Received February 26, 2020, accepted March 10, 2020, date of publication March 30, 2020, date of current version May 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2984179

What Ruined Your Cake: Impacts of Code Modifications on Bug Distribution

JUN AI[1], HAORAN GUO[1], AND W. ERIC WONG[2,3]
[1]School of Reliability and Systems Engineering, Beihang University, Beijing 100191, China
[2]Shanghai Business School, Shanghai 201400, China
[3]Department of Computer Science, The University of Texas at Dallas, Richardson, Texas 75080, USA

Corresponding authors: Jun Ai (aijun@buaa.edu.cn) and W. Eric Wong (ewong@utdallas.edu)

ABSTRACT In the current decade, software systems have been more intensively employed in every aspect of our lives. However, it is disappointing that the quality of software is far from satisfactory. More importantly, the complexity and size of today's software systems are increasing dramatically, which means that the number of required modifications is also increasing exponentially. Therefore, it is necessary to understand how function-level modifications impact the distribution of software bugs. In addition, other factors such as a function's structural characteristics as well as attributes of functions themselves may also serve as informative indicators for software fault prediction. In this paper, we perform statistical methods and logistic regression to analyze the possible factors that are related to the distribution of software bugs. We demonstrate our study from the following five perspectives: 1) the distribution of bugs in time and space; 2) the distribution of function-level modifications in time and space; 3) the relationship between function-level modifications and functions' fault-proneness; 4) the relationship between functional attributes and functions' fault-proneness; and 5) the relationship between software structural characteristics and functions' fault-proneness.

INDEX TERMS Bug, bug distribution, defected software systems, modifications, open source projects.

I. INTRODUCTION

Currently, we are in an era in which software plays an essential role in everyday life. The complexity and scale of software systems have increased tremendously, especially for those safety- or security-critical systems. According to [1], in the past 40 years, the size of software has increased exponentially. Yet the disappointing fact is that various quality-related issues still exist in the entire life cycle of software development. According to a National Institute of Standards and Technology report [2], software bugs cost the U.S. economy an estimated $59.5 billion annually. More importantly, due to the size and complexity of modern software systems, the situation even deteriorates, and there is also pressure from tighter schedules and reduced development budgets. In addition, the report also states that more than one-third of the costs would be eliminated if more proper quality assurance approaches were applied.

Software has been widely applied in various complex systems in every aspect of our lives. More importantly, the application of software in safety-critical areas, such as medicine and nuclear power generation, has a great impact on people's daily lives. When utilized in these and other, similar industries, software systems are usually used to control the behaviors of electromechanical components and monitor the interactions between them to ensure the smooth operations of the systems. Ideally, software should operate as expected and not contribute to hazards [3]. However, with the increasing complexity and size, software systems are becoming unexpectedly large with advanced algorithms, interactions with external systems, as well as the huge amounts of data transmitted. For example, if the system turns into an unsafe state or the operations performed by engineers directly or indirectly lead the system into hazard, the outcome can be catastrophic. In a recently published paper [4], the authors examined 104 accidents in recent decades to address the roles bugs played in those accidents and demonstrate lessons learned to improve our development practices in order to prevent similar accidents from happening in the future.

Boehm [5] carefully analyzed the costs to fix potential errors or bugs in different phases of the software life cycle and suggested that the later a bug is detected, the costlier it will be. In other words, it is optimal that developers can detect

The associate editor coordinating the review of this manuscript and approving it for publication was Steve Li.

the potential defects in their systems under development as early as possible. Hence, it is beneficial to identify fault-prone software modules in order to alleviate the cost of software debugging and bug fixing ([6]–[9]). Encouraging results in prior research indicate that it is possible to predict which modules are likely to be locations of defect occurrence using Random Forest ([10], [11]), Neural Network ([12], [13]), Support Vector Machines ([14], [15]), Logistic Regression ([16]–[18]), and Naive Bayes ([19]–[21]). In addition, there has been research on defect classification ([22], [23]), just-in-time defect prediction [24], and so on.

However, shortcomings still exist with respect to these existing approaches. First, most of these approaches assume functional-level modifications to be the indicators of bug locations. Undoubtedly, bugs are the direct outcomes of these modifications; however, other issues with respect to software systems can also serve as the reasons why specific modifications are very likely to induce bugs. For example, suppose that the complexity of one component is significantly higher than its normal acceptable level. In this case, the likelihood that the modifications towards this component introducing bugs also increases, not only because of its own complexity but also due to the complex interactions among the component and other components within the system.

Another issue is that no study focused on exploring how closely functional-level modifications are related to software bug distributions currently exists. To what degree can these modifications serve as an informative indicator of bug locations? Do all the modifications towards different functions have the same likelihood of inducing bugs? These questions all remain unanswered. In addition, most studies merely focus on file and class level, so it is impossible to provide more accurate prediction results for bug locations.

Many studies focus primarily on the analysis of code level measurements. However, software development is a dynamic process, and changes along with this process can have either positive or negative impacts on the distribution of software defects. Although researchers are aware of this, a close examination of process-related information from new perspectives and a careful evaluation of their relationship with fault proneness are still needed [25].

To solve these issues, we propose a thorough study on various factors that could have an impact on the distribution of software bugs. There are three novel aspects of our study.

First, we introduce the concept of software evolution into the area of software fault prediction. Unlike existing studies, we take the evolution of software systems into account in the process of determining influential factors of fault distribution. Instead of predicting bug locations based on one or several specific versions, we analyze more than 1,000 consecutive versions of open source projects to understand the impacts of these factors as well as how they change during the entire development process.

Second, we closely analyze the relationships between functional-level factors and bug distributions. Unlike existing metrics focusing on either class or file level, we introduce different functional-level factors that can be used as indicators of bug distributions.

Third, we also analyze how structural attributes of software systems can relate to fault distributions. Three functional-level attributes as well as 24 network metrics are included in our study to represent the software structures and then serve as the candidate factors for predicting software fault distributions.

The remainder of the paper is organized as follows: Section II describes in detail the research questions we are trying to solve. What follows in Section III contains the experimental setups used in this study. In Section IV, we introduce the evaluation methods used to demonstrate the correlation relationships. Our results and discussions are presented in Section V, followed by the threats to validity in Section VI. Other existing studies related to this paper are discussed in Section VII, while we draw our conclusions in the final section.

II. RESEARCH QUESTIONS

In this paper, we carefully examined the following five research questions to elicit informative and influential factors of software fault distribution as well as the degree to which they are related to fault proneness.

A. WHAT IS THE DISTRIBUTION OF DEFECTS IN TIME AND SPACE?

In this paper, we represent software from two unique dimensions: time and space. On one hand, we use time to represent the evolution process of software projects. As introduced in Section I, we consider software development as a continuous process and recognize that fault distribution may also vary during this process. For example, with respect to the 492 versions of Nginx, we analyze how bugs distribute within these versions, such as how many bugs each version contains and how the number of bugs changes during the software project development process. On the other hand, space is used to evaluate how bugs are distributed among functions within software projects. Unlike most existing projects, which only focus on file- or class-level fault distributions, we examine how many bugs are contained within distinct functions. Exploring the distribution of software defects in time and space helps us better understand software defects and serves as guidance in subsequent research.

B. WHAT IS THE DISTRIBUTION OF MODIFICATIONS IN TIME AND SPACE?

The terms time and space here are the same as in RQ 1. In this paper, the functional-level modifications are divided into six types, and the distribution of them on the releases (or versions) and the associated functions are studied respectively. To solve RQ 2, the distribution of modification types is carefully examined, and the results will serve as the fundamentals of the remaining research questions.

C. WHAT IS THE CORRELATION BETWEEN FUNCTION-LEVEL MODIFICATIONS AND THEIR CORRESPONDING FAULT-PRONENESS?

Though we acknowledge that modifications of software source code may induce bugs, it is still unknown to what degree these two aspects are related. Which type(s) of modifications are more likely to induce defects? To what degree are these modifications related to functions' defect-proneness? With the answers to these questions, we will have more detailed and effective indicators for software fault prediction analyses.

