaging-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.
nm	Network metric.
$\text{bug}_{\text{nm}}^{(\cdot)}$	Network metric of a bug.
$\text{version}_{\text{nm}}^{(\cdot)}$	Average network metric of bugs for a version.

Manuscript received February 3, 2018; revised June 18, 2018 and October 12, 2018; accepted May 6, 2019. Date of publication June 5, 2019; date of current version November 26, 2019. This work was supported by in part by the National Natural Science Foundation of China under Grant 61772055, in part by the Equipment Preliminary R&D Project of China under Grant 41402020102, in part by the Technical Foundation Project of Ministry of Industry and Information Technology of China under Grant JSZL2016601B003, and in part by the State Key Laboratory of Software Development Environment. The work of K. S. Trivedi was supported in part by the U.S. National Science Foundation under Grant CNS-1523994, in part by the National Natural Science Foundation of China under Grant 61872169, and in part by IBM under a Faculty Grant. Associate Editor: I. Gashi. *(Corresponding author: Zheng Zheng.)*

G. Xiao, Z. Zheng, B. Yin, X. Du, and K.-Y. 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 the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708 USA (e-mail: ktrivedi@duke.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2019.2916204

I. INTRODUCTION

OVER the past 25 years, the Linux operating system (OS) has been ubiquitously deployed in various areas. 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 has been 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. Since the Linux OS provides operating environments for 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 the highest possible reliability of the OS through exhaustive testing during the development period. Therefore, the activity of resolving bug reports provided through bug tracking systems (e.g., Bugzilla [2]) or static analysis tools (e.g., Coverity [3]) is a major task in the maintenance phase. A deep understanding of the fault characteristics in the Linux OS is thus essential and useful for improving its reliability and has consequently attracted considerable attention throughout its evolution [4]–[9].

It can be expected that comprehending the factors that trigger faults and/or propagate errors could provide valuable insights into the Linux OS development and maintenance phases. In 1985, Gray [10] considered bug types from the bug manifestation perspective. For example, software bugs that always produce failure on retry are regarded as "hard" bugs and are referred to as Bohrbugs (BOH), named after the solid and easily detected Bohr atom. By contrast, software bugs that manifest only transiently are considered "soft" bugs and are called Heisenbugs because of their uncertain characteristics. To clarify the relationships among different bug types, Grottke and Trivedi [11], [12] proposed formal definitions of software fault types as follows.

TABLE I
DEFINITIONS OF THE NAM/ARB SUBTYPES

Subtypes of **NAM** [8]	
LAG	There is a time lag between the activation of the bug and the manifestation of its consequent failure.
ENV	The interactions of the software application with its system-internal environment impact the bug's activation and/or error propagation.
TIM	The timing of inputs and operations is the main factor that impacts the fault activation and/or error propagation.
SEQ	The sequencing (i.e., the relative order) of inputs and operations is the main factor that impacts the fault activation and/or error propagation.
Subtypes of **ARB** [8]	
MEM	The root cause of MEMs is the accumulation of errors due to improper memory management, such as memory leaks or failure to flush buffers.
STO	The root cause of STOs is the accumulation of errors due to improper storage space management, such as excessive consumption of disk space by the bug.
LOG	The root cause of LOGs is leaks of other logical resources (system-dependent data structures, such as inodes or sockets that are not freed after use).
NUM	The root cause of NUMs is the accumulation of numerical errors, such as integer overflows or round-off errors.
TOT	The root cause of TOTs is a fault activation or error propagation rate that increases with the total system run time but is not induced by an accumulation of internal error states.

A BOH can be consistently reproduced under a well-defined set of conditions. In contrast, the term Mandelbug (MAN) refers to a bug whose activation and/or error propagation conditions are complex and thus is a complementary antonym of the term BOH. MANs can be further categorized into nonaging-related Mandelbugs (NAMs) and aging-related bugs (ARBs). The ARB is a type of bugs that can lead to the software aging phenomenon, i.e., to an increase in the failure rate and/or performance degradation over time [13], [14]. Based on the above classification, Cotroneo *et al.* [8] presented an extension to a more specific bug type classification for NAMs and ARBs. The definitions of NAM/ARB subtypes are presented in Table I. This was also the first paper to explore bug characteristics based on the fault triggering conditions in the Linux OS.

In the current paper, we present a study of fault-trigger-based bug characteristics for 5741 bug reports from the Linux kernel. This is a significant extension relative to the work presented in [8], which considered a data set of 346 bug reports. In addition, a further investigation of bug type characteristics is conducted from several perspectives, including an analysis of the proportions and evolution of the bug types, an analysis of regression bugs, an analysis of the relationship between bug type and fixing time, and an analysis of bug type characteristics based on network metrics. For each bug report, we carefully examined its description, the associated comments, and the attached files. The contributions of this paper can be summarized as the answers to the following five research questions.

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

Over the past 25 years, Linux has put out more than 1300 releases, from version 1.0 to 4.14. In addition, the development model of Linux has evolved. For example, releases before version 2.6 were divided into stable versions and development versions. Therefore, it is warranted to explore the proportions of bug types in Linux and how they change with the evolution of versions or with time as well as the variation in bug type proportions among versions. Moreover, comparisons of bug type proportions among products and repair locations are also conducted in this paper. The results of analyzing bug type proportions can be used for a model-based analysis [15], which can assist in the selection of appropriate complexity scenarios and parameter values. In addition, comparisons of bug types among products and repair locations can help us to better understand the relation between the distribution of bugs and the different Linux subsystems. The results can guide developers in applying specific testing strategies for different subsystems.

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

The maintenance of Linux becomes an increasingly difficult task as it evolves [16]. For example, regression bugs can occur. A regression bug is a bug that leads to the failure of a feature that worked normally in previous versions due to bug fixes and/or the implementation of new functionalities in more recent versions [17]. Therefore, it is interesting to explore the causes of regression bugs as well as the proportion of regression bugs, how it evolves over versions or time and how it impacts the evolution of the bug type proportions. The findings regarding the proportion of regression bugs can be utilized to interpret the quality of software changes. Developers can assess the quality of software changes by comparing the proportions of regression bugs counted in different versions or periods of development.

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

The bug management process proceeds through several states, including new, assigned, and resolved [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. To address this research question, we investigate the time spent by developers on fixing bugs to understand the impact of different bug types on the bug management process. Such an understanding can guide developers in applying appropriate testing strategies for different types of bugs.

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

A bug in a software system means that there is faulty code in the source code. Thus, the relationship between the evolution of the bug type proportions and the software structure information is examined. In this paper, we utilize complex network metrics to measure the structure information of the Linux OS. Large-scale software systems are among the most complex man-made systems, and the interactions among their fundamental components, such as those expressed by call graphs or class diagrams, can be abstracted as networks [18]–[20]. In our previous studies [21]–[23], we analyzed the topological and functional structures

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

Findings on bug types	
#1	Among the 5741 bug reports, actual bugs account for 76.3%, and nonbugs account for 23.7%. Of the nonbugs, approximately 75.2% of nonbugs are compile-time issues, feature requests or documentation issues.
#2	Among the 4378 actual bugs, the proportions of BOHs and MANs are 55.8% and 36.4%, respectively.
#3	NAMs account for 87.1% of MANs. The major subtypes of NAMs are TIM (37.3%), ENV (36.5%) and LAG (19.1%).
#4	ARBs account for 12.9% of MANs. The major subtype of ARBs is MEM (68.8%).
#5	The proportion of BOHs tends to grow slowly both with the evolution of versions and with time, whereas the proportion of NAMs tends to decrease slowly. The proportion of ARBs tends to decrease slightly over time. The proportions of all three types stabilize around constant values after approximately 4000 days.
#6	The proportion of MANs and its evolutionary trend are different among versions. For all selected versions, the proportions tend to stabilize around constant values over time since eventually, few new bugs will be reported.
#7	Driver bugs, i.e., bugs related to the products *Drivers* and *ACPI*, account for 51.6% of all classified bugs. In addition, the growth rates of the numbers of bugs related to the products *Drivers* and *ACPI* are faster than those of other products.
#8	A bug related to the product *Drivers*, *ACPI* or *Platform* is more likely to be a BOH; a bug in the product *File System* or *Core* (i.e., *Memory Management*, *Process Management* or *Timers*) is more prone to be a NAM or ARB; and an *IO/Storage* bug is more likely to be an ARB.
#9	The evolutionary trends of the bug type proportions differ among products. For example, the proportion of NAMs related to the product *File System* tends 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 for most bugs are related to the *drivers* directory.
#11	A bug whose repair location is related to the *drivers* directory is more likely to be a BOH, whereas a bug whose repair location is related to the *fs* directory is more likely to be a NAM or ARB. A bug whose repair location is related to the *core* directory (i.e., *kernel*, *mm* or *include*) is more likely to be an ARB, while 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	The proportion of BOHs among regression bugs is higher than that among nonregression bugs. Accordingly, a regression bug is more prone to be a BOH, whereas a nonregression 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 with time. Moreover, the proportion of regression bugs tends to stabilize around a constant value of 50% of the total bugs 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, and feature improvement and implementation.
#16	Approximately one-third of regression bugs are caused by bug fixes. In addition, it is found that regression bug chains occur, since the fix for one regression bug can lead to another regression bug.
Findings on time to fix	
#17	The average time needed to fix a 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 required to fix a nonregression bug.
Findings on software metrics	
#19	With the evolution of the clustering coefficient, a Linux version tends to possess a higher proportion of BOHs when its call graph has a smaller clustering coefficient.
#20	The characteristics of BOHs and MANs are significantly different in terms of the network metric of degree. The sum of the degrees (k^{out}, k^{in} and k) and the average and maximum degrees (k^{out} and k) for a MAN are significantly larger than those for a BOH.
#21	The characteristics of BOHs and MANs are not significantly different in terms of the network metrics of the clustering coefficient and betweenness.
#22	The characteristics of BOHs and MANs are significantly different in terms of the network metric of closeness. The average or minimum closeness for a BOH is significantly larger than that for a MAN.

of the Linux OS from a complex network perspective. This provides a research foundation for addressing research questions 4 and 5.

RQ5: Do the characteristics of different bug types differ in terms of certain network metrics?

To address this research question, we investigate the differences in bug type characteristics based on the complex network metrics considered in this paper. The answer to this research question can help us to better understand the relations between bug types and software structure information. In addition, similar to ARB prediction using software complexity metrics [24], the results can further be utilized to measure bug characteristics for bug prediction or classification, if statistically significant differences exist between the bug types in terms of these software complexity metrics. We label a bug and its bug type based on the affected functions, which are determined by inspecting the corresponding bug-fixing patch. The affected functions are nodes in the corresponding Linux call graph. Thus, the network metrics of the nodes with which a bug is labeled can be acquired and further utilized to represent the characteristics of the bug and its bug type. The analysis procedure is detailed in the Study Methodology section.

The contributions of this paper are summarized into 22 findings, as shown in Table II. 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 [25]. Several new analyses are conducted. For example, 1) in the bug type analysis, we present the detailed types of nonbugs and investigate the differences in bug type proportions among repair locations; 2) in the regression bug analysis, the causes of regression bugs are further examined and discussed; 3) in the fixing time analysis, the relationship between regression status and fixing time is presented; and 4) in the software metric analysis, we propose a procedure for analyzing bug type characteristics based on complex network metrics and compare the bug type characteristics in terms of different network metrics.

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

II. Research Data

To address the research questions presented in the Introduction, we collected two types of research data: the source code of the Linux kernel and Linux bug reports. These data and their collection procedure are described in detail as follows.

A. Linux OS

The Linux source code was obtained from the official website [26]. Linux was originally developed by Linus Torvalds in 1991. The development history of Linux can be classified into three stages according to the evolution of the development model [16], [23]. The first stage includes the releases from versions 1.0 to 2.5, and the second stage consists of the version 2.6 series. The third stage consists of all releases beginning with version 3.0. In the first stage, the versions were numbered in the form "a.b.c," where the first digit "a" represents the kernel version number and the major and minor version numbers are denoted by the second digit "b" and the third digit "c," respectively. Odd major version numbers correspond to development versions, whereas 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 the development model when releasing version 2.6. In this stage, 4 digits were used to denote each release, starting with 2.6.11 [27]. The third digit indicates the major version, with new functionality, whereas the fourth digit indicates the minor version, with bug fixes and security patches. In 2011, to celebrate the 20th anniversary of Linux, the developers retired the numbering method that was adopted for the version 2.6 series and returned to using 3 digits to denote all releases since version 3.0. It should be noted that starting with version 2.6.11, major versions have been released approximately every two or three months.

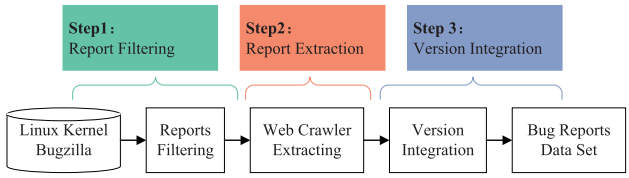

Fig. 1. Procedure for bug data collection and aggregation. Step 1: Report filtering. Step 2: Report extraction. Step 3: Version integration.

B. Linux Bug Data

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

1) *Step 1: Report filtering*: In this paper, the reports from the Linux kernel Bugzilla database were initially filtered based on the conditions of "Status: CLOSED" and "Resolution: CODE_FIX."
2) *Step 2: Report extraction*: Once the filtered list of target reports was obtained, each report on the list was downloaded to our 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.
3) *Step 3: Version integration*: It was necessary to process the recorded versions of the collected reports for the following two reasons. First, some users use distribution versions that are based on the Linux kernel but still report problems in the Linux kernel Bugzilla. For example, the 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 formally released versions of Linux. In addition, some users compile the latest source code from Git (for example, recorded versions "2.6.21-rc5-git9," "2.6.22-rc5-git8," and "2.6.23-rc6-git2") but do not used the formal release versions. Thus, the recorded versions were integrated into the major versions in accordance with the Linux version numbering method described in Section II-A. For example, recorded 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.

TABLE III
DETAILS OF THE DATA SET

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

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

III. Study Methodology

In this section, we first define the bug terminology used in this paper and describe the procedure applied for classifying bug types. Finally, a procedure for analyzing bug characteristics based on network metrics is proposed.

A. Terminology

Before introducing the terminology, we note that the terms *fault*, *bug*, and *defect* are all regarded as having the same meaning in this paper. We adopt the bug type classification from [8], [11], and [12]. A bug is categorized as a BOH or a MAN depending on the complexity of the fault triggering conditions. The definitions of BOHs and MANs are given as follows.

1) BOH: A bug that can be consistently reproduced under a well-defined set of conditions since its activation and/or error propagation are simple.
2) MAN: A bug that is difficult to reproduce since its activation and/or error propagation are complex. The complexity of the triggering conditions may be related to the influence of a direct factor, for example, a time lag between fault activation and failure occurrence. The complexity could also be due to indirect factors, for example, the system-internal environment, the timing of inputs and operations, or the sequencing of inputs and operations.

MANs are separated into two subtypes, i.e., NAMs and ARBs, according to whether such a bug would lead to the software aging phenomenon. As depicted in Fig. 2, NAMs and ARBs also have subtypes. The definitions of NAM/ARB subtypes are presented in Table I.

In addition, the definitions of regression and nonregression bugs are presented below.

1) *Regression Bug*: A bug that causes a feature that worked normally in previous versions to stop working after a certain event.
2) *Nonregression Bug*: A bug that leads to the failure of a new feature in the current version.

B. Bug Taxonomy

The procedures for bug report classification and the identification of regression bugs are presented in the following. For a given bug report, the classification procedure is separated into three steps, as shown in Fig. 3. Each step is described in detail as follows.

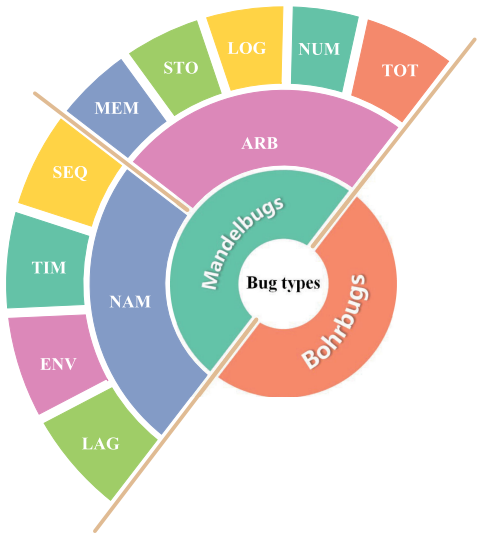

Fig. 2. Based on the complexity of the fault triggering conditions, bugs are classified as either BOHs or MANs. MANs can be further categorized into NAMs and ARBs. There are also subtypes of NAMs and ARBs [8].

1) *Step 1: Data Cleaning*. The bug report should first be inspected to confirm whether it truly represents a bug. In this paper, requests for new features or enhancements, compile-time issues (e.g., make errors or linking errors), documentation issues (e.g., missing or outdated documentation or harmless warning outputs), duplicates, and issues related to operator error are regarded as nonbugs and are removed from the analysis.
2) *Step 2: Extraction of Fault Triggers*. The description, discussion comments, patches, log files, and other files attached to 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 behavior, for example, the parameters or states of the program that were changed by the bug and the manner in which a changed parameter or state propagated; and 3) the manifestation of the failure, for example, what phenomena the user observed when the failure occurred.
3) *Step 3: Classification*. Finally, based on the extracted fault triggers and the bug manifestation phenomena as well as the definitions of each subtype of ARBs and NAMs, the bug report is checked to determine whether it qualifies as an ARB, NAM, or BOH. If a bug is identified as an ARB but there is insufficient information to determine its failure mechanics, its bug type is marked as *ARU*. Similarly, *NAU* is the label assigned to NAM bugs for which the information necessary to extract the activation and error

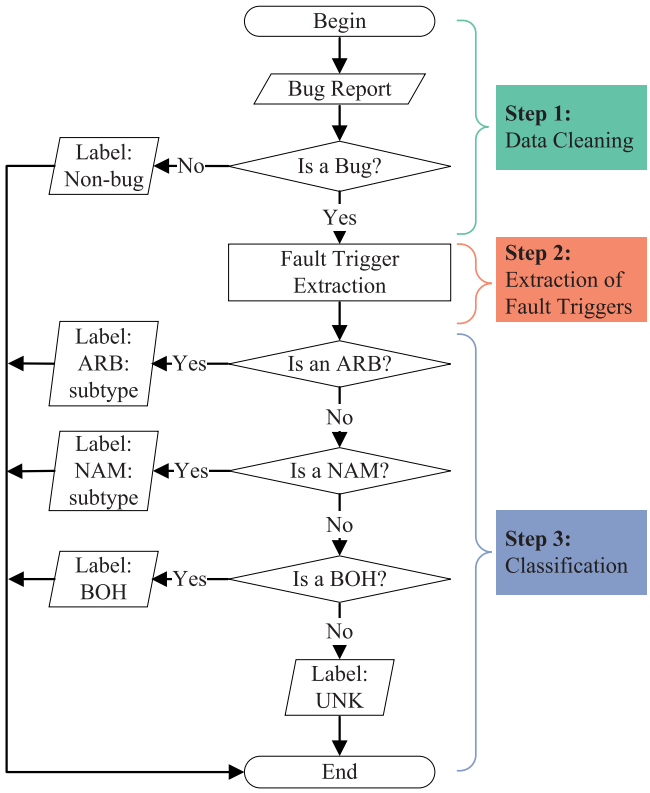

Fig. 3. Procedure of bug report classification. Step 1: Data cleaning. Step 2: Extraction of fault triggers. Step 3: Classification.

propagation conditions is lacking. Finally, if a report does not provide sufficient information to classify the corresponding bug as an ARB, NAM, or BOH, it is labeled as being of an unknown type (*UNK*).

In addition, regression bugs are identified according to two criteria. The first concerns the value of the regression flag in the bug report. If a bug was reported as a regression bug, it will have a value of "Yes" in the regression field on its reporting page. However, a bug report cannot be reliably determined to represent a regression or nonregression bug based solely on its regression flag, since the regression flag is submitted by the reporter, who may misclassify the bug. In addition, the regression flags of some reports are blank, because the reporters did not record this information. Therefore, the second criterion is based on an examination of the textual messages associated with a bug report (e.g., description and discussion comments) to determine whether it is consistent with the definition of a regression bug.

All the work done with respect to bug type classification and regression bug classification was manually performed by the authors, and when cases of suspect classification were encountered, they were resolved through cross-checks and discussions. To clarify the classification procedure, examples of fault-trigger-based bug type classification and examples of regression bug classification are presented in the following. These examples can serve as references for the form of the classification results. Furthermore, to enable other researchers to understand and implement our classification procedure more easily and for convenience in further analysis, our data have been released on our research website.[1]

TABLE IV
EXAMPLES OF CLASSIFIED BUGS

ID	Type	Description
1166	NAM/SEQ	"unplugging and then unmounting a USB DoK causes oops...Steps to reproduce: 1. plugging in the USB device 2. mounting it 3. unplugging 4. unmounting (note the order: unplug happens before unmount)."
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..."

Examples of fault-trigger-based bug type classification and examples of regression bug classification are presented in Tables IV and V, respectively.

In the following, we describe the process for the classification of the LAG, ENV, TIM, and SEQ subtypes. With respect to the LAG type, for the example bug ID-12684 shown in Table IV, we determine that the machine seems to perform normally after the first suspend/resume from its description, but it refuses to resume after a second suspend. We consider the bug to be activated during or before the first suspend/resume operation, whereas the failure manifests in the second suspend, which indicates that there is a time lag between the activation of the bug and the manifestation of its consequent failure. Moreover, the user description indicates that the failure can be rectified by forcibly saving and restoring the ACPI nonvolatile state. Therefore, this bug is classified as an LAG. Regarding the ENV type, for the example bug ID-11805 presented in Table IV, we can determine that the activation of the bug is due to the interaction with external hardware/storage, i.e., the mounting of the XFS partition. In addition, memory also has an impact on the action and/or error propagation. Thus, this bug is classified as an ENV. Regarding the TIM type, for the example bug ID-6045 shown in Table IV, from its description, we can infer that the timing of the operations, i.e., device discovery of the root disk and the attempt to mount the root file system, impacts the activation of the bug. Therefore, the bug is determined to be a TIM. With respect to the

[1][Online]. Available: https://guanpingxiao.github.io/data/linux.xlsx

TABLE V
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 ... 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 wire-less 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."

SEQ type, for the example bug ID-1166 shown in Table IV, it can be observed from this description that the sequencing (i.e., relative order) of the operations is the factor that impacts the activation. Thus, we classify this bug as an SEQ.

C. Bug Analysis Based on Network Metrics

To measure bug characteristics using network metrics, we need to know the functions that are affected by each bug. The affected functions are identified from the bug-fixing patch. For Linux bug reports, the associated patches are usually provided as attachments or Git commit IDs. In this paper, four representative complex network metrics, including two local metrics, i.e., the *degree* k and the *clustering coefficient* C, and two global metrics, i.e., the *betweenness* C_B and the *closeness* C_C, are selected. The degree of a node in a network is the number of edges connected to it. The clustering coefficient measures the probability that a node's neighbors are also neighbors of each other. The betweenness is a shortest-path-based metric representing the centrality of a node in the network, while the closeness is another measure of centrality. In the following, we detail the procedure for analyzing bug type characteristics based on these network metrics. The network modeling of the Linux call graph and the definitions of the selected network metrics are presented in Appendix A.

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

1) *Step 1: Extraction of Affected Functions*. The purpose of this step is to extract the affected functions from the bug-fixing patch (in diff file format) associated with a bug. The patch is first manually inspected to identify the changed statements, and then, we determine which functions contain those changed statements. The patch explicitly specifies the location of each changed statement in the source files, such as the line number or function. Therefore, we can easily determine the affected functions. These affected functions are recorded in a table, which also contains the ID of the bug and its bug type. For example, in Fig. 4, the changed statements in the bug-fixing patch for bug "ID-1" are in the functions func1 and func2. These functions are recorded in the table of affected functions. Notably, a bug will be discarded if the functions that are changed cannot be identified from the bug-fixing patch, for example, if the patch modifies only data structures.
2) *Step 2: Acquisition of Network Metrics*. Once the affected functions have been extracted in step 1, a table of the affected functions is obtained. We then output a table consisting of the network metrics associated with all of the functions from the corresponding Linux call graph [23]. To obtain the network metrics associated with the affected functions, a Python tool written by us is utilized to automatically search for and record these network metrics by matching the function names between the two tables. The network metrics considered in this paper include the degree k, the clustering coefficient C, the betweenness C_B, and the closeness C_C of each affected function. For example, the in-degree k^{in} of func1 is 1, whereas its out-degree k^{out} is 2, as shown in Fig. 4.
3) *Step 3: Representation of Bug Characteristics*. After step 2, the network metrics of the functions affected by the bug have been obtained. To represent the characteristics of the bug, several operations (i.e., sum, average, maximum, and minimum) are applied for the integration of the network metrics. For example, we can use the sum of the out-degrees of the affected functions func1 and func2 as a network metric for bug "ID-1," as depicted in Fig. 4. The details of the integration methods are elaborated in Appendix A.C.

IV. PROPORTIONS AND EVOLUTION OF BUG TYPES

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

A. Overall Proportions and Evolution of the Bug Types

Finding #1*: Among the 5741 bug reports, actual bugs account for 76.3%, and nonbugs account for 23.7%. Of the nonbugs, approximately 75.2% of nonbugs are compile-time issues, feature requests or documentation issues.*

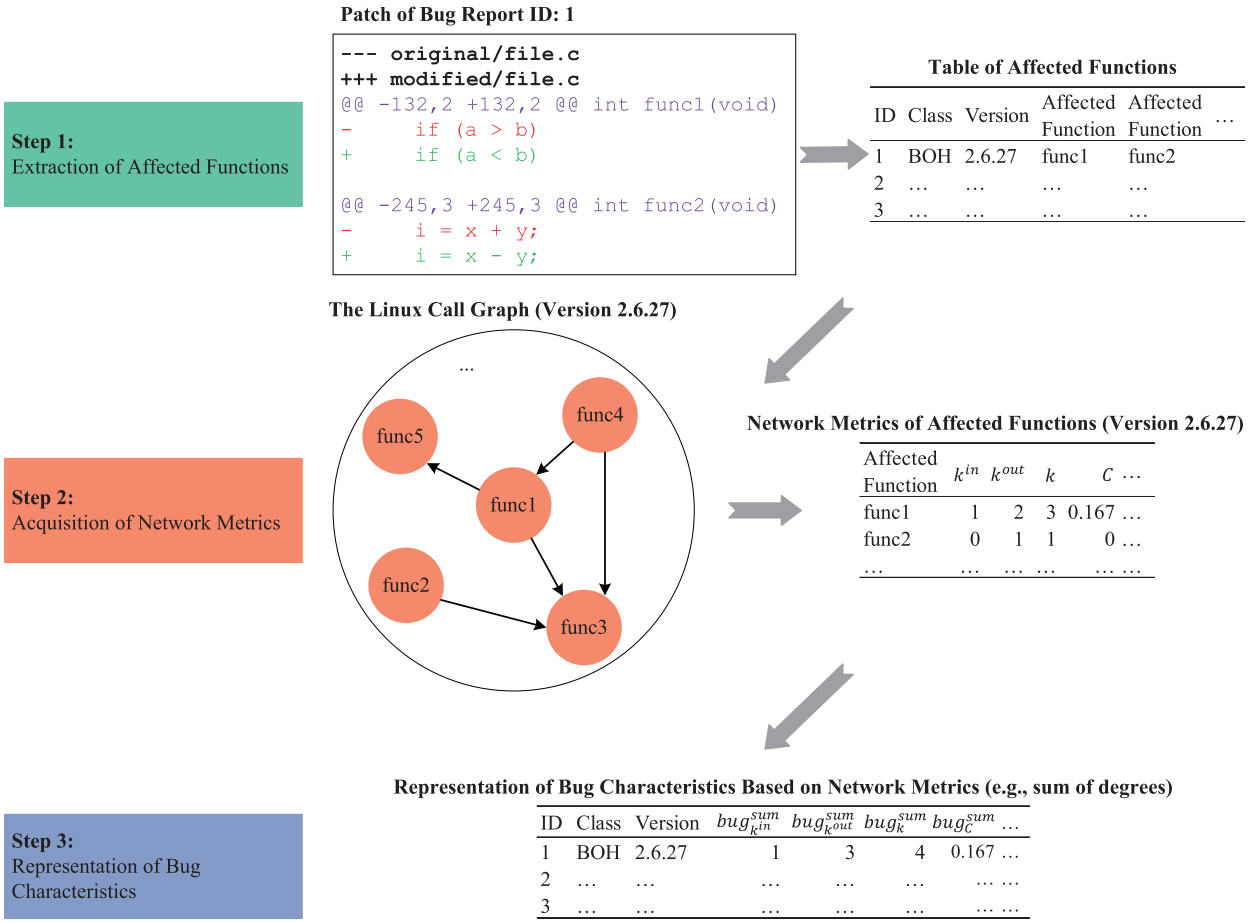

Fig. 4. Procedure for analyzing bug characteristics based on network metrics. Step 1: Extraction of affected functions. Step 2: Acquisition of network metrics. Step 3: Representation of bug characteristics.