D. WHAT IS THE CORRELATION BETWEEN ATTRIBUTES OF FUNCTIONS AND THEIR CORRESPONDING FAULT-PRONENESS?

Instead of considering all functions equal, we elicit different attributes of functions to represent the characteristics of functions in a more demonstrative way. For example, in this paper, the duration of a software function is defined as how many consecutive versions have the specific function. Are the functions with longer or shorter durations more likely to contain bugs? We also examined several other attributes of functions and analyzed their relationships with functions' fault-proneness.

E. WHAT IS THE CORRELATION BETWEEN SOFTWARE STRUCTURAL CHARACTERISTICS AND THE FAULT-PRONENESS OF FUNCTIONS?

Software systems have been widely employed in various areas of our lives, which results in different software having unique structural characteristics. Instead of treating different software systems the same, we also performed experiments to demonstrate the correlation between the structural characteristics and functions' fault-proneness. This information can also help us better customize the fault-proneness analysis approach to produce more effective and accurate prediction results.

III. EXPERIMENTAL SETUPS

In the following section, we first briefly introduce the subject open source projects that have been used in this paper.

A. SUBJECT PROJECTS

In this paper, we analyzed four open source projects from GitHub. Table 1 presents the detailed information of these projects. The Total Functions and Total Calls in Table 1 are calculated based on the most recent version.

Nginx [26] is a free and open source project that provides a HTTP server, a mail server, and a generic TCP/UDP proxy server. The software has been widely used in a large number of websites and is known for its stability and performance as well as its ease of usage and configuration.

Gedit [27] is a small and lightweight UTF-8 text editor that is part of the GNOME environment. The project uses the lastest GNOME and GTK+ libraries with support for Drag and Drop (DnD) from Nautilus (the GNOME file manager), the use of the GNOME help system, the Virtual File System GVfs, and the GTK+ print framework.

Nagioscore [28] is the core of an open source project Nagios from GitHub, which can be utilized to monitors systems, networks, and infrastructures. In addition, the project also provides supports for monitoring and alerting services for servers, switches, and applications.

Redis [29] is another open source project that can be used as cache or databases. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, and geospatial indexes with radius queries.

TABLE 1. Basic information of projects.

Software	Versions	Commits	Total Functions	Total Calls
Nginx	492	6271	2239	5839
Gedit	356	11262	2917	3927
Nagioscore	101	2816	2319	6259
Redis	201	6494	4364	11831

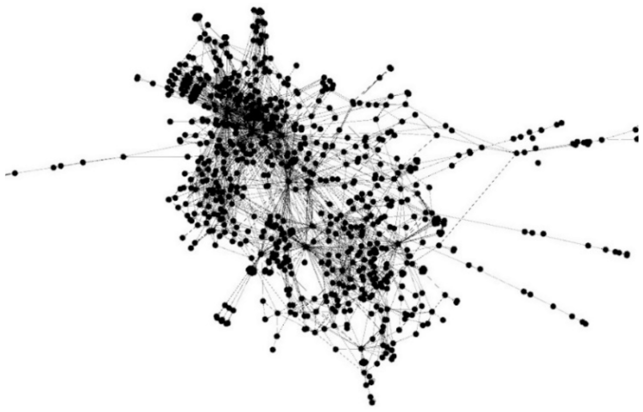


FIGURE 1. Network of Nagioscore V1.0.

B. SUBJECT PROJECTS

In this subsection, we describe the procedures used in this paper to collect data related to bugs, functions, and software structural characteristics.

1) CONSTRUCTION OF SOFTWARE NETWORKS

In this paper, we use Function Call Graphs (FCGs) to form software complex networks, which helps to more explicitly illustrate the structures of software systems. The networks are then used as foundations for the follow-up analyses. An example of FCG based on Nagioscore V1.0 is presented in Figure 1. In the FCG, we use nodes to represent unique functions, while edges stand for the calling relationships between pairs of functions. Necessary information to establish the FCGs is extracted using Doxygen [30], a de facto standard tool for generating documentation of multiple programing languages, including C, C++, Java, Python, and so on.

2) EXTRACTION OF DEFECT-RELATED INFORMATION

In order to locate bugs in open source projects, information related to commits for fixing bugs is extracted from the issue tracking system. In our study, we first collect information related to bug fixing commits. With these commits, a list of functions that are likely to contain bugs can be generated. Then, we try to pinpoint the exact location of each bug as well as their introduction time/version. In our study, we use the issue tracking system provided by GitHub and the BugTracer implemented by our team to help us retrieve information related to bugs. Details of the extraction process are described in the following steps:

Step 1. Similar to previous works ([31], [32]), we use keywords such as "fix," "bug," "defect," "fault," and "patch" in the commit comments to filter the commits that are related to bug fixing. These commits are closely related to bugs, since their goal is to fix potential bugs in their previous releases. By doing this, potential bugs within each version can be collected. For instance, consider the following comment of a commit: "Fixing the segmentation faults during the execution of garbage collections." It is obvious that this commit is closely related to a bug, and the modified code should be responsible for fixing this bug.

Step 2. With Step (1) performed, a number of records that contain the information related to bugs would be stored in our database. In addition, the versions with the code introduced to fix bugs can also be identified, and we call them "bug-fixing versions," denoted as $U_{fix} = \{V_{fix-1}, V_{fix-2}, \ldots, V_{fix-n}\}$.

With the bug-fixing version set achieved in Step 2, we then determine the locations of bugs within versions of software systems based on commits that corrected a failure in the program's source code. Ideally, with respect to each bug, we can get two unique versions, V_{intro} and V_{fix}, representing the version introducing the corresponding bug and the version fixing it, respectively.

Though developers evidently marked these version as bug-fixing versions, it is possible that the modifications made in this version are not purely related to fixing a specific bug. In other words, the faulty statements should be isolated from other modified statements that focus on implementing features. Therefore, the procedures listed below are used to derive the version set $\{V_{intro}, V_{fix}\}$ and the corresponding modified functions set $\{F_{fix-1}, F_{fix-2}, \ldots, F_{fix-k}\}$ to fix a particular bug. A similar approach has also been utilized in [33].

Step 3. Locate potential bug-fixing statements. With respect to a specific version in U_{fix}, say V_{fix-m}, which fixes the mth bug, we compare the current version with its prior version and locate statements that are modified between the two versions. For a specific bug-fixing version, the statements form another set, denoted as $S_{fix-m} = \{S_{fix-1}, S_{fix-2}, \ldots, S_{fix-t}\}$.

Step 4. Determine the last versions that modify each statement in S_{fix-m}. These versions are denoted as $V' = \{V_{modified-1}, V_{modified-2}, \ldots, V_{modified-l}\}$. Also, $Num_{modified-i}$ represents the number of statements in S_{fix-m} most recently modified by the ith version in V', where $1 \leq i \leq l$.

Step 5. Rank the versions in V' based on the their $Num_{modified-i}$ in descending order. V' will be updated as $\{V_{ordered-1}, V_{ordered-2}, \ldots, V_{ordered-l}\}$.

Step 6. Add versions in V' to V_{final} one by one until the sum of $Num_{modified}$ of versions in V_{final} is larger than 90% of the number of statements in S_{fix-m} achieved in Step 2. Versions in V_{final} act as candidates for V_{intro}.

Step 7. Manually inspect by determining which statements in S_{fix-m} are the root cause of the bug, and select versions in V_{final} that lastly modify these statements. The following three scenarios should be considered: (a) If only one version is selected, this version will be considered as the Vintro for the mth bug. (b) If multiple versions are selected, the most recently released version is regarded as Vintro. (c) If none of the versions in Vfinal is selected, we abandon the mth bug.

With the bugs locating and introducing versions, we can determine the faulty functions of each version during the entire life cycle of software systems as well as the introduced and fixed time of each bug.