TABLE VI
NUMBERS AND PERCENTAGES OF ACTUAL BUGS AND NONBUGS

	# of Reports	% of Reports
Actual bugs	4378	76.3
Nonbugs	1363	23.7
Total	5741	100.0

TABLE VII
NUMBERS AND PERCENTAGES OF NONBUGS SUBTYPES

	# of Reports	% of Reports
Compile-time issues	547	40.1
Feature requests	350	25.7
Documentation issues	128	9.4
Other	338	24.8
Total	1363	100.0

Table VI illustrates the classification results for the collected bugs. From the bug type classification results in Table VI, it can be observed that actual bugs account for 76.3% of all collected bugs, whereas the percentage of nonbugs is 23.7%. This result was tested via the chi-square test, with a null hypothesis of no significant difference between the numbers of reports corresponding to each type. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 1583.4$, $df = 1$, $p < 0.001$). Therefore, we can reject the null hypothesis. It should be noted that in this paper, as described in Section III-B, reports related to requests for features or enhancements, compile-time issues, documentation issues, duplicates, or operator error are regarded as nonbugs. Bug report triage is an important task with the purpose of determining whether a report is meaningful [28]. It can be observed from Table VII that approximately 75.2% of the nonbugs are compile-time issues, feature requests, or documentation issues. The results presented in Table VII were tested via the chi-square test, with a null hypothesis of no significant difference in the numbers of reports corresponding to each nonbug subtype. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 257.9$, $df = 3$, $p < 0.001$); thus, the null hypothesis can be rejected. Although these reports could represent unfriendly experiences for users, either there is no urgent need to organize them for integration into the Linux development process, or they can usually be resolved easily (e.g., compile-time issues and documentation issues). However, ensuring that bug report data

TABLE VIII
NUMBERS AND PERCENTAGES OF ACTUAL BUG TYPES

	# of Reports	% of Reports
BOH	2444	55.8
NAM	1386	31.7
ARB	205	4.7
UNK	343	7.8
Total	4378	100.0

TABLE IX
NUMBERS AND PERCENTAGES OF NAM SUBTYPES

	# of Reports	% of Reports
TIM	516	37.3
ENV	506	36.5
LAG	265	19.1
SEQ	10	0.7
NAU	89	6.4
Total	1386	100.0

are of high quality can not only reduce the burden on bug tracking system maintainers, but also benefit research on measurements and predictions based on the data. 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 on the reporting page, there could be a custom drop-down field specifying the types of reported problems, for example, "Bug," "Feature Request," "Documentation Issue," and "Compile-time Issue." Alternatively, the bug writing guidelines could suggest that reporters prefix the summary of each report with the corresponding text "Bug:," "Feature Request:," "Compile-time Issue:," or "Documentation Issue:." In addition, since approximately 40% of nonbugs are compile-time issues, developers should compile their source code before releasing a new version.*

***Finding #2**: Among the 4378 actual bugs, the proportions of BOHs and MANs are 55.8% and 36.4%, respectively.*

The total numbers and percentages of each bug type, i.e., BOHs, NAMs, ARBs, and UNKs, are presented in Table VIII. The total number of classified bugs, including BOHs, NAMs, and ARBs, is 4035, which accounts for 92.2% of all actual bugs. The results presented in Table VIII were tested via the chi-square test, with a null hypothesis of no significant difference in the numbers of reports corresponding to each bug type. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 2980.4$, $df = 3$, $p < 0.001$), which indicates that there is a significant difference in the numbers of bugs of the different types as reported in Table VIII. More than half of the actual bugs in Linux are BOHs, as shown in Table VIII. This result indicates that BOHs account for a large proportion of the bugs in Linux. BOHs still pose a serious problem. This finding confirms similar results from previous studies, which have found remarkably large numbers (proportions) of BOHs in other software systems (e.g., MySQL: 56.6%, HTTPD: 81.1%, AXIS: 92.5%, and Android: 65.2%) [8], [29] and even in mature critical systems (e.g., JPL/NASA space mission systems: 61.4%) [30]. Although BOHs are easy to reproduce and to debug when detected, they are still difficult to detect in large and complex software systems, e.g., the Linux kernel. This situation may be attributed to the inefficacy of testing activities.

Moreover, MANs, including NAMs and ARBs, account for 36.4% of the actual bugs. Obviously, they constitute a nonnegligible portion of Linux bugs. Compared with those in other software systems, the proportion of MANs in Linux is close to those in MySQL (38% [8]), space mission onboard software (36.5% [30]), and the Android OS (31.4% [29]). Since the fault triggering conditions of MANs are more complex than those of BOHs, specific testing methods and fault tolerance techniques should be developed to handle them.

***Implications**: To mitigate BOHs, we suggest conducting sufficient testing before release using, for example, the Linux Test Project [31]. To mitigate the nonnegligible proportion of MANs, we suggest developing specific testing methods, such as combinatorial testing [32], and cost-effective fault tolerance techniques, such as environment diversity [33].*

***Finding #3**: NAMs account for 87.1% of MANs. The major subtypes of NAMs are TIM (37.3%), ENV (36.5%), and LAG (19.1%).*

Table VIII shows that NAMs and ARBs account for 31.7% and 4.7%, respectively, of the 4378 actual bugs. We further explore the proportions of subtypes of NAMs and ARBs, the two subcategories of MANs. Table IX shows the numbers and percentages of NAM subtypes. These results were tested via the chi-square test, with a null hypothesis of no significant difference in the numbers of reports corresponding to each NAM subtype. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 780.4$, $df = 4$, $p < 0.001$), meaning that the null hypothesis can be rejected. It can be observed from Table IX that TIMs (37.2%), ENVs (36.5%), and LAGs (19.1%) constitute the most prevalent subtypes of NAMs. These results are consistent with those of a previous study [8]. It is reasonable for TIMs and ENVs to exist in high proportions due to the characteristics of an OS. The Linux OS inherently must handle concurrent activities, access shared resources, and manage hardware, which will inevitably lead to timing-related problems, such as deadlock (example: "ID-26232: Multiple framebuffer oops and sysfs attribute deadlock") and race conditions (example: "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 LAG subtype, the root causes are usually data corruption problems or incorrect state changes. When data are corrupted or a state value is incorrect, failures will manifest once these errors propagate through the system. In Linux, the most common faults of the LAG subtype are null pointer dereference problems, such as "ID-10048: ipv4/fib_hash.c: fix NULL dereference."

***Implications**: Since TIMs, ENVs, and LAGs constitute the major subtypes of NAMs, approaches to debugging, testing, or fault tolerance for mitigating the impact of NAMs in Linux should focus on such bugs. More specifically, to handle TIMs, it is suggested that more attention should be paid to thread conflicts and*

TABLE X
NUMBERS AND PERCENTAGES OF ARB SUBTYPES

	# of Reports	% of Reports
MEM	141	68.8
STO	16	7.8
NUM	12	5.8
LOG	11	5.4
TOT	3	1.5
ARU	22	10.7
Total	205	100.0

locking mechanisms in Linux. To test for ENVs, the focus should be placed on the Linux hardware interfaces, whereas for LAGs, the values of data variables and state variables, especially those that are passed in modules or subsystems, should be carefully examined.

***Finding #4**: ARBs account for 12.9% of MANs. The major subtype of ARBs is MEM (68.8%).*

The numbers and percentages of the ARB subtypes are presented in Table X. These results were tested via the chi-square test, with a null hypothesis of no significant difference in the numbers of bug reports corresponding to each ARB subtype. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 406.6$, $df = 5$, $p < 0.001$). Thus, we can conclude that there is a significant difference in the counts shown in Table X. The MEM subtype accounts for more than two-thirds (68.8%) of the ARBs. This result is close to those for other software systems [8], [29], [30]. Linux is written in the C language, in which memory management is handled by the developers. This makes it more prone to software aging. In addition, leaks are associated with storage, numerical problems, and other logical resources.

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

In the following, we present the results of analyzing the evolution of the bug type proportions. The evolutionary analysis is conducted from two perspectives: evolution over versions and evolution over time. As described in Section II-B, all recorded versions were integrated into major versions. We calculated the statistics of the classified bugs corresponding to these integrated versions. To ensure the validity of the analysis results, a continuous series of adjacent versions (i.e., 2.6.15 to 3.0) with more than 50 bugs were chosen to analyze the evolution of the bug type proportions over versions. In addition, the evolution of the bug type proportions over time was calculated. Since the life cycles of two major versions could overlap (for example, the version 3.7 series was maintained from December 2012 to March 2013, whereas the version 3.8 series was maintained from February 2013 to May 2013), all versions were considered in the temporal analysis. The results of the evolutionary analysis are depicted

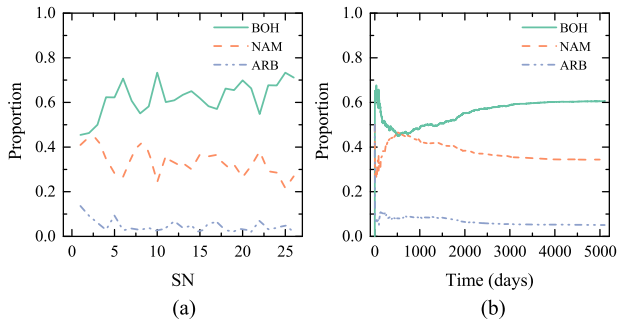

Fig. 5. Evolution of bug type proportions among classified bug reports. (a) Evolution over versions. Note that SN represents a sequential number that is assigned to each version according to its release date; e.g., the sequence number of version 2.6.15 is 1, and that of version 3.0 is 26. This notation will be used throughout the remainder of the paper. (b) Evolution over time.

TABLE XI
MANN–KENDALL TREND TEST RESULTS FOR FIG. 5

		Z	p value	Trend
Evolution	BOH	3.22	0.001	Increasing
over	NAM	-2.76	0.006	Decreasing
versions	ARB	-1.39	0.165	No Trend
Evolution	BOH	91.37	<0.001	Increasing
over	NAM	-86.55	<0.001	Decreasing
time	ARB	-93.46	<0.001	Decreasing

in Fig. 5. It is noted that a data point in Fig. 5(a) represents a proportion relative to the number of bugs in a specific version, whereas a data point in Fig. 5(b) represents a proportion relative to the cumulative number of bugs up to that time.

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

It is apparent in Fig. 5 that the proportion of BOHs tends to increase slowly with the evolution of versions and with time. In contrast, the proportion of NAMs tends to decrease. In addition, the proportion of ARBs tends to decrease slightly and to become more stable over versions and time than the proportions of BOHs and NAMs do. Moreover, as shown in Fig. 5(b), the proportions of all three types stabilize around constant values after approximately 4000 days. The evolutionary trends seen in Fig. 5 were tested by means of the Mann–Kendall trend test [36], [37]. The results of the Mann–Kendall trend test in this paper were calculated based on R [38]. As shown in Table XI, the test results indicate that for a significance level of $\alpha = 0.05$, the evolutionary trends of the proportions of BOHs and NAMs over both versions and time are statistically significant, whereas the trend of the evolution of the proportion of ARBs over versions is not statistically significant. The evolutionary 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

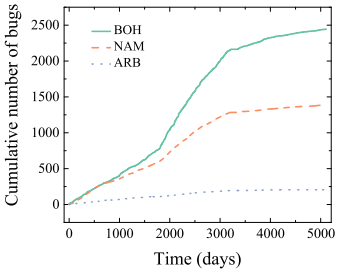

Fig. 6. Cumulative number of bugs of each bug type.

June 22, 2015, whereas version 4.2 was released on August 30, 2015. As Linux evolves, its complexity continuously grows, as reflected by the increasing number of lines of code [16] and the increasing number of functions [23]. Meanwhile, a massive number of features are introduced, which might lead to more BOHs in newly released versions. Although code changes can also introduce NAMs, the proportion of NAMs slowly decreases due to the faster rate of increase of the cumulative number of BOHs compared with that of NAMs, as shown in Fig. 6. Moreover, the results related to Finding #5 are further explained and verified by the subsequent detailed analyses of the bug types in relation to products (i.e., Section IV-C) and regression status (i.e., Sections V-A and V-B).

***Implications**: Because of the frequent-release nature of the Linux development paradigm, it is suggested that developers should pay greater attention to bugs introduced in new features and should conduct continuous functional testing activities.*

B. Comparison of Bug Types Among Versions

In this section, four versions, i.e., 2.6.0, 2.6.24, 2.6.27, and 2.6.32, are selected to explore the evolution of the bug type proportions over the lifetime of a version and to compare this bug type evolution among versions. These four versions are those with the most bug reports, and two of them are long-term supported versions (i.e., 2.6.27 and 2.6.32). The results for these versions are presented in Fig. 7, in which the evolution of the MAN proportion and the evolution of the cumulative number of bugs are shown. Note that a data point on the thick line represents the proportion of MANs relative to the cumulative number of bugs up to that time, while a data point on the thin line represents the cumulative number of bugs up to that time.

***Finding #6**: The proportion of MANs and its evolutionary trend are different among versions. For all selected versions, the proportions tend to stabilize around constant values over time since eventually, few new bugs will be reported.*

It can be observed from Fig. 7(a) that the proportion of MANs was 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 attributable to the implementation of a new CPU scheduler. In versions before 2.6.0, Linux used a single run queue that relied on a linked list of threads to manage all runnable tasks. However, to ensure better scalability on SMP systems, with version 2.6.0, Linux began to utilize a per-CPU

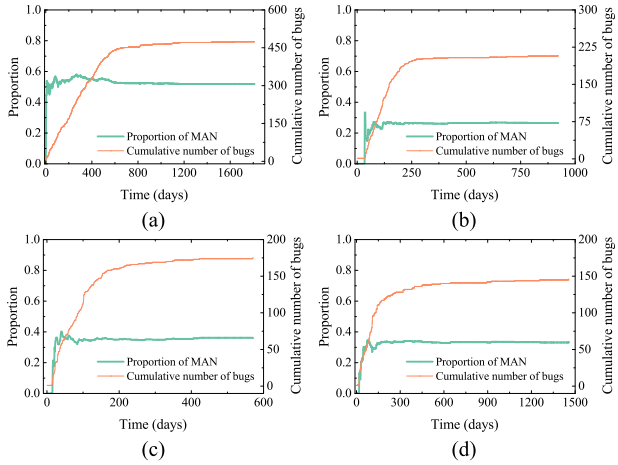

Fig. 7. Evolution of the MAN proportion and the cumulative number of bugs for four selected versions: (a) 2.6.0, (b) 2.6.24, (c) 2.6.27, and (d) 2.6.32.

lock rather than the single run-queue lock for task management. Therefore, the kernel is preemptive beginning with version 2.6.0, since it can respond immediately to interactive processes [39]. Since the developers needed time to adapt to the new scheduler, this feature might have led to more NAMs, especially timing-related bugs, such as race conditions and deadlocks.

In addition, Fig. 7(c) shows that the proportion of MANs tended to increase in version 2.6.27. The trend was tested via the Mann–Kendall trend test. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($Z = 16.9$, $p < 0.001$), indicating an increasing trend. This result may be because version 2.6.27 was one of the long-term supported versions. These are special versions that are supported by the developers over a very long period. During maintenance, a long-term supported version may become more stable. Thus, the proportion of difficult-to-fix bugs (i.e., MANs) will increase, whereas the proportion of easily isolated and reproduced bugs (i.e., BOHs) will decrease. Moreover, the proportion of MANs in these versions tends to stabilize over time because fewer new bugs are reported. This phenomenon may be due to one of the following two reasons. Fewer bugs may exist in these versions, or a similar number of bugs may still exist, but fewer users/developers may use/maintain these versions, resulting in fewer bug reports.

***Implications**: The different proportions of MANs in different versions might be due to the different major new features included in those versions. Compared with the other three versions, Linux 2.6.0 introduced a significant breakthrough (i.e., the new scheduler mechanism [39]), and it also possessed the highest proportion of MANs. Thus, developers can expect more/fewer MANs depending on whether the next release will include major new features.*

C. Comparison of Bug Types Among Products

Linux consists of several functional products, such as drivers, file systems, and 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.

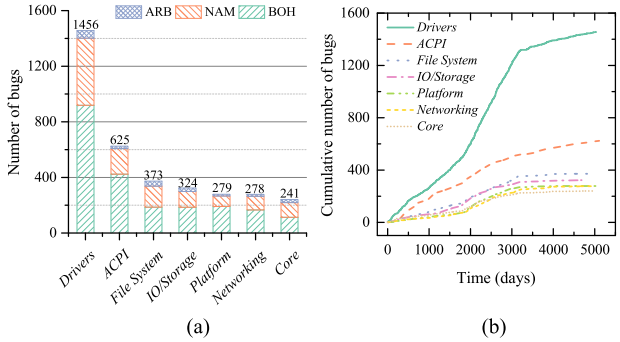

Fig. 8. Numbers and evolution of classified bugs among products. (a) Numbers of bugs. (b) Evolution of bugs with time. *Core* includes three products: *Memory Management*, *Process Management* and *Timers*.

TABLE XII
CORRELATIONS BETWEEN BUG TYPES AND PRODUCTS

	BOH	NAM	ARB
Drivers	919 (**2.1**)	482 (-0.8)	55 (-3.0)
ACPI	423 (**3.7**)	185 (-2.5)	17 (-3.0)
File System	186 (-4.7)	151 (**2.9**)	36 (**4.2**)
IO/Storage	185 (-1.5)	114 (0.5)	25 (**2.3**)
Platform	191 (**2.6**)	75 (-2.6)	13 (-0.3)
Networking	166 (-0.5)	98 (0.5)	14 (0.0)
Core	113 (-4.7)	106 (**3.4**)	22 (**3.0**)

Note: The values in parentheses are the standardized Pearson residuals for the independence testing, and those in bold are those that exceed a value of 2.

In the Linux kernel Bugzilla, the first step of reporting a bug is to select the product (e.g., *Drivers*, *File System*, *Memory Management*, or *Process Management*) to which the bug is related. The statistics of the bug type proportions as calculated for different products and the evolutionary trends in the number of bugs are depicted in Fig. 8. The products presented here are those that possess the highest numbers of bugs. The numbers of BOHs, NAMs, and ARBs in these products, respectively, account for 89.3%, 87.4%, and 88.8% of the total numbers of BOHs, NAMs, and ARBs. The differences in the bug counts in the products shown in Fig. 8 were tested via the chi-square test, with a null hypothesis of no significant difference in the numbers of bug reports for each product. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 2233.6$, $df = 6$, $p < 0.001$). Thus, the null hypothesis can be rejected.

Finding #7*: Driver bugs, i.e., bugs related to the products Drivers and ACPI, account for 51.6% of all classified bugs. In addition, the growth rates of the numbers of bugs related to the products Drivers and ACPI are faster than those of other products.*

From Fig. 8(a), we find that the numbers of bugs in the products *Drivers* (1456) and *ACPI* (625) together account for 51.6% of all classified bugs (4035). Finding #7 confirms a result similar to that from a previous study [9], in which driver bugs were found to account for 52.9% of all bugs from a small data set sampled from the same kernel Bugzilla. In addition, the growth rates of the numbers of bugs related to these two products are faster than those for other products, as can be observed in Fig. 8(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 in their source codes accounts for approximately 50% of the total number of functions [23]. Notably, the name of the product *ACPI* is short for advanced configuration and power interface, which indicates that this 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 during Linux testing, developers should pay more attention to device drivers.*

As shown in Fig. 8(a), the bug type proportions differ among different products. We utilized the chi-square test to determine whether there is an association between bug type and product, with the null hypothesis that bug type is not associated with the product. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 83.9, df = 12, p < 0.001$). Therefore, we can reject the null hypothesis and conclude that there is a significant association between bug type and product.

To further test the independence of each bug type and each product, we calculated the standardized Pearson residual, which is the residual divided by its standard error [40], with the results shown in Table XII. Note that since the standardized Pearson residuals follow a standard normal distribution (with mean 0 and standard deviation 1), a standardized Pearson residual is significant if its absolute value is greater than 2 [40]. When a standardized Pearson residual is significant, a positive value indicates that the observed frequency is significantly greater than would be expected by chance. For example, as shown in Table XII, the standardized Pearson residual value for the product *Drivers* and the bug type BOH is 2.1, indicating that bugs related to product *Drivers* are more likely to be BOHs. By contrast, a negative value implies that the observed frequency is significantly less than would be expected by chance. For example, as depicted in Table XII, the standardized Pearson residual value for the product *Drivers* and the bug type ARB is -3.0, indicating that bugs related to product *Drivers* are less likely to be ARBs.

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

In the following, we focus on the positive standardized Pearson residual values. As presented in Table XII, the standardized Pearson residual values for the products *Drivers*, *ACPI*, and *Platform* and the bug type BOH are greater than 2, whereas the standardized Pearson residual values for the products *File System* and *Core* and the bug types NAM and ARB are greater than 2. In addition, the standardized Pearson residual value for the product *IO/Storage* and the bug type ARB is greater than 2. The different bug type manifestations among products might be attributable to inherent differences in the products. With respect

to the products *Drivers*, *ACPI*, and *Platform Specific/Hardware*, although these products serve as bridges between an OS and devices, failures in these products will be observed by users as more direct manifestations. In addition, Linux drivers have a higher proportion of BOHs might be because of the reason that more than 60% of driver bugs belong to the categories of device protocol violations (38%) and generic programming bugs (23%), whereas the rest of driver bugs are related to OS protocol violations (20%) and concurrency bugs (19%) [41]. When the driver runs in a manner that violates the required hardware protocol, device protocol violations occur and often result in the hardware failing to provide its required service. It is found that most of device protocol violations and generic programming bugs usually consistently manifest at the user side, i.e., devices will function incorrectly every time. For example, device protocol violations ("ID-1285: radeonfb do not correctly detect LCD" and "ID-13377: Microphone no longer works on Toshiba Satellite A100") and generic programming bugs ("ID-323: double logical operator drivers/net/fc/iph5526.c" and "ID-39842: savagefb.h CARD SERIES definition typo"). Note that the consistent failure manifestations at the user side do not indicate that these bugs can be easy to detect during testing, as it is often impractical and not cost-effective to test the driver with all supported hardware. Therefore, bugs occurring in these products are more likely to be BOHs. By contrast, the products *File System*, *IO/Storage*, and *Core* (i.e., *Memory Management*, *Process Management* and *Timers*) are considered to be basic, core functions of the OS, which means that the interactions among these products tend to be more complex and tightly coupled [22]. Accordingly, bugs related to these products are more likely to be NAMs or ARBs.

***Implications**: We suggest that different testing strategies should be selected for testing different products. For example, more functional testing and compatibility testing should be performed when testing the product Drivers, whereas combinatorial testing [32] might be useful for testing products such as File System, IO/Storage, and Core, since bugs related to these products are more prone to be NAMs or ARBs.*

Moreover, the evolution of the bug type proportions among the selected products is explored, as shown in Fig. 9. A data point in Fig. 9 represents a proportion relative to the cumulative number of bugs up to that time. The trends in Fig. 9 were examined by means of the Mann–Kendall trend test, with the results shown in Table XIII. For a significance level of $\alpha = 0.05$, the proportions of BOHs in all products exhibit statistically significant increasing trends, while the proportions of NAMs exhibit statistically significant decreasing trends in all products except *File System* (increasing) and *IO/Storage* (no trend). For ARBs, the proportions exhibit statistically significant decreasing trends in all products except *Drivers* (increasing).

***Finding #9**: The evolutionary trends of the bug type proportions differ among products. For example, the proportion of NAMs related to the product File System tends 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.*

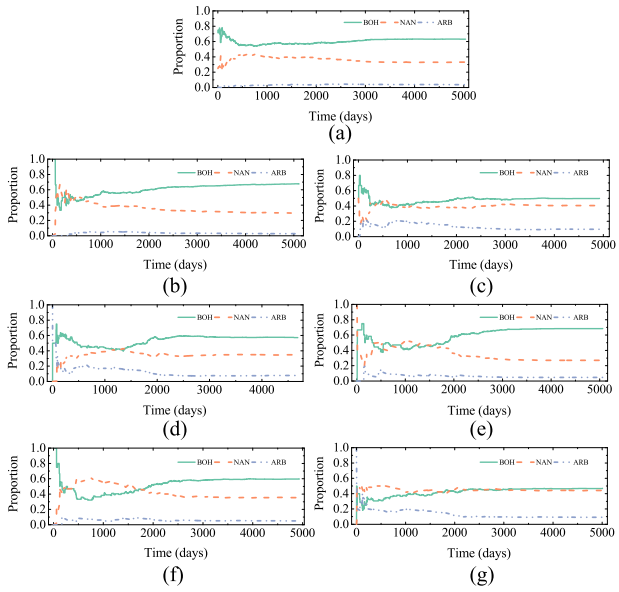

Fig. 9. Evolution of bug type proportions in selected products, including (a) *Drivers*, (b) *ACPI*, (c) *File System*, (d) *IO/Storage*, (e) *Platform*, (f) *Networking*, and (g) *Core* (i.e., *Memory Management*, *Process Management*, and *Timers*).

TABLE XIII
MANN–KENDALL TREND TEST RESULTS FOR FIG. 9

		Z	p value	Trend
Drivers	BOH	67.57	<0.001	Increasing
	NAM	-68.76	<0.001	Decreasing
	ARB	34.83	<0.001	Increasing
ACPI	BOH	94.74	<0.001	Increasing
	NAM	-94.81	<0.001	Decreasing
	ARB	-58.413	<0.001	Decreasing
File System	BOH	51.23	<0.001	Increasing
	NAM	22.87	<0.001	Increasing
	ARB	-69.36	<0.001	Decreasing
IO/Storage	BOH	27.68	<0.001	Increasing
	NAM	0.45	0.654	No Trend
	ARB	-61.70	<0.001	Decreasing
Platform	BOH	80.96	<0.001	Increasing
	NAM	-79.77	<0.001	Decreasing
	ARB	-77.54	<0.001	Decreasing
Networking	BOH	75.656	<0.001	Increasing
	NAM	-72.58	<0.001	Decreasing
	ARB	-46.05	<0.001	Decreasing
Core	BOH	89.05	<0.001	Increasing
	NAM	-19.04	<0.001	Decreasing
	ARB	-89.42	<0.001	Decreasing

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

The findings presented in this section, i.e., Findings #7, #8, and #9, illustrate that driver bugs, which are more likely to be BOHs, account for the largest proportion of bugs in Linux. In addition, the proportion of BOHs tends to increase over time in all products. These results indicate that more BOHs will be

TABLE XIV
NUMBERS AND PERCENTAGES OF CLASSIFIED BUGS THAT HAVE PATCHES IN THEIR REPORTS

	# of Reports	% of Reports
Patch found	2821	69.9
Patch not found	1214	30.1
Total	4035	100.0

newly introduced over time, thus providing further evidence for Finding #5.

D. Comparison of Bug Types Among Repair Locations

The products mentioned in bug reports are closely related to the directories of the Linux kernel source code. For example, the source code for the product *Drivers* is mainly located in the *drivers* directory. However, there could be discrepancies between the products recorded in reports and the actual root causes in the source code, since the reporters might misjudge which products have problems. In the following, we report the calculated statistics for the classified bugs that have patches. Furthermore, we examine the patches for the classified bugs to determine the corresponding repair locations. For example, the code fix for "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 for this bug is in the *drivers* directory. When code fixes are related to several directories, the directory with major changes is considered the repair location.

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

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

The statistical results for the classified bugs that have patches associated with their reports are depicted in Table XIV. More than two-thirds of the classified bugs have patches. It should be noted that the lack of a patch for a bug does not necessarily indicate that said bug has not been fixed; rather, it indicates only that the patch is not provided in the report. Furthermore, we investigate the proportions of the bug types among five repair locations, i.e., *drivers*, *arch*, *fs*, *core* (*kernel*, *mm*, and *include*), and *net*, as shown in Fig. 10. The numbers of BOHs, NAMs, and ARBs in these repair locations, respectively, account for 95.5%, 95.6%, and 96.2% of all BOHs, NAMs, and ARBs in all repair locations. The difference in the numbers of reports for each repair location in Fig. 10 was examined via the chi-square test, with a null hypothesis of no significant difference in the numbers of bug reports for each repair location. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 3204.0$, $df = 4$, $p < 0.001$), which implies that the null hypothesis can be rejected. Fig. 10 shows that the *drivers* directory accounts

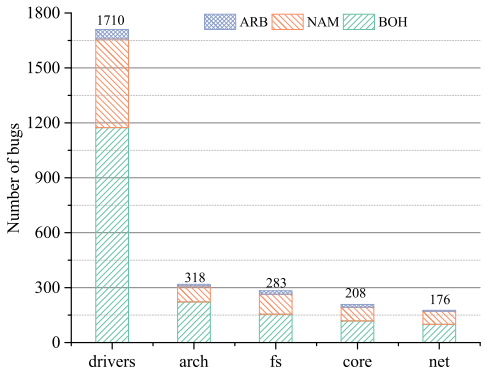

Fig. 10. Numbers of bugs based on their repair locations. The location *core* includes three directories, i.e., *kernel*, *mm*, and *include*.

TABLE XV
CORRELATIONS BETWEEN BUG TYPES AND REPAIR LOCATIONS

	BOH	NAM	ARB
drivers	1174 (**4.3**)	484 (-3.5)	52 (-2.4)
arch	222 (1.7)	87 (-1.4)	9 (-0.9)
fs	155 (-4.1)	109 (**3.0**)	19 (**2.8**)
core	118 (-2.8)	75 (1.8)	15 (**2.8**)
net	100 (-2.5)	71 (**2.9**)	5 (-0.6)

Note: The values in parentheses are the standardized Pearson residuals for the independence testing, and those in bold are those that exceed a value of 2.

for the most repair locations, since most bugs are related to the product *Drivers*.

Moreover, the chi-square test was used to determine whether there is an association between bug type and repair location, with the null hypothesis that bug type is not associated with a repair location. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 47.0$, $df = 8$, $p < 0.001$). Thus, the null hypothesis can be rejected, and there is a significant association between bug type and repair location. To explicitly test the independence of each bug type and each repair location, the standardized Pearson residuals were calculated, and the results are presented in Table XV. The standardized Pearson residual value for the directory *drivers* and the bug type BOH is greater than 2, and the standardized Pearson residual values for the directory *fs* and the bug types NAM and ARB are also greater than 2. In addition, the standardized Pearson residual values for the directory *net* and the bug type NAM and for the directory *core* and the bug type ARB are greater than 2.