3) EXTRACTION OF INFORMATION RELATED TO FUNCTION-LEVEL MODIFICATIONS

With respect to the function-level modifications, we focus primarily on the following six types of modifications:

Addition, denoted as *AD*. Addition of newly implemented functions can be observed frequently during the development of software systems. We determine the addition of new functions by comparing the network of a version with its previous version. If a specific node does not exist in the current version, we define it as a newly added function, and the *AD* value for this function is one; otherwise, the value is zero.

Modified Code, denoted as *MC*. This category of modification is used to represent whether the code within a function has been modified when compared to its prior version. In our study, we extract the code within each function and evaluate the corresponding md5 value. If the md5 values remain the same between two consecutive versions, it represents that the function is not modified in these two versions, and the *MC* value is zero. Otherwise, the contents of the function have been modified, and the *MC* value is one.

In-Degree Change, denoted as *Deg-In*. Once the function calls from other functions to a specific function, the in-degree of this function changes. In this study, we use a Boolean value to represent whether this set between two consecutive versions has changed. In other words, *Deg-In* equals one if the set of functions calling the corresponding function changes; otherwise, *Deg-In* equals zero.

Out-Degree Change, denoted as *Deg-Out*. Contrary to *In-Degree* change, for a particular function, *Deg-Out* represents changes in the set of functions called by the corresponding function. Similar to *Deg-In*, we also use a Boolean value to represent it.

Degree Change, denoted as *Deg-Ch*. *Deg-Ch* represents when either the *Deg-In* or *Deg-Out* is one. If one of the two measurements is one, it means that the calling relationships between this function and other functions are changed. Therefore, the *Deg-Ch* should also be one.

Overall Modification, denoted as *OM*. If any of the previously discussed modifications exists, the *OM* value for a specific version is one. Otherwise, the *OM* value is zero.

4) EXTRACTION OF FUNCTION ATTRIBUTES

In the process of software development, when analyzing a newer release, we should not only consider the modifications of the current version, but also consider the influence of the historical changes made to the functions. Therefore, in addition to different types of modifications, we also consider several other attributes with respect to each function. The following three unique types of attributes, namely *Time*, *DiffCount*, and *HisBug*, are included in our study and are described in more detail as follows:

Time: This attribute measures the duration during which a function has existed in the project. The *Time* of a particular function equals the total number of versions from the version in which it is introduced to the version where it is deleted. If the function exists in the most recent version of the project, this version is used as the last version of the function's duration.

DiffCount: With the function-level modification information collected, we use *DiffCount* to represent the total number of function-level modifications that have been made to a particular function during the life cycle of the project. From a mathematical point of view, the *DiffCount* of a function equals the sum of the *AD*, *MC*, *Deg-In*, and *Deg-Out* of this function in all previous versions.

HisBug: In this paper, we also consider the factor of whether a function contains bug(s) in any of its previous versions. Our intuition is that a function that had bugs in the past may also contain bugs in the subsequent releases. For example, fixing one bug may introduce additional new bugs in the same functions.

5) NETWORK MEASURES

For each function node, we use an ego network and a global network to analyze its structural characteristics from two angles: local and global. For the ego network, its nodes include a sole central node ("ego") and all the nodes that the ego connects directly ("alters"), and its edges include the edges between the ego and alters and the edges among the alters. The ego network focuses on the study of the characteristics of a single node in the local network. At the same time, each alter and its alters can also form an ego network, and the ego network of all nodes is combined to form the complete global network. The global network contains all the nodes and edges.

TABLE 2. Ego and global measures used in this study ([38], [39]).

Ego Network Measures:12	
Size	number of nodes that ego is directly connected to
Ties	number of directed ties corresponds to the number of edges
Pairs	number of unique pairs of nodes, i.e., Size×(Size−1)
Density	% of possible ties that are actually present, i.e., Ties/Pairs
nWeakComp	number of weak components in the ego network
pWeakComp	number of weak components normalized by size
2StepReach	number of nodes ego can reach within two steps normalized by Size
2StepPct	2stepreach/(N-1)
ReachEffic	2StepReach normalized by the sum of the size of the ego's every neighbor's ego network
Broker	number of pairs not directly connected to each other
nClosed	the number of closed triads ego is involved in
Global Network Measures:12	
Degree	number of nodes adjacent to a given node
Out_Degree	number of edges pointing out of the node
In_Degree	number of edges pointing in to the node
Ripple_Degree	number of functions that the node can call directly or indirectly
Closeness	sum of the lengths of the shortest paths from a node to all other nodes
Betweenness	measures how many shortest paths between other entities it occurs
PageRank	From Google web page sorting algorithm based on the link, which is not only influenced by the number of other nodes of the pointing node, but also related to the importance of other nodes pointing to the node.
Clus_Coef	measures the density of a node's open neighborhood
K_Cores	the maximal subgraph with minimum degree at least k
Eigenvector	assigns relative scores to all nodes in the dependency graphs
Efficiency	normalizes EffiSize to the total size of the network
Constraint	measures how strongly an node is constrained

The original table was published in [34] and reproduced as follows

Similar to [34], we use the following 24 network measures shown in Table 2, including 12 ego network measures and 12 global network measures, to analyze the relationship between software structural characteristics and defects from two unique perspectives: global and local. Based on the FCGs, a widely-used library Networkx [35] is applied to help us obtain the measurements of the software network structures.

IV. EVALUATING METHODOLOGIES

In this section, we will describe in detail all the methods used to demonstrate whether a specific factor is related to the distribution of bugs. In Section A, we discuss the basic knowledge with respect to the power law distribution. What follows in Section B is a conditional probability-based method to analyze the correlation between modifications and bug distributions. In Section C, the logistic regression is introduced.

A. POWER LAW DISTRIBUTION

In statistics, the power law is a functional relationship between two quantities, where a relative change in one quantity results in a proportional relative change in the other quantity, independent of the initial size of those quantities; one quantity varies as a power of another. For example, we use the following formula to calculate the area of a circle:

$$S = \pi r^2$$

Suppose that the radius r is doubled; the area S is therefore multiplied by a factor of four. In this case, we conclude that S and r conform to the power law distribution.

In this study, we find that the distribution of bugs and function-level modifications conforms to the power law distribution, which can be defined as follows:

$$p(x) \propto x^{-alpha}, \quad \text{for } x \geq xmin$$

The above definition can be utilized to verify if the distribution under analysis is consistent with the power law distribution. In the above equation, $p(x)$ represents the number of bugs or the value of a particular type of modifications on software element (function or version) with an index of x. Then, Plfit [36], which is a Python implementation of a power law distribution fitter, is used to check the consistency with respect to the power law distribution. Plfit examines if the power law is an acceptable fit based on the "p-value" calculated using the Monte-Carlo test. If the p-value is higher than 0.1, we would say that the data conforms to the power law distribution [37].

B. CONDITIONAL PROBABILITY-BASED METHOD

The following method is also employed to help us further explore the correlation between a specific type of modifications and the faulty functions:

For each type of modification, we divide the functions in a software system into four disjoint groups, which can be represented as $Func_{XB}$, $Func_{X\bar{B}}$, $Func_{\bar{X}B}$, and $Func_{\bar{X}\bar{B}}$, respectively. $Func_{XB}$ indicates the groups of faulty functions that have one or several modifications of type X; $Func_{X\bar{B}}$ contains the functions that do not contain any bugs but have been modified by X modifications; $Func_{\bar{X}B}$ possesses faulty functions with no X modifications; and $Func_{\bar{X}\bar{B}}$ represents the groups of correct functions with no X modifications.