Furthermore, we compare the bug densities among repair locations in version 2.6.0, which has the largest number of bugs. Table XVI shows the estimated bug density for each repair location. Note that the bug density in Table XVI is calculated as the ratio of the number of bugs in a given version to the code size (lines of code, calculated with cloc[2]). Compared to previous studies (e.g., mean bug density of open source projects: 4.66 bugs/kLoC [42] and Linux: 0.34 bugs/kLoC [8]), the much lower

[2]cloc (Count Lines of Code) v1.76: [Online]. Available: https://github.com/AlDanial/cloc.

TABLE XVI
ESTIMATED BUG DENSITY FOR EACH REPAIR LOCATION IN VERSION 2.6.0

Repair location	# of Bugs	kLoC	# of Bugs/kLoC
drivers	**143**	**1798.7**	**0.0795**
arch	36	668.1	0.0539
fs	21	366.5	0.0573
core	6	409.4	0.0147
net	7	228.1	0.0307

Note: The maximum value in each column is shown in bold.

TABLE XVII
NUMBERS AND PERCENTAGES OF REGRESSION AND NONREGRESSION BUGS

	# of Reports	% of Reports
Regression bugs	2020	50.1
Nonregression bugs	2015	49.9
Total	4035	100.0

bug density in Table XVI is because of the reason that the source of bugs only includes the classified bugs, but excludes reports that are still open and UNK. Therefore, the bug density in the table is not the real bug density, as it only represents the selected bugs found in a specific time window. Accordingly, Table XVI is used to compare the relative differences in the bug densities for different repair locations. It can be observed from the table that the *drivers* directory has not only the largest number of bugs, but also the highest bug density, although its code size is quite large. This observation confirms a similar result from a previous study [4], in which Linux drivers have been shown to have the highest bug density. Finding #11 provides further evidence of the correlation between bug type and product.

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

V. CHARACTERISTICS OF REGRESSION BUGS

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

A. Proportion of Regression Bugs

As described in Section III-A, a bug that can cause a normal feature that worked in previous versions to misbehave or fail completely in more recent versions is classified as a regression bug. In this section, we first present the calculated statistics for the numbers of regression bugs and nonregression bugs, as depicted in Table XVII. The difference in the numbers of regression bugs and nonregression bugs in Table XVII was tested via the chi-square test, with a null hypothesis of no significant difference in the numbers of bug reports in each category. For a significance level of $\alpha = 0.05$, the test result ($\chi^2 = 0.006$, $df = 1$, $p = 0.94$) shows that the null hypothesis cannot be rejected. It is apparent from Table XVII that approximately

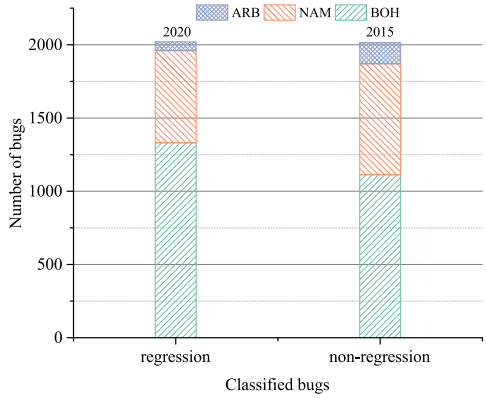

Fig. 11. Comparison of bug types between regression and nonregression bugs.

TABLE XVIII
CORRELATIONS BETWEEN BUG TYPES AND REGRESSION STATUSES

	BOH	NAM	ARB
Regression bug	1331 (**6.9**)	629 (-4.3)	60 (-6.1)
Nonregression bug	1113 (-6.9)	757 (**4.3**)	145 (**6.1**)

Note: The values in parentheses are the standardized Pearson residuals for the independence testing, and those in bold are those that exceed a value of 2.

half of the classified bugs in Linux are regression bugs, i.e., problems with existing normal features being broken. In comparison with other software systems, it is found that the proportion of regression bugs in Linux is close to that in Google Chromium (51.1% [43]).

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

***Finding #13**: The proportion of BOHs among regression bugs is higher than that among nonregression bugs. Accordingly, a regression bug is more prone to be a BOH, whereas a nonregression bug is more likely to be an NAM or ARB.*

A comparison of the bug type proportions between regression and nonregression bugs is shown in Fig. 11. The proportion of BOHs among regression bugs is higher than that among nonregression bugs. In contrast, more MANs are observed among nonregression bugs. We used the chi-square test to analyze whether there is an association between bug type and regression status, with the null hypothesis that the bug type is not associated with whether a bug is a regression bug. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 66.5$, $df = 2$, $p < 0.001$). This result implies that the null hypothesis can be rejected and that there is a significant association between bug type and regression status. To further test the independence of each bug type and regression status, we calculated the standardized Pearson residuals, as shown in Table XVIII. The standardized Pearson residual value for regression bugs and the BOH type is greater than 2, and the standardized Pearson residual values for nonregression bugs and the NAM and ARB types are also greater than 2. These results indicate that regression bugs are more likely to be BOHs than MANs. Regression bugs are annoying to Linux OS users because when they encounter

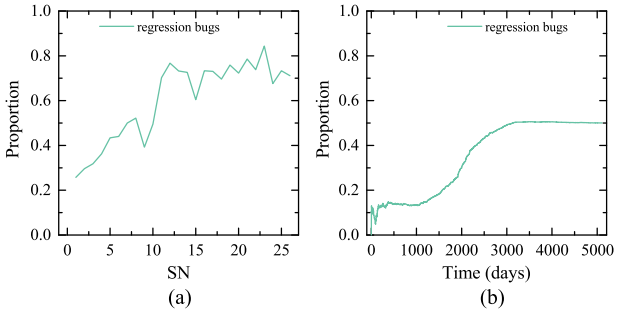

Fig. 12. Evolution of the proportions of regression bugs among the classified bug reports. (a) Evolution over versions. (b) Evolution over time.

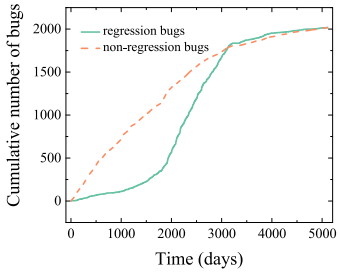

Fig. 13. Cumulative numbers of regression and nonregression bugs.

severe regression bugs, users may be unwilling to upgrade their OSs, although new versions might offer more features or security enhancements. As a result, users may continue to use older OS versions, making their systems more prone to security problems.

***Implications**: We suggest that developers implement more regression testing before releasing a new version to reduce the occurrence of existing normal features being broken, since half of the bugs are of the regression type. When dealing with nonregression bugs, specific testing methods such as combinatorial testing [32] will be more effective, since a nonregression bug is more likely to be an NAM or ARB.*

B. Evolution of Regression Bugs

In this section, we explore the evolutionary trends of the proportions of regression bugs. The evolutionary analysis is conducted from two perspectives, namely, evolution over versions and evolution over time, as shown in Fig. 12(a) and (b), respectively. The cumulative numbers of regression and nonregression bugs are shown in Fig. 13. Note that to ensure the validity of the analysis results, a continuous series of adjacent versions (i.e., 2.6.15 to 3.0) with more than 50 bugs was chosen to analyze the evolution of the bug type proportions over versions. A data point in Fig. 12(a) represents a proportion relative to the number of bugs in a specific version, whereas a data point in Fig. 12(b) represents a proportion relative to the cumulative number of bugs up to that time. The evolutionary trends in Fig. 12 were tested by means of the Mann–Kendall trend test, with the results shown in Table XIX. For a significance level of $\alpha = 0.05$, the results are statistically significant; i.e., the proportion of regression bugs increases over versions and over time.

TABLE XIX
MANN–KENDALL TREND TEST RESULTS FOR FIG. 12

	Z	p value	Trend
Evolution over versions	4.45	<0.001	Increasing
Evolution over time	77.32	<0.001	Increasing

***Finding #14**: The proportion of regression bugs tends to increase with the evolution of versions and with time. Moreover, the proportion of regression bugs tends to stabilize around a constant value of 50% of the total bugs after approximately 3500 days.*

With the evolution of Linux, an increasing number of features are introduced. For example, version 2.4 supports fewer than 40, 15, 40, and 30 types of device drivers, platform architectures, file systems, and network protocols, respectively. By contrast, in version 4.1, these numbers are increased to more than 110, 25, 60, and 50, respectively. During the evolution of the OS, a massive number of code changes are implemented in Linux due to the introduction of a significant number of features. These changes inevitably lead to regression bugs. Bug-fixing activities are another cause of regression bugs; examples include "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 related to Finding #14 are expected, and they indicate that more BOHs will be newly introduced over time, since regression bugs are more likely to be BOHs, according to Finding #13. This provides further evidence supporting Findings #2 and #5.

***Implications**: Since the proportion of regression bugs increases over versions and time and such bugs are more likely to be BOHs in Linux, we suggest that developers be more careful when implementing code changes, such as newly introduced features or bug fixes, and that continuous regression testing should be conducted before releasing a new feature or fixing a bug [17].*

C. Causes of Regression Bugs

In this section, the causes of regression bugs are analyzed based on manual inspection of the descriptions and comments in the reports. In regression bug reports, the reporters usually state that the bugs are caused by changes from certain Git commits. In addition, since the affected versions are often very close to previous versions (e.g., a feature may work normally in version 2.6.31.1 but fail in version 2.6.31.2), maintainers may ask the reporters to run git bisect, a binary search, to find the first bad commit (i.e., the first change leading to the bug). Two authors of this paper independently inspected the causes of regression bugs by examining the descriptions (for bug fixing or feature change) of the bad commits, as provided in the reports. For example, the description of bug ID-10679 states that the problem was caused by "pcspkr: fix dependancies." Thus, we consider the cause of the bug to be a bug fix. Note that not all reports specify the bad commit and that for some commits, it is difficult to determine from their descriptions whether they are intended to fix bugs or change features. Thus, the causes of such bugs were labeled as

TABLE XX
NUMBERS AND PERCENTAGES OF CAUSES OF REGRESSION BUGS

	# of Reports	% of Reports
Feature change	1046	54.9
Bug fix	613	32.1
Unknown	248	13.0
Total	1907	100.0

TABLE XXI
EXAMPLES OF DEVELOPMENT ACTIVITIES RELATED TO FEATURE CHANGES

Activity	Description	Example
Code optimization	Code cleanup and simplification	ID-32222
	Logic improvement	ID-12538
	Code conversion and refactoring	ID-10364
Improvement	Improvement of an existing feature	ID-11875
Implementation	Implementation of a new feature	ID-51881
Disabling	Disabling of an existing feature	ID-14700

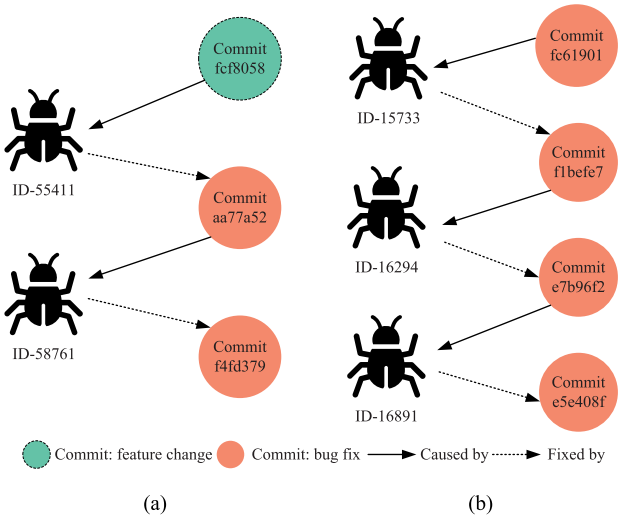

Fig. 14. Two typical examples of regression bug chains initially caused by (a) a feature change or (b) a bug fix.

unknown. To ensure consistent results, cross-checks were performed, and conflicting cases were resolved through discussion to reach a consensus among the authors. Since Git has been used to track changes to the Linux kernel since version 2.6.12 [44], we investigated only the regression bugs for version numbers starting from 2.6.12 to ensure the validity of the analytical results. As a result, 1907 regression bugs were selected, which account for 94.4% of all regression bugs.

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, and feature improvement and implementation.*

The numbers and percentages of the causes of regression bugs as identified via manual inspection are presented in Table XX. These results were examined via the chi-square test, with a null hypothesis of no significant difference in the numbers of regression bugs with different causes. For a significance level of $\alpha = 0.05$, the test result is statistically significant ($\chi^2 = 502.1$, $df = 2$, $p < 0.001$). Therefore, the null hypothesis can be rejected. We find that more than half of the regression bugs are due to activities related to feature changes, as depicted in Table XX. Moreover, we identified four kinds of development activity examples related to features changes, as summarized in Table XXI. With the evolution of Linux, it is often necessary to update "ancient" code, i.e., to perform code optimization, such as removing or implementing simplifications to obsolete code. If these code optimization activities are not handled properly, they will inevitably lead to regression bugs. In addition, the maintenance of existing features and the introduction of new features can introduce regression bugs.

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

Table XX shows that approximately one-third of regression bugs are introduced by bug fixes. From inspection of these regression bugs, an interesting phenomenon is observed: chains of regression bugs occur, which means that the fix for one regression bug can be the cause of another regression bug. Two typical regression bug chains are illustrated in Fig. 14. The first type of regression bug chain is initially caused by a feature change, as shown in Fig. 14(a). Regression bug "ID-55411" was caused by "commit fcf8058: cpufreq: Simplify cpufreq_add_dev()." To fix this bug, the 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."

By contrast, the second type of regression chain is initiated by a bug fix, as depicted in Fig. 14(b). "commit fc61901: agp/intel-agp: Clear entire GTT on startup" was introduced in version 2.6.32.4 for bug-fixing purposes, but it caused the problem reported as bug "ID-15733." Later, this bug was addressed by "commit f1befe7: agp/intel: Restrict GTT mapping to valid range on i915 and i945." Unfortunately, since this fix was incorrect, it resulted in another 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, care should be taken in conducting activities such as code cleanup and simplification, code conversion and refactoring, and feature improvement and implementation. In addition, with respect to regression bug chains, further study will be required to determine whether they are prevalent in Linux as well as their characteristics, since such chains will inevitably increase the burden of software maintenance. We suggest developing techniques for capturing the relationships between regression bugs and their causes and fixes, such as representing these relationships as networks, which could be further used to predict regression bugs. As a developer*

TABLE XXII
COMPARISON OF FIXING TIMES BETWEEN BOHS AND MANS

	Average fixing time (days)	Standard deviation
BOHs	148.7	268.1
MANs	182.0	287.8

TABLE XXIII
COMPARISON OF FIXING TIMES BETWEEN REGRESSION BUGS AND NONREGRESSION BUGS

	Average fixing time (days)	Standard deviation
Regression bugs	109.1	222.5
Nonregression bugs	214.6	313.0

commented in a regression bug report, "I'd like to avoid a regression fix for a regression fix for a regression fix."