With the four groups divided, $N_{X,B}$, $N_{X,\bar{B}}$, $N_{\bar{X},B}$, and $N_{\bar{X},\bar{B}}$ are used to represent the number of functions within each group. In addition, N represents the number of functions included in a specific version. We have the following:

$$\begin{cases} N_X = N_{X,B} + N_{X,\bar{B}} \\ N_{\bar{X}} = N_{\bar{X},B} + N_{\bar{X},\bar{B}} \\ N_B = N_{X,B} + N_{\bar{X},B} \\ N_{\bar{B}} = N_{X,\bar{B}} + N_{\bar{X},\bar{B}} \end{cases}$$

We define the P_X as the proportion of newly-added functions in the total functions, which can be calculated with the following formula:

$$P_X = \frac{N_X}{N}$$

Also, we use $P_{X,B|B}$ to demonstrate the percentage of faulty functions that have at least one modification of type X:

$$P_{X,B|B} = \frac{N_{X,B}}{N_B}$$

Following the same evaluation criteria utilized in [40], if $P_{XB|B}$ is significantly greater than P_X, the bugs are more likely to be in functions with X modifications.

Take the addition of new functions as an example. Without loss of generality, assume that the $P_{AD,B|B}$ of a specific version is 0.85, and it has a P_{AD} value of 0.25. These two numbers indicate that 85% of bugs in the software system are actually from those 25% newly added functions.

C. LOGISTIC REGRESSION

The following method is also employed to help us further explore the correlation between a specific type of modifications and the faulty functions:

In our study, we used logistic regression to help us determine if a specific network measure discussed in Section III B serves as a predictive variable for fault-proneness. Logistic regression is a widely-used classification model that can be utilized to determine which of two values the dependent variable should be. Therefore, it is appropriate for our study, since a function of a software system either contains a bug or is bug-free.

When performing logistic regression, an important aspect that needs to be considered is the identification of influential observations, which can change the regression results [41]. If a small subset of influential observations is included, it could have a disproportionate impact on the estimation. In other words, the model estimates are more likely to be calculated based on these outliers than the rest of data [41].

Similar to other studies [34], we applied the method described in [41] to exclude influential observations (whose Cook's distance is equal to or larger than 1 [42]) to ensure an accurate analysis of the correlation between various factors and their abilities to predict fault-proneness.

Two different types of logistic regression models exist. The first is the univariate logistic regression model, which can be used to determine the individual impacts brought by the independent variable on the dependent variable. Another type of model is the multivariate logistic regression model, which tries to figure out the combinatorial effects brought by multiple independent variables.

Obviously, the univariate logistic regression is a better fit for our study to identify the relationships between each software structural measure and functions' fault-proneness.

The univariate logistic regression model is based on the following:

$$\Pr(Y = 1|X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}$$

where X is the independent variable, and Y is the dependent variable, which can only be equal to either zero or one. β_0 and β_1 are the regression coefficients and can be estimated by maximizing the following:

$$\prod_{i=1}^{n} \Pr(y_i = 1|x_i)^{y_i}(1 - \Pr(y_i = 0|x_i))^{1-y_i}$$

where x_i is the ith value of independent variable X, and y_i is the corresponding value of dependent variable Y.

With the logistic regression model, we use the following statistics to evaluate whether each network measure is closely related to bug location:

- *p-value*. The *p-value* is the most widely-used statistic to determine if the dependent variable Y significantly depends on the independent variable X. The null hypothesis is $H_0 : \beta_1 = 0$, which means the independent variable X has no impact on the dependent variable Y. If the null hypothesis is rejected, it represents that Y is dependent on X. Usually, we set the cutoff for significance as 0.05. We reject the null hypothesis if we achieve a p-value smaller than 0.05 and conclude that Y is dependent on X.
- *Odds ratio*. The odds ratio is another commonly used statistic to quantify the correlation between two variable X and Y. The independent variable is considered to have no impact on the dependent variable if the odds ratio is equal to 1. A positive relationship between X and Y can be established if the odds ratio is greater than 1, while an odds ratio smaller than 1 indicates a negative relationship.

V. RESULTS AND DISCUSSIONS

In this section, we will discuss the results regarding the five research questions proposed previously.

A. RQ1: WHAT IS THE DISTRIBUTION OF DEFECTS IN TIME AND SPACE?

The cumulative distributions of software bugs with respect to software versions and functions for the projects under analyses are shown in Figure 2. In this part, we only include the figures for the distribution of bugs based on data collected from Redis and Nginx. However, similar results can also be observed in figures for other software systems used in this study.

In the figures, the X-axis represents the number of bugs in each version (see Figure 2(a) and (b)) or in each function (see Figure 2(c) and (d)), while the Y-axis stands for the cumulative percentage of versions or functions, respectively. For example, the data point (2, 0.267) in Figure 2(a) represents that $(0.267 \times 100\% =)$ 26.7% versions of Redis contain at least two bugs. Similarly, with respect to Nginx, the data point (3, 0.210) in Figure 2(b) demonstrates that $(0.210 \times 100\% =)$ 21% versions of Nginx have three or more bugs.

Considering from the *time* point of view, it can be observed from Figure 2(a) and (b) that most of the versions are bug-

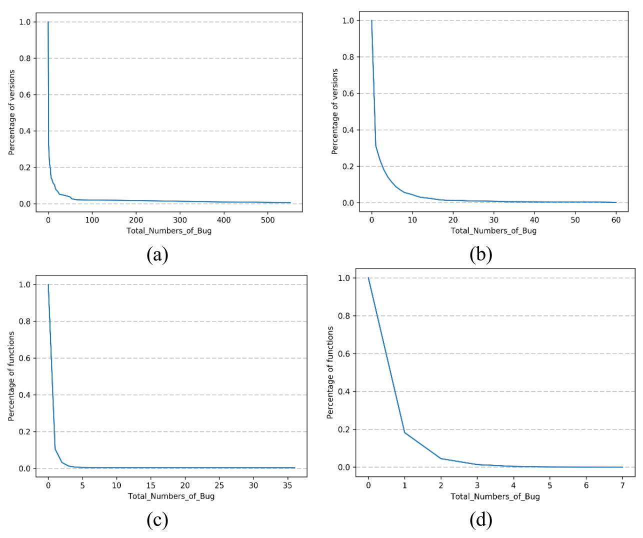


FIGURE 2. Bug distribution in *time* and *space*. (a) Redis in *time*; (b) Nginx in *time*; (c) Redis in *space*; (d) Nginx in *space*.

free. For example, with respect to Redis in Figure 2(a), the y value for $x = 1$ is 0.325, which represents that only $(0.325 \times 100\% =)$ 32.5% of all versions of Redis contain at least one bug and the remaining 67.5% of versions are bug-free. Similarly, as illustrated in Figure 2(b), the y value for $x = 1$ is 0.317, which means that $(0.317 \times 100\% =)$ 31.7% of all versions of Nginx are defected, while the rest are bug-free. Consequently, our first observation is that bugs are more likely to exist in a relatively small portion of versions.

When it comes to the distribution of bugs in *space*, similar findings can be observed. It is noteworthy that most bugs concentrate on a relatively small percentage of functions, instead of being evenly distributed in all functions. As demonstrated in Figure 2(c), the data point (2, 0.032) represents that only $(0.032 \times 100\% =)$ 3.2% of all functions contain at least two bugs. Moreover, when $x = 1$, the y value is 0.105, which means that $(1 - 0.105 \times 100\% =)$ 89.5% of all functions in Redis are bug-free. Similar observations can also be made in Figure 2(d). For instance, the point (1, 0.182) indicates that only $(0.182 \times 100\% =)$ 18.2% of all functions of Nginx are defected. Therefore, another observation is that bugs are more likely to exist in a relatively small portion of functions, instead of being distributed in all functions.

In addition, from Figure 2(c) and (d), we can also conclude that the distribution of bugs roughly conforms to the Pareto principle ([43], [44]), which indicates that a small percentage (20%) of software modules contains the majority (80%) of faults. For example, the data point (1, 0.105) in Figure 2(c) represents that of all the versions of Redis, bugs are concentrated in only $(0.105 \times 100\% =)$ 10.5% of the functions.

In addition, statistical validation is also performed to verify that the distribution of bugs in *time* and *space* fits to the power law distribution. The fitting parameters are shown in Table 3. According to [37], if the p-value is greater than 0.1, the data sets satisfactorily fits the power-law distribution. As can be

TABLE 3. Fitting parameters of bug distribution in *time* and *space*.