VI. RELATIONSHIP BETWEEN BUG TYPE AND FIXING TIME

In this section, we present the analytical results for *RQ3: What is the relationship between bug type and fixing time?* This research question has two aspects, i.e., the difference in fixing time between BOHs and MANs and that between regression and nonregression bugs.

According to the definitions of BOHs and MANs, the fault triggering conditions of an MAN are more complicated than those of a BOH. Therefore, it is expected that more time will be required to fix an MAN. In the following, the time to fix a bug is estimated as the difference between the reporting time and the resolution time (i.e., the time when the resolution was marked as CODE_FIX), since no fixing times are 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 times and their standard deviations for BOHs and MANs are presented in Table XXII. The average time taken to fix an MAN is 182.0 days, whereas fixing a BOH takes an average of 148.7 days. We used the Wilcoxon–Mann–Whitney test [45] to further verify these results, with the null hypothesis that the fixing times for BOHs and MANs are sampled from the same distribution. For a significance criterion of $\alpha = 0.05$, we obtained a p value of less than 0.001. This result indicates that the null hypothesis can be rejected. Accordingly, we conclude that the time taken to fix a MAN tends to be longer than that taken to fix a BOH, which is consistent with previous studies regarding HTTPD [8], AXIS [8], and Android [29].

The significantly different fixing times between BOHs and MANs might be attributable to the different times required to handle them during the bug management process. The management process for a bug progresses through several states, including unconfirmed, new, assigned, and resolved [6]. The majority of the difference between the times taken to fix a BOH and an MAN might be because of the different transition times between the assigned and resolved states. Due to the difference in the complexity of the fault triggering conditions between BOHs and MANs, developers usually require considerable time to obtain sufficient information to detect the underlying root causes in the code to resolve an MAN. In addition, the nondeterministic nature of MANs could also result in more time taken to reproduce an MAN. Consequently, a longer time is required to fix an MAN.

Implications*: Due to the longer fixing time, specific testing methods (e.g., combinatorial testing [32]) and cost-effective fault tolerance techniques (e.g., environment diversity [33]) would be helpful for handling MANs.*

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

We also investigated the difference in fixing time between regression and nonregression bugs, as shown in Table XXIII. The average time taken to fix a regression bug (109.1 days) is significantly shorter than that taken to fix a nonregression bug (214.6 days). These results were again verified by means of the Wilcoxon–Mann–Whitney test [45], with the null hypothesis that the fixing times for regression and nonregression bugs are sampled from the same distribution. For a significance criterion of $\alpha = 0.05$, we obtained a p value of less than 0.001. This result indicates that the null hypothesis can be rejected. Therefore, fixing a regression bug tends to require less time than is required to fix a nonregression bug. This result is reasonable since the causes of regression bugs can usually be found quickly. For most regression bugs, especially recent ones, reporters or developers can use git bisect to search for the first bad Git commit changes that cause the regression bugs. In some circumstances, a regression bug is solved simply by reverting the initial bad changes.

Implications*: Although the average time taken to fix a regression bug tends to be shorter than that taken to fix a nonregression bug, more care should be taken in fixing regression bugs, since inappropriate regression fixes can 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 the bug types based on software metrics and present the analytical results for *RQ4: Is there any software metric that can reflect the evolution of the bug type proportions?* and *RQ5: Do the characteristics of different bug types differ in terms of certain network metrics?*

A. Correlations Between Bug Type Proportions and Network Metrics

To demonstrate a software metric that can be utilized to reflect the evolution of the bug type proportions, we investigate the relationships between the proportion of BOHs and complex network metrics, including the clustering coefficient C, the degree k, the betweenness C_B, and the closeness C_C. To ensure the validity of the analytical results, we selected only the versions with more than 50 bugs and calculated their network metrics for the corresponding Linux call graph [23]. Fig. 15 depicts the relationships between the proportion of BOHs and the four metrics.

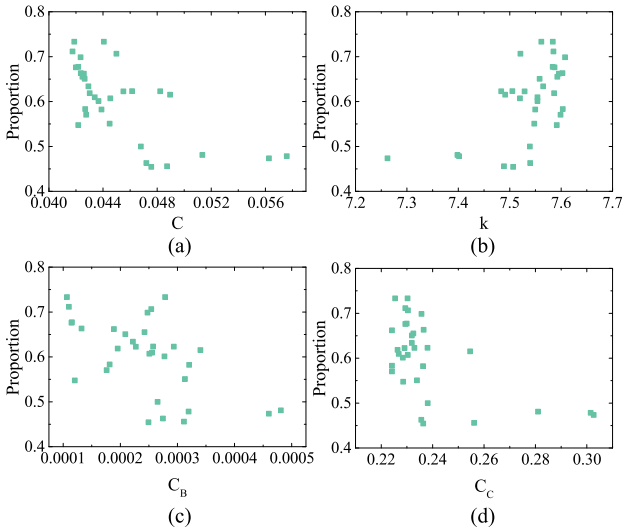

Fig. 15. Relationships between the proportion of BOHs and network metrics. (a) Clustering coefficient C, (b) degree k, (c) betweenness C_B, and (d) closeness C_C.

TABLE XXIV
SPEARMAN'S CORRELATION COEFFICIENTS FOR THE PROPORTION OF BOHS AGAINST THE NETWORK METRICS

	BOH - C	BOH - k	BOH - C_B	BOH - C_C
ρ	-0.7	0.5	-0.5	-0.4
p value	<0.001	0.003	0.001	0.020

In addition, to further analyze these relationships, the Spearman correlation analysis was applied, with the results shown in Table XXIV. Since the MAN type is the complementary antonym of the BOH type, we only present the analysis results of the BOH type in Fig. 15 and Table XXIV, and the correlations for the MAN type have opposite relationships comparing with that of the BOH type.

Finding #19*: With the evolution of the clustering coefficient, a Linux version tends to possess a higher proportion of BOHs when its call graph has a smaller clustering coefficient.*

As shown in Table XXIV, for a significance level of $\alpha = 0.05$, all Spearman correlations are statistically significant. The sample size here 32. According to the rule of thumb for interpreting the size of a correlation coefficient [46], there is a strong correlation (i.e., $0.7 \leq |\rho| < 0.9$) between the proportion of BOHs and the clustering coefficient C, while the relationships between the proportion of BOHs and the other network metrics have moderate (i.e., $0.5 \leq |\rho| < 0.7$) or low correlations (i.e., $0.3 \leq |\rho| < 0.5$). Therefore, we are only interpreting the relationship between the proportion of BOHs and C. As described in Appendix A.B, the C is utilized to evaluate the tendency of a network to form tightly connected neighborhoods. According to the results for the evolution of the Linux call graph [23], C decreases with the evolution of versions. Therefore, the strong negative correlation between the proportion of BOHs and C indicates that with the evolution of C, the proportion of BOHs tends to increase. In contrast, because of its strong positive correlation with C, the proportion of MANs tends to decrease. This phenomenon might be attributable to the fact that a large C indicates tight local connections among functions, which could lead to more interactions among internal functions. As a result, more complex bug manifestations (i.e., MANs) could occur. By contrast, a small C indicates loose local connections among functions, which could result in less complex bug manifestations (i.e., BOHs).

Implications*: The clustering coefficient can be utilized as an indicator of the expected bug type proportions in future Linux versions. More specifically, for a future release, we can calculate the clustering coefficient of its call graph to qualitatively measure its bug type proportions.*

B. Analysis of Bug Type Characteristics Based on Network Metrics

In this section, we further analyze the characteristics of the bug types based on network metrics. The functions affected by each bug were extracted from its associated bug-fixing patch. As reported in Section IV-D, there are patches associated with 2821 of the classified bugs. After performing the analysis procedure described in Section III-C, to ensure the validity of the results, we identified 1359 bugs whose affected functions could be determined and extracted from their patches, spanning 18 versions, with each version containing at least 50 bugs. In the following, we explore the differences in the bug type characteristics based on four network metrics, including the degree k, the clustering coefficient C, the betweenness C_B, and the closeness C_C. We calculated the average network metric values corresponding to the bug types for each version based on the four integration methods introduced in Appendix A.C (i.e., $\text{version}_{\text{nm}}^{\text{sum}}$, $\text{version}_{\text{nm}}^{\text{ave}}$, $\text{version}_{\text{nm}}^{\text{max}}$, and $\text{version}_{\text{nm}}^{\text{min}}$). The detailed values of the network metrics corresponding to the bug types for each version are provided in Appendix B. With the null hypothesis that the network metrics for BOHs and MANs are sampled from the same distribution and for a significance criterion of $\alpha = 0.05$, the results were tested using the Wilcoxon–Mann–Whitney test [45]. Comparisons of the bug type characteristics based on the network metrics are presented in Table XXV.

Finding #20*: The characteristics of BOHs and MANs are significantly different in terms of the network metric of degree. The sum of the degrees (k^{out}, k^{in}, and k) and the average and maximum degrees (k^{out} and k) for a MAN are significantly larger than those for a BOH.*

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

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

It is apparent from Table XXV 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 functions affected by a MAN tend to be implemented with more function calls and to be called by more other functions. Additionally, a comparison of k^{out} and

TABLE XXV
COMPARISONS OF BUG TYPE CHARACTERISTICS BASED ON NETWORK METRICS

k^{out}	$version^{sum}_{k^{out}}$	$version^{ave}_{k^{out}}$	$version^{max}_{k^{out}}$	$version^{min}_{k^{out}}$
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}	$version^{sum}_{k^{in}}$	$version^{ave}_{k^{in}}$	$version^{max}_{k^{in}}$	$version^{min}_{k^{in}}$
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	$version^{sum}_{k}$	$version^{ave}_{k}$	$version^{max}_{k}$	$version^{min}_{k}$
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	$version^{sum}_{C}$	$version^{ave}_{C}$	$version^{max}_{C}$	$version^{min}_{C}$
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	$version^{sum}_{C_B}$	$version^{ave}_{C_B}$	$version^{max}_{C_B}$	$version^{min}_{C_B}$
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	$version^{sum}_{C_C}$	$version^{ave}_{C_C}$	$version^{max}_{C_C}$	$version^{min}_{C_C}$
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**

k^{in} shows that the differences in the characteristics of BOHs and MANs seem to be more strongly reflected by k^{out}. The average and maximum k^{out} values of the functions affected by a MAN are significantly larger than those for a BOH, which indicates that the functions affected by an MAN tend to be more complex on average than those affected by a BOH and that the affected function with the most function call implementations also tends to be more complex for an MAN than for a BOH. Thus, the complexity difference between BOHs and MANs is well reflected in the differences in their characteristics in terms of the network metric of degree.

In addition, it is found from Table XXV that the characteristics of the different bug types in terms of the network metrics of the clustering coefficient and betweenness are not significantly different. Although the same metric, i.e., C, is used, the analysis here is different from that in Section VII-A. The analysis in Section VII-A was performed by considering the entire Linux call graph, whereas the statistical results for C in this section are focused on the functions affected by a bug. The bug manifestation process not only is influenced by the affected functions, but also could be impacted by the error propagating functions. For C_B, the results indicate that we cannot consider the characteristics of the different bug types to be significantly different based on this metric.

Furthermore, Table XXV shows that the characteristics of the bug types in terms of the network metric of closeness are significantly different. The average closeness for a BOH is significantly larger than that for an MAN. A node with a higher closeness is closer to other nodes. Therefore, the results of the closeness analysis could explain why BOHs manifest more directly and consistently than MANs do.

***Implications**: The characteristics of the different bug types can be distinguished based on two complex network metrics, i.e., degree and closeness. The results can be further utilized to predict MANs at the function level, file level, or subsystem level. For example, similar to ARB prediction using software complexity metrics [24], for file-level MAN prediction, we can use the network metrics of the affected functions to represent the characteristics of source files that contain MANs. The network characteristics can be used along with or integrated with other software complexity metrics (e.g., program size or McCabe's cyclomatic complexity) to train a classification model on examples of MANs using machine learning algorithms. The trained classifier can then be applied to new files to predict whether they are "MAN-prone" or "MAN-free." In addition, the network metrics can be further utilized as features for the automatic classification of bug types.*

VIII. THREATS TO VALIDITY

The validity of empirical studies is naturally subject to limitations. Since our examination focused on bugs in the Linux kernel, we do not intend to present any general implications regarding the bug characteristics in all software systems. Although many of the findings have been compared to those for other projects from previous studies, there is still a need to compare the novel findings of this paper with findings for other software systems, e.g., real-time OSs [47]. Additionally, we identify the following threats.

A. Selection of Bug Reports

In this paper, the bug data were exclusively drawn from closed and fixed reports. The reason is that reports of bugs that have not yet been fixed may contain incomplete and inaccurate information. The bug type proportions could be influenced by considering these future closed and fixed bug reports, since the properties of the fixed and unfixed bugs may differ. For example, MANs may tend to be fixed less frequently than BOHs. However, analyses of fixed bugs have also found remarkably large numbers (proportions) of BOHs in other software systems [8], [29] and even in mature critical systems [30]. Thus, it is possible that our focus on fixed bugs has not biased the results. To further verify our results, in our future work, we plan to examine reports of bugs that have not been fixed. In addition, the bug characteristic analysis was performed based on bug data from the Bugzilla database for the official Linux kernel. There are several other bug sources available for various Linux distributions, such as Arch Linux, Gentoo Linux, and Ubuntu Linux. Similar analyses conducted on these bug sources could indicate different bug characteristics compared with the analysis results based on official Linux bug data.

B. Manual Inspection and Analysis

In this paper, the bug classification and analysis were manually performed. Four of the authors were involved in the manual classification and analysis, and all authors participated in the experimental design and the discussions of the results. The time consumption for all of the manual work was as follows.

1) The classification of the bug reports, including the fault-trigger-based bug types and regression bugs, took us approximately four months.
2) The determination of the repair locations took nearly two weeks.
3) The analysis of the causes of the regression bugs took us approximately one month.
4) The extraction of the affected functions took approximately one month.
5) The analysis and discussion of the results took nearly one and a half months.

To ensure the consistency of the results obtained by different authors, cross-checks were performed, and conflicting cases were resolved through discussion to reach a consensus among the authors. In addition, for the manual analysis, several tools (existing or written by us) were utilized to help us ensure the correctness and consistency of the results. For example, for bug counting, we first counted bugs based on different filtering conditions (e.g., bug type, product, and version) using Microsoft Excel. We then wrote Linux bash command scripts (using, e.g., grep and uniq) to verify the results automatically. Regarding figure generation, most of the figures used in the analysis were generated by OriginPro, a data analysis and graphing software. For the statistical testing of the results, the tests were performed using SPSS [48] and R [38] to ensure the correctness and consistency of the test results. However, as in all empirical studies in which manual inspection is needed, the possibility of classification mistakes and manual inspection mistakes could not be completely avoided in this paper.

C. Correctness of Data Information

Since bug reports are reported by users and developers, the correctness of the provided information (e.g., products and versions) may affect the results of the relevant analyses in this paper. To mitigate this impact, problematic reports (e.g., omissions, errors, and nonbugs) were excluded from the analyzed data set as much as possible. For example, in the first step of the bug classification process, nonbugs, which accounted for 23.7% of all reports, were identified and excluded from the analysis. Although the kernel Bugzilla triages and marks duplicate reports, we still found duplicates among the collected reports. In addition, we specified unknown types (i.e., UNK, NAU, and ARU) to account for those reports that were difficult to classify due to insufficient or uncertain information. For example, UNKs accounted for 7.8% of all actual bugs, NAUs accounted for 6.4% of all NAMs, and ARUs accounted for 10.7% of all ARBs.

D. Bug Type Definitions

The bug types are defined based on the bug manifestation properties in terms of the fault triggering conditions. However, the fault triggering conditions could be different for different types of software systems, for example, the environment in the case of the ENV subtype of NAMs. With respect to the Linux OS, we consider the operating conditions of any external hardware devices, mountable file systems, running applications, and so on to constitute the environment. However, with respect to non-OS software systems, the OS itself would be the environment.

E. Dynamic Aspects of Error Propagation and Bug Impact

The dynamic aspects of error propagation and bug impact were mainly identified from the information provided in the bug reports. In this paper, a bug report not only contains the textual description about the failure behavior, but also includes attached files (if available) related to the reproduction, diagnosis, and fixing [6], [8]. In the Linux kernel Bugzilla, to clearly describe the failures encountered and to help developers resolve bugs, in addition to reporting the failure behavior, reporters usually attach or would be required to attach one or more of the following files, such as test cases (e.g., steps to reproduce), crash log files (e.g., *dmesg* log file, *syslog* log file), configuration, and system/device information files (e.g., *lspci* output file, *lsusb* output file). These attached files can provide partial dynamic aspects of failure behavior. For example, the call trace in a *dmesg* log file recorded a list of kernel functions executed just before a failure, which provides us the dynamic execution information before the failure. By examining the textual description and forum discussion of a bug report together with examining the attached files, it can help us better identify the type of a bug. In addition, the patches submitted by developers would also be examined to assist us in determining the bug type. For example, if a fixing patch only revised a typo in the code, we can infer that the bug can be consistently reproduced under a well-defined set of conditions. Besides, to ensure the accuracy, during the classification process, a bug report with insufficient information, such as coarse-grain textual description about the failure or insufficient file attachments, would be labeled as an unknown type (e.g., UNK). Although bug reports with insufficient information have been excluded to improve the accuracy, we recognize that using bug reports by also considering the attached files may not be able to fully identify the dynamic aspects and failure behavior, affecting the bug classification, especially the classification of the subtypes LAG, ENV, TIM, and SEQ, and the results of subsequent analyses related to the classification, such as bug type proportions and bug type characteristic analysis.

F. Evolutionary Analysis

Several factors may influence the evolution of bugs. With regard to the version evolution analysis, a new release can motivate users to migrate to the new version. Thus, bugs in the previous versions will subsequently be less reported. In addition, with regard to the temporal evolution analysis, bug reporting may

decrease as a result of fixes to the current version taking more time.

G. Fixing Time

The time required to fix a bug was computed as the difference between the reporting time and the resolution time (i.e., the time when the resolution was marked as CODE_FIX). Although this is a better approximation of the actual fixing time than the whole lifetime of the report (i.e., the closing-opening time difference) is, it does not reflect the actual time taken to fix the bug, especially if the reporter misuses the bug tracking tool. For example, a developer might report a bug only when he/she already has the fix ready. Thus, the results assume a low average impact of the potential misuse of the tracking tool. In addition, the fixing time could be impacted by the patch review process (e.g., whether the appropriate maintainers are available and how busy they are). Thus, we assume that the patch review process has the same impact on all estimated times.

H. Network Metric Analysis of Bug Types

In Section VII-A, the relative proportions were utilized to analyze the correlations between bug types and network metrics. Unlike for some software systems, for which a major version is released only after the last minor version of the previous major version, the development periods of version series of the Linux kernel usually overlap. Consequently, several version series exist at the same time [16]. In addition, several factors, such as the maintenance time, release time, and the number of users of a given version, can impact the number of bugs reported for that version. Because of these mentioned reasons, the numbers of bugs are not comparable among versions. Therefore, we used the relative proportions of the bug types in the analysis. To ensure the validity of the proportions, we selected only the versions with more than 50 bugs. In Section VII-B, the analysis of the bug type characteristics in terms of complex network metrics relies heavily on the associated bug-fixing patches because the affected functions were extracted from these patches. The correctness of the patches could thus affect the analysis results. In addition, for the network analysis, we used global metrics, i.e., closeness and betweenness, to measure the effects of error propagation. We recognize that this approach might be inaccurate. Moreover, the differences in the bug type characteristics in terms of the network metrics could be different in other software systems. In this sense, the analysis procedure and findings should be regarded as a framework for the use of complex network metrics to analyze 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 closely related work on bug characteristic analysis from the bug manifestation perspective. Then, we present several other works regarding regression bug analysis. Afterward, we introduce several studies that have analyzed the Linux OS from the complex network perspective. Finally, the differences between this paper and previous studies on software bug analysis based on the code in patches are presented and discussed.

Several papers defining the general characteristics of bugs exist, such as the IEEE Std. 1044 scheme [49], the Hewlett–Packard scheme [50], and the orthogonal defect classification (ODC) scheme [51]. ODC categorizes bugs based on several attributes, of which the most important is the bug type, which captures the semantics of the fix applied by the programmers and the bug trigger. The classification utilized in this paper is based on the bug manifestation perspective. In 1985, Gray [10] proposed a systematic abstraction of the manifestation of bugs. For easily reproducible bugs, he described them as solid or hard faults and designated them as BOHs. For transient reproducible bugs, he described them as soft or elusive bugs and designated them as Heisenbugs. Subsequently, the term MAN was proposed to represent a type of bug whose underlying causes are complex and whose manifestations are chaotic and nondeterministic [52]. To clarify the relationships among different definitions of bug types, Grottke and Trivedi [11], [12] proposed detailed definitions of BOHs and MANs. They defined MANs as the complementary opposite counterparts of BOHs, whereas Heisenbugs were defined as a subset of MANs. Moreover, depending on whether they can cause the software aging phenomenon, MANs were further classified into ARBs and NAMs. In 2013, Cotroneo *et al.* [8] provided a more detailed subtype classification for ARBs and NAMs according to the different kinds of complexity present in their fault triggering conditions.

On the basis of the above classification scheme, several studies have addressed bug classification and related bug characteristic analyses for various software systems [8], [29], [30]. Grottke *et al.* [30] explored the faults found in the onboard software for 18 JPL/NASA space missions. In that paper, among the 520 faults detected in all 18 missions, 61.4% of the faults were identified as BOHs, and 36.5% of the faults were classified as MANs. Cotroneo *et al.* [8] investigated bugs in four open-source software systems, including Linux, MySQL, HTTPD, and AXIS. They found that the proportion of MANs tended to stabilize around a constant value during the lifecycle of each of the four projects. Moreover, Qin *et al.* [29] performed a fault-trigger-based bug classification for the Android OS by examining 513 bug reports. In this paper, it was found that 31.4% of the bugs were MANs. Other studies related to the bug manifestation perspective are summarized as follows. Chandra and Chen [53] investigated faults occurring in Apache web server, GNOME desktop, and MySQL database environments. They found that 5%–14% of the faults were triggered by transient conditions, such as timing and synchronization issues. These faults naturally fixed themselves during recovery. Cotroneo *et al.* [54] 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 have focused on specific types of bugs, including ARBs [55] and concurrency bugs [56].

Here, we summarize several studies on regression bugs. Nir *et al.* [57] found that regression bugs were usually caused by bug fixes included in patches. Shihab *et al.* [58] 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 the best indicators of change risk. Khattar *et al.* [43] conducted an in-depth characterization study of regression bugs in the Google Chromium project. One interesting finding was that 51.1% of the bugs in Google Chromium are regression bugs.

Regarding the complex network analysis of software, several related studies have been performed. Large-scale software systems can be considered to be among the most sophisticated man-made systems and can be abstracted as networks [18]. An OS is a typical software product that provides an execution environment for the software that runs on the system. In 2008, Zheng *et al.* [59] proposed two new network growth models to better describe Gentoo Linux. Gao *et al.* [21] modeled the kernel directory of the Linux kernel as a complex network. In their work, it was found that with regard to the robustness of the kernel network to intentional attacks, nodes with high in-degrees providing basic services will cause more damage to the system as a whole. Wang *et al.* [22] investigated the coupling relationships among components in the Linux OS from the perspectives of topology and function, and they further studied the network impact of failures. Recently, Xiao *et al.* [23] performed an evolutionary 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 call graph were revealed.

In one class of studies, software bugs are analyzed based on the code in the patches developed to address them. Durães and Madeira [60] utilized the source code in a set of patch and diff files from several open-source projects to classify a total of 668 faults according to the ODC scheme. It was found that simple programmer mistakes were typically responsible for software failures. In addition, a state-of-the-art technique called G-SWFIT was proposed to emulate software faults and assess their impact. Fonseca *et al.* [61], [62] analyzed the source code of security patches for widely used web applications. They reported that only a small subset of the software fault types was related to security problems. Tan *et al.* [9] classified 583 bugs collected from Mozilla, Apache, and Linux into three types (i.e., memory bugs, concurrency bugs, and semantic bugs) based on their root causes by manually inspecting the source code of the corresponding patches. They reported that semantic bugs were the dominant class. In contrast to these studies, our work focuses on bug characteristics from the perspective of bug manifestation in terms of fault triggers rather than from the perspective of the developers' involvement in the coding mistakes, as in the case of classifying and analyzing bugs based on specific mistakes in the source code.

X. Conclusion

In this paper, a comprehensive empirical study of bug characteristics in the Linux OS from an evolutionary perspective was presented based on 5741 bug reports. First, we defined the bug classes and the steps performed in the manual inspection. The analysis was conducted from four perspectives, namely, bug types, regression status, fixing time, and software metrics, and yielded 22 findings and implications that can be useful to the developers and users of the Linux OS. Among these findings, some of them reinforced the existing body of knowledge on bugs (e.g., the trends and relative proportions of BOHs/MANs, the dominance of driver bugs among OS bugs, and the longer fixing time for MANs), and others of them were expected (e.g., the evolutionary trends of regression bugs and the shorter fixing time for regression bugs). To better interpret the findings, we identified two types of findings that are nonobvious. First, the results of analyzing the associations between bug types and Linux subsystems (i.e., Findings #8 and #11) and between bug types and regression statuses (i.e., Finding #13) revealed which bug types are more prone to occur in which Linux subsystems (e.g., *Drivers* bugs are more likely to be BOHs, whereas *File System* bugs are more likely to be NAMs or ARBs) and that nonregression bugs tend to be MANs. These associations can guide developers in applying different testing strategies to test different subsystems or to test for regression/nonregression bugs. In addition, the findings related to the software complexity metrics (i.e., Findings #19 to #22) revealed that the characteristics of BOHs and MANs exhibit statistically significant differences in terms of such metrics. These results can be further utilized to predict and classify MANs.

There is abundant future work to be done in this field. We present the following directions of research that deserve to be pursued.

1) Automatic classification of fault-trigger-based bug types. In our previous study [63], we utilized a deep learning method to automatically classify bugs, achieving an accuracy of 0.691. We plan to improve on this result by considering the characteristics of the different bug types in terms of network metrics.
2) Automatic representation of the relationships between regression bug reports and their causes and fixes. The results of a further analysis have revealed that the occurrence of regression bug chains is nonnegligible in Linux [64]. It would be interesting to develop a technique for formalizing the regression relationships to enable further studies of regression bugs in Linux and other software systems.
3) Use of the network metric analysis procedure to examine the bug type characteristics in other software systems.

Appendix A

Here, we illustrate the network modeling of the Linux call graph and then present the definitions of the complex network metrics selected to measure the characteristics of the bug types from a network perspective. Finally, the definitions of the integration methods for network metrics are given.

A. Network Modeling

The Linux kernel is primarily implemented based on the C programming language. The functionality realization of a software system developed in C predominantly depends on the function calls, which can commonly be represented as a call graph, as shown in Fig. 16. In this paper, we define the static call graph of the Linux kernel as a directed network $G(N, E)$, where $N = \{v_1, v_2, \ldots, v_n\}$ is a set of n nodes, each of which represents a function in the source code of the Linux kernel,

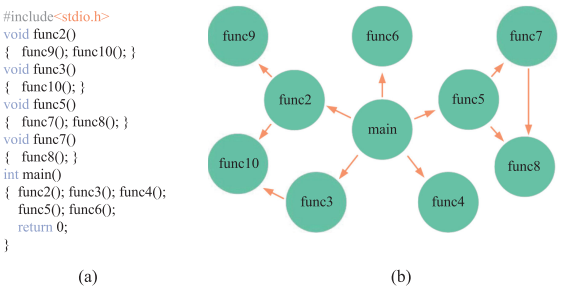

Fig. 16. Depiction of the abstraction of an example C-language program as a directed network. (a) is an illustration of the source code of the program, whose static call graph, which is depicted in (b), can be modeled as a directed network.

and $E = \{e_1, e_2, \ldots, e_m\}$ is a set of m edges, each of which, $e_i = (v_s, v_t)$ $(i = 1, 2, \ldots, m)$, denotes a call between a pair of functions, i.e., nodes v_s and v_t $(v_s, v_t \in N)$. In this paper, we model the Linux OS in the form of directed networks and focus on the largest weakly connected part of each network.

B. Network Metrics

We choose four representative complex network metrics, of which two metrics represent local properties, i.e., the degree k and the clustering coefficient C, and the other metrics represent global properties, i.e., the betweenness C_B and the closeness C_C. These network metrics are defined as follows.

1) Degree: The degree of a node in a network is the number of edges connected to it. For a directed network, a node has two types of degrees: an in-degree and an out-degree. The in-degree of a node in the Linux call graph represents the number of functions calling it, whereas the out-degree represents the number of functions that it calls. For example, as shown in Fig. 16, 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 by k_i^{in} and k_i^{out}, respectively. In addition, the undirected degree of node i is denoted by k_i and is calculated as

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

2) Clustering Coefficient: The clustering coefficient of a node measures the probability that a node's neighbors are also neighbors of each other. In the Linux call graph, a larger clustering coefficient indicates more tightly connected interactions among the neighboring functions of a node. For a directed network, the clustering coefficient of a node i is calculated as [65]

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

where the value of a_{ij} is 1 if an edge from node i to j exists and $a_{ij} = 0$ otherwise. For example, as depicted in Fig. 16, the clustering coefficient of func5 is 0.167. The clustering coefficient of the entire network is calculated as

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

3) Betweenness: The betweenness is a shortest-path-based metric representing the centrality of a node in the network. It measures the number of shortest paths that pass through the node. The expression for the betweenness of a node i is [66]

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

where σ_{st} is the total number of shortest paths from node s to t, while $\sigma_{\text{st}}(i)$ is the number of those paths that pass through node i. For a large network, C_B can be normalized as shown in A5. For example, in the network depicted in Fig. 16, the betweenness of func5 is 1

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

4) Closeness: The closeness is another measure of centrality. It is calculated as the reciprocal of the sum of the length of the shortest paths between a node and all other nodes in the network. A node with a larger closeness is more centrally located in the network and is closer to all other nodes. The closeness of a node i is defined as [66]

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

where $d(i, j)$ is the distance between nodes i and j. In addition, C_C can be normalized as shown in (A7). For example, in the network depicted in Fig. 16, the closeness of func5 is 1

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

C. Integration Methods for Network Metrics

In this paper, four methods of integrating the network metrics are presented, including the *SUM*, *AVERAGE*, *MAXIMUM*, and *MINIMUM* operations. These integration methods are defined as follows. For a given bug, suppose that the number of affected functions extracted from its corresponding patch is q. Then, the network metric representations of the bug are calculated as follows.