Software	Fitting parameters in *time*			Fitting parameters in *space*		
	alpha	*xmin*	*p-value*	*alpha*	*xmin*	*p-value*
Nginx	2.313	3	0.74	4.373	3	0.15
Gedit	2.158	7	0.47	4.489	4	0.58
Nagioscore	2.490	5	0.99	3.035	4	0.13
Redis	1.780	4	0.12	3.493	3	0.20

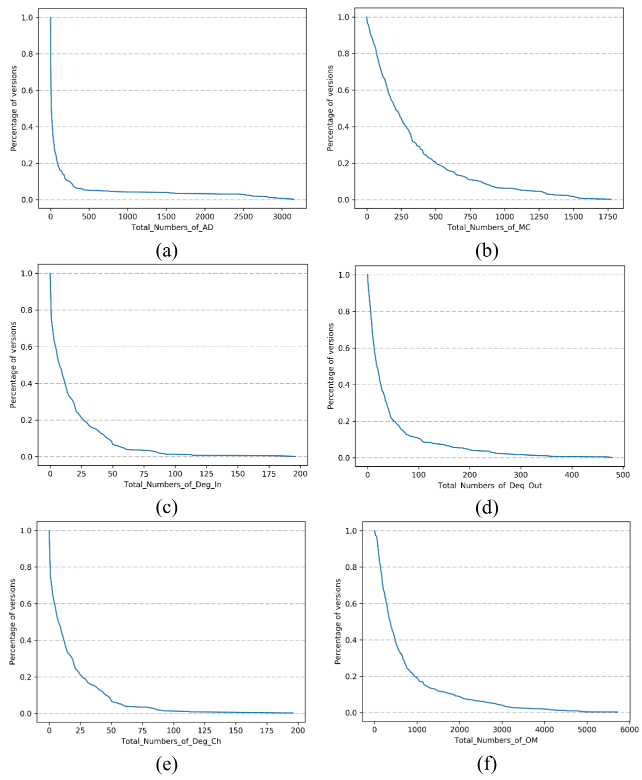


FIGURE 3. Modification distribution in *time* for Gedit. (a) *AD*; (b) *MC*; (c) *Deg-In*; (d) *Deg-Out*; (e) *Deg-Ch*; (f) *OM*.

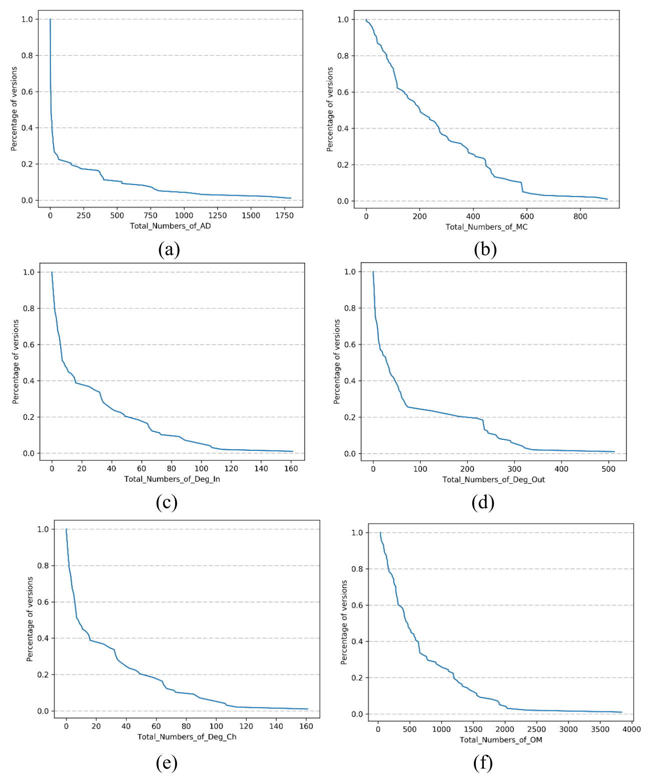


FIGURE 4. Modification distribution in *time* for Nagioscore. (a) *AD*; (b) *MC*; (c) *Deg-In*; (d) *Deg-Out*; (e) *Deg-Ch*; (f) *OM*.

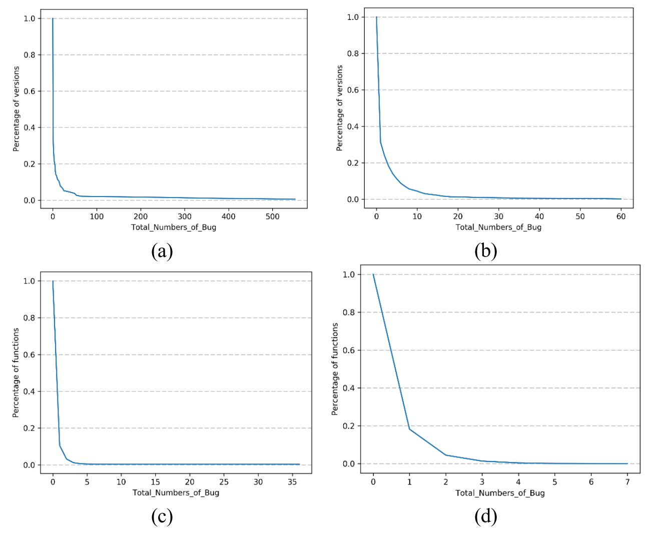


FIGURE 5. Modification distribution in time for Nginx. (a) Deg-In; (b) Deg-Out; (c) Deg-Ch; (d) OM.

observed from Table 3, all p-value for the eight scenarios are greater than 0.1, which means that the distribution of bugs in *time* and *space* can be fitted using the power law distribution.

In summary, with respect to RQ1, our observations are as follows:

- *Bugs are more likely to exist in a relatively small portion of versions*
- *Bugs are more likely to exist in a relatively small portion of functions, instead of being evenly distributed in all functions*
- *The distribution of bugs roughly conforms to the Pareto principle*
- *The distribution of bugs in time and space conforms to the power law distribution*

B. RQ2: WHAT IS THE DISTRIBUTION OF MODIFICATIONS IN TIME AND SPACE?

The distribution characteristics of bugs and the distribution characteristics of modifications are also discussed. Figure 3 to Figure 5 illustrate data regarding the distribution of different types of modifications. Here, we only present the data for Gedit, Nagioscore, and part of Nginx. However, we would like to emphasize that similar phenomena can also be observed for other projects used in this study.

In these figures, the X-axis represents the number of a specific type of modification in each version (see Figure 3 and Figure 4) or in each function (see Figure 5), while the Y-axis stands for the cumulative percentage of versions or functions, respectively. For example, the data point (50, 0.248) in Figure 3(a) represents that ($0.248 \times 100\% =$) 24.8% versions of Gedit contain more than 50 *AD* modifications.

If we consider the distribution from the *time* point of view, it can be observed that all the curves in Figure 3 and Figure 4 roughly follow similar trends:

- For most of the curves, the beginning part is relatively steep. This can be interpreted as the fact that the number of modifications for most of the versions is relatively small. For example, as demonstrated in Figure 3(a), when $x = 10$, the y value is 0.515; however, when x increases to 100, the y value drops dramatically to 0.206. This indicates that for the majority of versions, the number of modifications is relatively small.
  However, there do exist exceptions. Compared with other figures, the first part of the curve in Figure 4(b) exhibits a linear-style decrease. In this case, the MC modifications with respect to Nagioscore tend to be more evenly distributed in all versions when compared with all other modification types.
- The remaining parts of these curves resemble each other and show similar trends. After the first part discussed above, the remaining parts follow a gentler decreasing pattern. For example, as demonstrated in Figure 4(d), the data point (74, 0.255) represents that ($0.255 \times 100\% =$) 25.5% of versions of Nagioscore contains more than 74 *Deg-Out* modifications; when the x value increases to 220, the y value only decreases by less than 13%, from 0.255 to 0.194.