1) SUM: the sum of the affected functions' network metrics

$$\text{bug}_{\text{nm}}^{\text{sum}} = \sum_{p=1}^{q} \text{nm}(p), 1 \leq p \leq q \tag{A8}$$

where nm denotes a specific network metric. For example, when the clustering coefficient C is considered, the corresponding network metric of the bug is denoted by $\text{bug}_C^{\text{sum}}$. Note that nm has the same meaning in the following expressions.

2) AVERAGE: the average of the affected functions' network metrics

$$\text{bug}_{\text{nm}}^{\text{ave}} = \frac{\sum_{p=1}^{q} \text{nm}(p)}{q}, 1 \leq p \leq q. \tag{A9}$$

3) MAXIMUM: the maximum of the affected functions' network metrics

$$\text{bug}_{\text{nm}}^{\text{max}} = \max(\text{nm}(p)), 1 \leq p \leq q. \tag{A10}$$

TABLE XXVI
DETAILED VALUES FOR TABLE XXV: 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 XXVII
DETAILED VALUES FOR TABLE XXV: 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

4) MINIMUM: the minimum of the affected functions' network metrics

$$\text{bug}^{\min}_{\text{nm}} = \min(\text{nm}(p)), 1 \le p \le q. \tag{A11}$$

The average network metrics of the bugs for a version are further obtained by averaging the corresponding network metrics of all bugs present in that version. Suppose that the number of bugs present in a given version is t. The average network metrics of the bugs for that version are calculated using the following expressions.

1) *SUM*:

$$\text{version}^{\text{sum}}_{\text{nm}} = \frac{\sum_{s=1}^{t} \text{bug}^{\text{sum}}_{\text{nm}}(s)}{t}, 1 \le s \le t. \tag{A12}$$

TABLE XXVIII
DETAILED VALUES FOR TABLE XXV: k

version	$version^{sum}_{k}$		$version^{ave}_{k}$		$version^{max}_{k}$		$version^{min}_{k}$	
	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 XXIX
DETAILED VALUES FOR TABLE XXV: C

version	$version^{sum}_{C}$		$version^{ave}_{C}$		$version^{max}_{C}$		$version^{min}_{C}$	
	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

2) *AVERAGE*:

$$\text{version}^{\text{ave}}_{\text{nm}} = \frac{\sum_{s=1}^{t} \text{bug}^{\text{ave}}_{\text{nm}}(s)}{t}, 1 \le s \le t. \tag{A13}$$

3) MAXIMUM:

$$\text{version}^{\max}_{\text{nm}} = \frac{\sum_{s=1}^{t} \text{bug}^{\max}_{\text{nm}}(s)}{t}, 1 \le s \le t. \tag{A14}$$

4) MINIMUM:

$$\text{version}^{\min}_{\text{nm}} = \frac{\sum_{s=1}^{t} \text{bug}^{\min}_{\text{nm}}(s)}{t}, 1 \le s \le t. \tag{A15}$$

APPENDIX B

The detailed values of the network metrics for bugs of different types for the analysis in Section VII-B are provided in Tables XXVI–XXXI.

TABLE XXX
DETAILED VALUES FOR TABLE XXV: 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 XXXI
DETAILED VALUES FOR TABLE XXV: 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

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions for improving the paper.

REFERENCES

[1] LWN Distributions List. 2017. [Online]. Available: https://lwn.net/Distributions/
[2] Kernel.org Bugzilla Main Page. 2017. [Online]. Available: https://bugzilla.kernel.org/
[3] Coverity Scan - Static Analysis. 2017. [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," *ACM SIGOPS Operating Syst. Rev.*, vol. 35, no. 5, pp. 73–88, 2001.
[5] P. J. Guo and D. R. Engler, "Linux kernel developer responses to static analysis bug reports." in *Proc. USENIX Annu. Techn. Conf.*, 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," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 305–318, 2011.
[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 Int. Symp. Softw. Rel. Eng.*, Pasadena, CA, 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 Symp. Rel. Distrib. Softw. Database Syst.*, Los Angeles, CA, 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, pp. 107–109, Feb. 2007.
[13] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Proc. IEEE Int. Symp. Fault-Tolerant Comput.*, Pasadena, CA, USA, Jun. 1995, pp. 381–390.
[14] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *Proc. IEEE Int. Conf. Softw. Rel. Eng. Workshops*, Seattle, WA, USA, Nov. 2008, pp. 1–6.
[15] M. Grottke, D. S. Kim, R. Mansharamani, M. Nambiar, R. Natella, and K. S. Trivedi, "Recovery from software failures caused by Mandelbugs," *IEEE Trans. Rel.*, vol. 65, no. 1, pp. 70–87, Jul. 2016.
[16] 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.
[17] J. Johnson, J. Kenefick, and P. Larson, "Hunting regressions in GCC and the linux kernel," in *Proc. Linux Conf. Au.*, Adelaide, Australia, Jan. 2004, pp. 1–15.
[18] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Phys. Rev. E*, vol. 68, no. 4, Oct. 2003, Art. no. 046116.
[19] 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.
[20] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 1, Sep. 2008, Art. no. 2.
[21] 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.
[22] 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.
[23] G. Xiao, Z. Zheng, and H. Wang, "Evolution of Linux operating system network," *Physica A*, vol. 466, pp. 249–258, Jan. 2017.
[24] D. Cotroneo, R. Natella, and R. Pietrantuono, "Predicting aging-related bugs using software complexity metrics," *Perform. Eval.*, vol. 70, no. 3, pp. 163–178, Mar. 2013.
[25] 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 Int. Symp. Softw. Rel. Eng.*, Toulouse, France, Oct. 2017, pp. 101–111.
[26] The Linux Kernel Archives. 2017. [Online]. Available: https://www.kernel.org/
[27] 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.
[28] 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, Aug. 2011, Art. no. 10.
[29] 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 Int. Symp. Dependable Comput.*, Christchurch, New Zealand, Jan. 2017, pp. 135–144.
[30] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Chicago, IL, USA, Jul. 2010, pp. 447–456.
[31] P. Larson, "Testing linux with the linux test project," in *Proc. Ottawa Linux Symp.*, 2002, pp. 265–273.
[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 Int. Symp. Softw. Rel. Eng. Workshops*, 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, "Detecting kernel memory leaks [LWN.net]," 2006. [Online]. Available: https://lwn.net/Articles/187979/
[35] D. Marjamäki, "Cppcheck: A tool for static c/c++ code analysis," 2013. [Online]. Available: http://cppcheck.sourceforge.net/
[36] H. B. Mann, "Nonparametric tests against trend," *Econometrica J. Econometric Soc.*, vol. 13, no. 3, pp. 245–259, Jul. 1945.
[37] M. G. Kendall, *Rank Correlation Methods.* Oxford, U.K.: Griffin, 1948.
[38] R. C. Team *et al.*, "R: A language and environment for statistical computing," 2018. [Online]. Available: https://www.r-project.org/
[39] 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.
[40] A. Agresti, *Categorical Data Analysis*, 2nd ed. Hoboken, NJ, USA: Wiley, 2003.
[41] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser, "Dingo: Taming device drivers," in *Proc. ACM Eur. Conf. Comput. Syst.*, Apr. 2009, pp. 275–288.
[42] S. M. A. Shah, M. Morisio, and M. Torchiano, "An overview of software defect density: A scoping study," in *Proc. Asia-Pacific Softw. Eng. Conf.*, Dec. 2012, pp. 406–415.
[43] 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 Softw. Eng. Conf.*, 2015, pp. 50–59.
[44] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the Linux kernel variability model," in *Proc. 14th Int. Conf. Softw. Product Lines: Going Beyond*, 2010, pp. 136–150.
[45] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*. Boca Raton, FL, USA: CRC Press, 2003.
[46] D. E. Hinkle, W. Wiersma, and S. G. Jurs, *Applied Statistics for the Behavioral Sciences*, 5th ed. Boston, MA, USA: Houghton Mifflin, 2003.
[47] Z. Zheng and G. Xiao, "Evolution analysis of a UAV real-time operating system from a network perspective," *Chin. J. Aeronaut.*, vol. 32, no. 1, pp. 176–185, 2019.
[48] J. Pallant, *SPSS Survival Manual: A Step by Step Guide to Data Analysis Using SPSS*, 4th ed. Berkshire, U.K.: McGraw-Hill, 2010.
[49] "I. S. 1044-2009, Standard Classification for Software Anomalies," 2010.
[50] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Upper Saddle River, NJ, USA: Prentice-Hall, 1992.
[51] R. Chillarege *et al.*, "Orthogonal defect classification—A concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, Nov. 1992.
[52] E. Raymond, Ed., *The New Hacker's Dictionary*. Cambridge, MA, USA: MIT Press, 1991.
[53] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *Proc. IEEE Int. Conf. Dependable Syst. Netw.*, Jun. 2000, pp. 97–106.
[54] 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.
[55] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging analysis of the Linux operating system," in *Proc. IEEE Int. Symp. Softw. Rel. Eng.*, San Jose, CA, USA, Nov. 2010, pp. 71–80.
[56] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," *ACM Sigplan Notices*, vol. 43, no. 3, pp. 329–339, 2008.
[57] D. Nir, S. Tyszberowicz, and A. Yehudai, "Locating regression bugs," in *Proc. Springer Haifa Verification Conf.*, 2007, pp. 218–234.
[58] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Cary, NC, USA, Nov. 2012, Art. no. 62.
[59] 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.
[60] J. A. Durães and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, Nov. 2006.
[61] J. Fonseca and M. Vieira, "Mapping software faults with web security vulnerabilities," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2008, pp. 257–266.
[62] J. Fonseca, N. Seixas, M. Vieira, and H. S. Madeira, "Analysis of field data on web security vulnerabilities," *IEEE Trans. Dependable Secur. Comput.*, vol. 11, no. 2, pp. 89–100, Mar./Apr. 2014.
[63] X. Du, Z. Zheng, G. Xiao, and B. Yin, "The automatic classification of fault trigger based bug report," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops*, Toulouse, France, Oct. 2017, pp. 259–265.
[64] G. Xiao, Z. Zheng, B. Jiang, and Y. Sui, "An empirical study of regression bug chains in Linux," *IEEE Trans. Rel.*, 2019. [Online]. Available: https://doi.org/10.1109/TR.2019.2902171
[65] G. Fagiolo, "Clustering in complex directed networks," *Phys. Rev. E*, vol. 76, no. 2, Aug. 2007, Art. no. 026107.
[66] L. C. Freeman, "Centrality in social networks conceptual clarification," *Soc. Netw.*, vol. 1, no. 3, pp. 215–239, Jan. 1978.

Guanping Xiao (S'18) received the B.Sc. degree in automation from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2012 and the M.Sc. degree in control theory and control engineering from the Civil Aviation University of China, Tianjin, China, in 2015. He is currently working toward the Ph.D. degree in guidance, navigation, and control with Beihang University, Beijing, China.

He was a Visiting Ph.D. Student with the School of Software, University of Technology Sydney, Sydney, Australia in 2018. His research interests include software reliability and empirical software engineering.

Zheng Zheng (SM'18) received the Ph.D. degree in computer software and theory from the Chinese Academy of Sciences, Beijing, China, in 2006.

He is currently a Full Professor in Control Science and Engineering with the School of Automation Science and Electrical Engineering, Beihang University, Beijing, China. In 2014, he was with the Department of Electrical and Computer Engineering, Duke University, working as a Research Scholar. His research interests include software dependability, unmanned aerial vehicle path planning, artificial intelligence applications, and software fault localization.

Beibei Yin received the Ph.D. degree in guidance, navigation, and control from Beihang University, Beijing, China, in 2010.

She has been a Lecturer in control science and engineering with Beihang University since 2010. Her main research interests include software testing, software reliability, and software cybernetics.

Kishor S. Trivedi (LF'17) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology Mumbai, Mumbai, India, in 1968, and the M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign, Champaign, IL, USA, in 1972 and 1974, respectively.

He holds the Fitzgerald Hudson Chair with the Department of Electrical and Computer Engineering, Duke University (Duke), Durham, NC, USA. He has been with the Duke faculty since 1975. He is the author of a well-known text entitled, *Probability and Statistics With Reliability, Queuing and Computer Science Applications* (Prentice-Hall, 1982); a thoroughly revised second edition (including its Indian edition) of this book has been published by Wiley. The book recently has been translated into Chinese. He has also published two other books entitled, *Performance and Reliability Analysis of Computer Systems* (Kluwer Academic Publishers, 1996) and *Queueing Networks and Markov Chains* (Wiley, 1988). His latest book, *Reliability and Availability Engineering* (Cambridge University Press), was published in 2017. He has authored or coauthored more than 600 articles and has supervised 46 Ph.D. dissertations. His research interests include reliability, availability, performance, and survivability of computer and communication systems and in software dependability.

Dr. Trivedi is a Golden Core Member of the IEEE Computer Society. He is the recipient of IEEE Computer Society Technical Achievement Award for his research on Software Aging and Rejuvenation. His h-index is 97. He works closely with the industry in carrying our reliability/availability analysis, providing short courses on reliability, availability, and in the development and dissemination of software packages such as HARP, SHARPE, SREPT, and SPNP.

Xiaoting Du received the B.Sc. degree in automation from Yantai University, Yantai, China, in 2014, and the M.Sc. degree in guidance, navigation, and control in 2018 from Beihang University, Beijing, China, where she is currently working toward the Ph.D. degree in guidance, navigation, and control.

Her research interests include software reliability and data mining.

Kai-Yuan Cai received the B.S., M.S., and Ph.D. degrees in control science and engineering from Beihang University (Beijing University of Aervvonautics and Astronautics), Beijing, China, in 1984, 1987, and 1991, respectively.

He has been a Full Professor in control science and engineering with the Beihang University since 1995. He is a Cheung Kong Scholar (Chair Professor), jointly appointed by the Ministry of Education of China, and the Li Ka Shing Foundation of Hong Kong in 1999. His main research interests include software testing, software reliability, reliable flight control, and software cybernetics.