Therefore, we can conclude that a large number of modifications are deployed on a small portion of versions.

Similar observations can also be noted from the *space* point of view. Without loss of generality, we only include the figures for Nginx, as illustrated in Figure 5. Except for Figure 5(d), all other figures exhibit similar characteristics as those in Figure 3 and Figure 4, which means that a large number of modifications focus on a small number of functions. Though Figure 5(d) is slightly different from the other three figures, we can still achieve the same conclusion.

Statistical validation is also performed to verify that the distribution of modifications in *time* and *space* fits the power law distribution. The fitting parameters are shown in Table 4. As can be observed in Table 4, all p-values for *AD*, *Deg-In*, *Deg-Out*, and *Deg-Ch* are greater than 0.1, which means that the distribution of modifications in *time* and *space* can be fitted using the power law distribution. However, based on Figure 3 to Figure 5, *MC* and *OM* do not conform to the power law distribution significantly. Therefore, *MC* and *OM* are not included in Table 4.

In summary, with respect to RQ2, our observations are as follows:

- *Modifications are more likely to concentrate on a relatively small number of versions*

TABLE 4. Fitting parameters for modifications power law distribution.

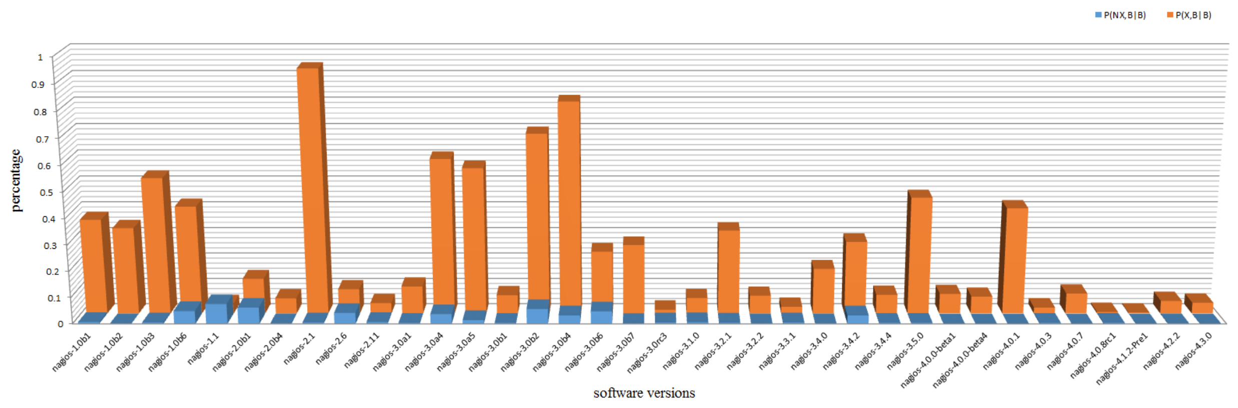
Software		Fitting parameters in *time*			Fitting parameters in *space*		
		alpha	*xmin*	*p-value*	*alpha*	*xmin*	*p-value*
Nginx	*AD*	1.95	0.94	0.89	/	/	/
	Deg-Ch	1.64	0.46	0.69	1.62	0.001	0.507
	Deg-In	1.73	0.58	0.58	1.67	0.006	0.786
	Deg-Out	1.59	0.54	0.74	1.54	0.007	0.709
Gedit	*AD*	1.53	0.82	0.48	/	/	/
	Deg-Ch	1.59	0.92	0.79	1.55	0.022	0.901
	Deg-In	1.48	1.28	0.82	1.558	0.012	0.722
	Deg-Out	1.61	0.74	0.90	1.51	0.012	0.597
Nagioscore	*AD*	1.62	0.51	0.13	/	/	/
	Deg-Ch	1.67	0.82	0.11	1.46	0.009	0.18
	Deg-In	1.56	0.48	0.18	1.43	0.009	0.15
	Deg-Out	1.64	0.71	0.11	1.47	0.008	0.25
Redis	*AD*	2.58	1.29	0.26	/	/	/
	Deg-Ch	2.49	0.89	0.38	1.61	0.003	0.687
	Deg-In	2.25	0.74	0.49	1.58	0.004	0.697
	Deg-Out	2.37	0.65	1.83	1.56	0.004	0.752

- *Most of the modifications are made to several "key" versions*
- *AD, Deg-In, Deg-Out, and Deg-Ch conform to the power law distribution*
- *MC and OM show no significant characteristics of the power law distribution*

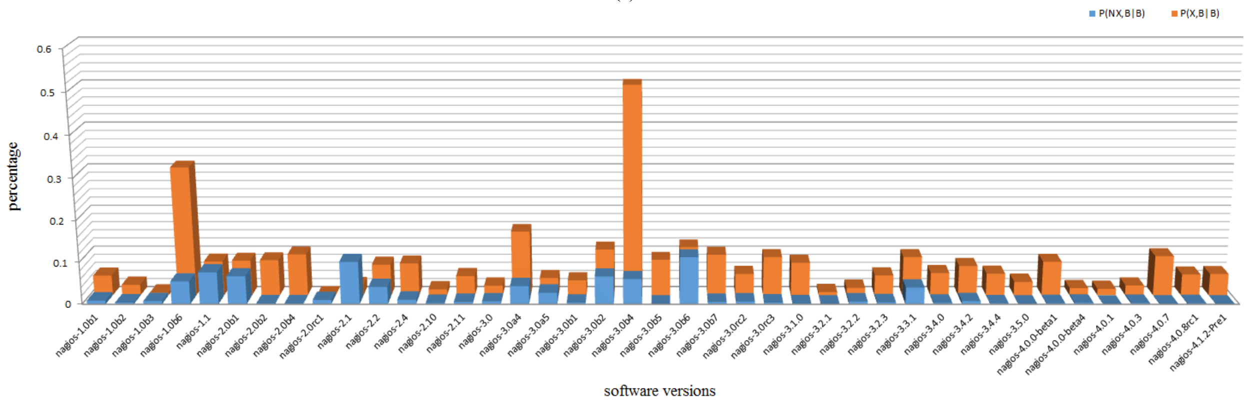
C. RQ3: WHAT IS THE CORRELATION BETWEEN FUNCTION-LEVEL MODIFICATIONS AND THEIR CORRESPONDING FAULT-PRONENESS?

In this subsection, we further investigate the correlation between different function-level modifications and fault-proneness of functions. Here, we use the conditional probability-based method discussed in Section IV B. Based on the description in Section IV B, $P_{X,B|B}$ represents the percentage of defected functions that go through the modification X. Generally speaking, with respect to a function-level modification type X, if $P_{X,B|B}$ is significantly greater than $P_{\overline{X},B|B}$, it indicates that the modification X is closely correlated with whether a specific function contains bug(s).

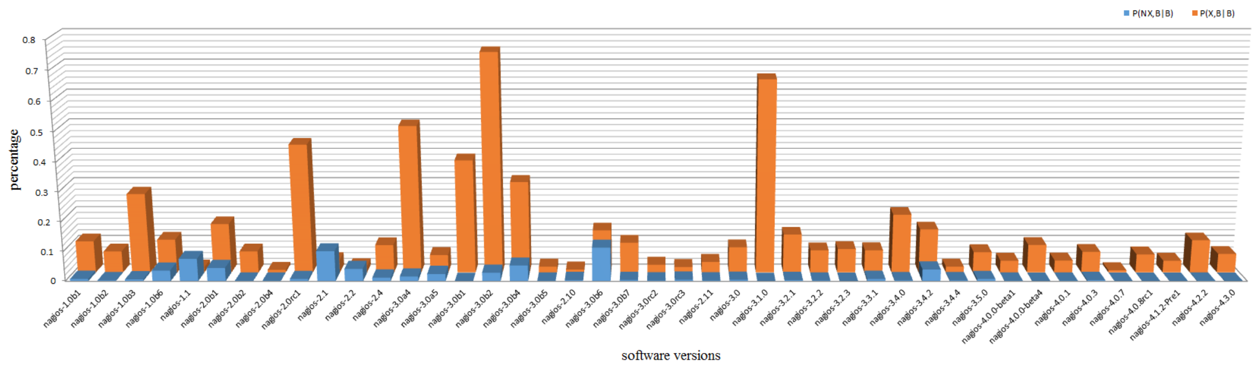
In Figure 6, we only include the figures for *AD*, *Deg-In*, *Deg-Out*, and *Deg-Ch* collected from Nagioscore. Note that similar phenomena can also be observed for other types of modifications with respect to other projects. In these figures, we only include those versions that contain at least one bug, and other bug-free versions are ignored in this subsection. In



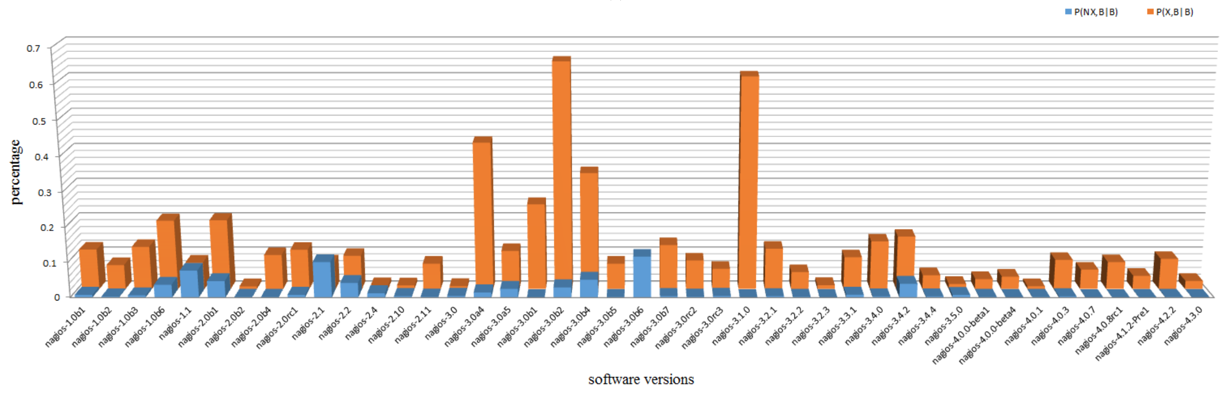
(a)



(b)



(c)



(d)

FIGURE 6. Correlation between function modifications and functions' defect-proneness for Nagioscore. (a) *AD*; (b) *Deg-In*; (c) *Deg-Out*; (d) *Deg-Ch*.

TABLE 5. Correlations between function modifications and a function's defect-proneness LR results.

Logistic regression results		Nginx	Gedit	Nagioscore	Redis
AD	p-value	0.000	0.025	0.038	0.047
	Odds ratio	10.62	13.05	14.17	14.47
Deg-Ch	p-value	0.034	0.041	0.000	0.001
	Odds ratio	1.95	3.67	0.78	0.91
Deg-In	p-value	0.000	0.009	0.000	0.027
	Odds ratio	19.50	20.51	54.45	28.33
Deg-Out	p-value	0.028	0.000	0.009	0.035
	Odds ratio	6.51	2.33	5.24	4.86

these figures, we use orange bars to represent $P_{X,B|B}$ and blue bars for $P_{\overline{X},B|B}$.

As seen in Figure 6, we find that for most of the scenarios, $P_{X,B|B}$ is significantly greater than $P_{\overline{X},B|B}$. As illustrated in Figure 6(a), $P_{AD,B|B}$ is greater than $P_{\overline{AD},B|B}$ in 35 of 36 versions. For example, with respect to version 2.1, $P_{AD,B|B}$ is 0.932, while $P_{\overline{AD},B|B}$ is only 0.005. Taking version 3.0b2 in Figure 6(a) as another example, $P_{AD,B|B}$ is 0.670, whereas $P_{\overline{AD},B|B}$ is 0.057.

However, several exceptions exist. For example, as presented in Figure 6(c), for version 2.1, $P_{Deg-In,B|B}$ is 0.024 while $P_{\overline{Deg-In},B|B}$ is 0.102. This indicates that for this particular version, bugs distributed more on the functions without *Deg-In* modifications.

For further verification, we use logistic regression discussed in Section IV C to analyze the correlation between function-level modifications and defects. The results are shown in Table 5. Based on the table, all the p-values for *AD*, *Deg-In*, *Deg-Out*, and *Deg-Ch* are smaller than 0.005, which shows that those function-level modifications are closely correlated with whether the function contains bug(s) or not. In addition, their corresponding odds ratios are all greater than 1, which represents that these modifications are positively related to the functions' fault-proneness. In other words, the increase in these four types of modifications will very likely induce defects into the functions being modified.

We would also like to mention that the correlation between MC/OM and bug distribution is not remarkably significant and is not included in Figure 6 and Table 5.

In summary, with respect to RQ3, our observations are as follows:

- *AD, Deg-In, Deg-Out, and Deg-Ch are positively related to whether a function contains bug(s) or not*
- *MC and OM do not significantly correlate to whether a function is defected or not*

D. RQ4: WHAT IS THE CORRELATION BETWEEN ATTRIBUTES OF FUNCTIONS AND THEIR CORRESPONDING FAULT-PRONENESS?

In this subsection, we will explore the correlation between attributes of functions and their corresponding fault-proneness. As discussed in Section III B, the following three attributes will be considered: *Time*, *DiffCount*, and *HisBug*.

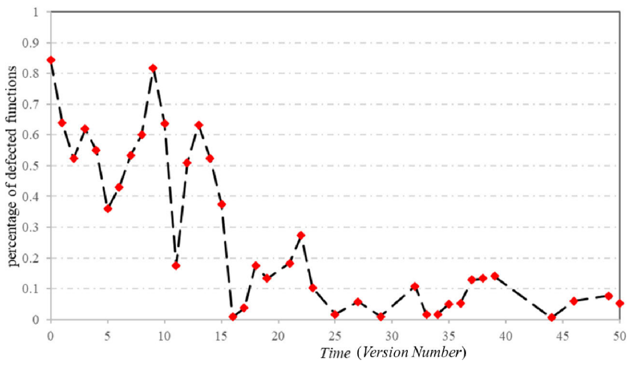


FIGURE 7. Distribution of defects with time.

1) DISTRIBUTION OF BUGS WITH TIME

Intuitively, our conjecture is that the shorter a function exists in a software system, the more likely that the function may contain bug(s). If a function is removed very shortly after it is added into the system, it is extremely likely that the function is not compatible with other functions and therefore can be rather difficult to implement. In such cases, the function could induce more bugs than other functions normally do.

In Figure 7, each data point represents the percentage of functions being defected with a predefined *Time* value. For example, the data entry (0, 0.844) represents that for all those functions defined in one version and then deleted in the very next version, ($0.844 \times 100\% =$) 84.4% of the functions are defected. Similarly, (23, 0.103) indicates that only ($0.103 \times 100\% =$) 10.3% of functions with a Time value of 23 are defected.

It is noteworthy that the percentage of defected functions decreases significantly with the increase in a function's Time. For example, the average percentage of defected functions with *Time* values between zero and 15 is ($0.548 \times 100\% =$) 54.8%. However, the value decreases significantly to ($0.083 \times 100\% =$) 8.3% for functions with *Time* values greater than 15. The downward trend indicates that a function is more likely to contain bug(s) if its Time value is small. Therefore, if the lifetime of a function is relatively short, the possibility of the function containing bug(s) is high.

2) DISTRIBUTION OF BUGS WITH DIFFCOUNT

As discussed in Section III B, we use *DiffCount* to represent the total number of function-level modifications made to a particular function during the life cycle of the project. Intuitively speaking, *DiffCount* should be positively related to the possibility of the function being defected. This is because the more modifications that are made to a particular function, the more likely programmers may make errors and introduce bugs.

In Figure 8, the X-axis represents the total number of function-level modifications in different functions. The Y-axis on the left shows the number of functions, and the color bar on the right represents the number of bugs in those functions in the same vertical bar. For example, when

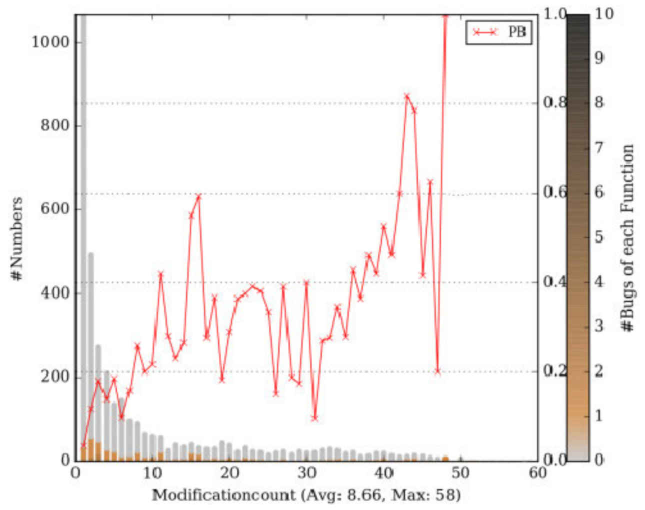


FIGURE 8. Distribution of defects with *DiffCount*.

DiffCount is equal to three, the grey vertical bar at x = 3 has a length of 202, indicating that 202 functions with three function-level modifications do not contain any bug. Similarly, the orange bar underneath has a length of 37, which means that 37 functions with three modifications contain one bug. It can be observed that with the increase in *DiffCount*, the number of functions with the number of modifications decreases. The same applies to the number of bug-free functions and of defected functions.

We also include a red line indicating the percentage of defected functions with a specific number of *DiffCount*. For example, the data point (12, 0.260) represents that (0.260 × 100% =) 26.0% of functions with a *DiffCount* of 12 are defected. Unlike the number of functions, the percentage of defected functions increases significantly with the increase in *DiffCount*. In other words, the more function-level modifications that are made to a function, the higher the possibility that the function is defected.

3) DISTRIBUTION OF BUGS WITH HISBUG

We also use the mosaic plots (as in Figure 9) to describe the correlation between *HisBug* and the functions being defected. In these plots, we use the rectangular area to represent the percentage of functions with a specific property. For example, the area of the rectangle marked with (0, 0) represents the percentage of bug-free functions that are also bug-free in all its previous versions. Similarly, the area of the rectangle marked with (1, 1) shows the percentage of defected functions which also had bugs in their previous versions.

Without loss of generality, we only include the mosaic plots for Nagioscore 2.6 and 3.0b2. However, most versions of projects share similar characteristics. According to these plots, most of the functions do not contain bugs. For example, as illustrated in Figure 9(a), 72.5% of all functions are bug-free, not only in their current version but also in their previous versions.

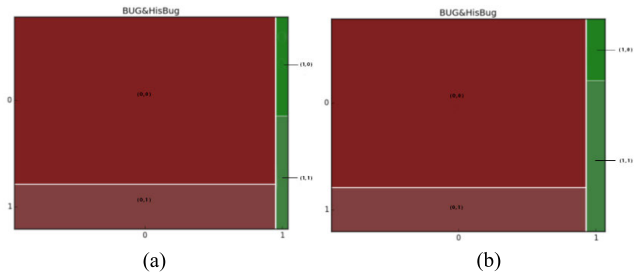


FIGURE 9. Mosaic plots for HisBug. (a) Nagioscore-2.6; (b) Nagioscore-3.0b2.

TABLE 6. Correlations between function modifications and functions' defect-proneness LR results.

Logistic regression results		Nginx	Gedit	Nagioscore	Redis
Time	p-value	0.034	0.003	0.004	0.019
	Odds ratio	0.310	0.378	0.028	2.610
DiffCount	p-value	0.016	0.009	0.006	0.045
	Odds ratio	1.209	1.912	3.920	2.640
HisBug	p-value	0.004	0.009	0.000	0.000
	Odds ratio	5.687	1.262	2.032	3.113

As shown in Figure 9(b), the rectangular area (0, 0) is 0.710, whereas the area is 0.200 for rectangle (0, 1). This indicates that for functions without bugs in their previous versions, only one-third of them are defected in the current version. However, the areas of rectangles (1, 0) and (1, 1) are 0.025 and 0.065, respectively. In other words, for the functions that were defected in previous versions, almost 61.5% are still defected.

Therefore, the possibility of a function containing defects increases if it contains bugs in a previous version.

For further verification, we use logistic regression discussed in Section IV C to analyze the correlation between functions' attributes and fault-proneness. The results are shown in Table 6. It can be observed that all the p-values are less than 0.05, which represents that these three attributes are closely correlated to functions' fault-proneness. In addition, the odds ratios of these attributes also indicate that *Time* is negatively related to functions' fault-proneness, while *DiffCount* and *HisBug* are positively related to fault-proneness.

In summary, with respect to RQ4, our observations are as follows:

- *Time is negatively related to whether a function contains bug(s) or not*
- *DiffCount and HisBug are positively related to whether a function contains bug(s) or not*

E. RQ5: WHAT IS THE CORRELATION BETWEEN SOFTWARE STRUCTURAL CHARACTERISTICS AND THE FAULT-PRONENESS OF FUNCTIONS?

In this section, we use logistic regression to analyze the relationship between FCNs structural characteristics and function's defect-proneness. The results of global network measures are shown in Table 7, while Table 8 presents the results of ego network measures.

TABLE 7. Global network measures.

Logistic regression results		Nginx	Gedit	Nagioscore	Redis
Degree	p-value	0.023	0.009	0.003	0..048
	Odds ratio	13.69	10.58	9.272	26.10
Out_Degree	p-value	0.016	0.001	0.000	0.026
	Odds ratio	785.26	191.24	2811.68	289.62
In_Degree	p-value	0.436	0.529	0.120	0.685
	Odds ratio	0.619	0.596	0.196	2.585
Ripple_ Degree	p-value	0.017	0.022	0.000	0.019
	Odds ratio	32.764	15.288	10.655	14.694
Closeness	p-value	0.049	0.035	0.001	0.019
	Odds ratio	3.479	5.496	2.8589	2.187
Betweenness	p-value	0.049	0.024	0.016	0.039
	Odds ratio	981.48	645.26	7521.581	487.69
PageRank	p-value	3.486	0.789	0.375	1.287
	Odds ratio	0.486	0.389	0.200	1.756
Clus_Coef	p-value	0.004	0.008	0.029	0.034
	Odds ratio	3.426	0.947	0.982	1.576
K_Cores	p-value	0.004	0.000	0.000	0.002
	Odds ratio	14.537	10.489	9.376	28.469
Eigenvector	p-value	0.018	0.009	0.007	0.004
	Odds ratio	32.471	68.290	20.608	18.465
Efficiency	p-value	0.000	0.024	0.000	0.008
	Odds ratio	98.147	258.76	150.551	157.46
Constraint	p-value	0.017	0.002	0.001	0.014
	Odds ratio	-1.276	-0.048	-0.5198	-0.189

In Table 7 and Table 8, the rows marked gray indicate that the measures do not satisfy the "relevant" standard. In other words, these measures are not related to whether a function contains bugs or not.

In summary, with respect to RQ5, our observations are listed in Table 9.

VI. CROSS-COMPARISON

For comparison, we use logistic regression to analyze the relationship between 19 traditional source code metrics shown in Table 10 and functions' defect-proneness on the four open source projects used in this paper. These traditional source code metrics include Lines-of-code Metrics, Halstead Metrics, McCabe's Cyclomatic Metrics, and Maintainability Index Metrics. The values of these metrics are computed using CMT++.

The results of traditional source code metrics are shown in Table 11. In Table 11, the cells marked gray indicate that the metrics satisfy the "relevant" standard, that is to say, these metrics can be considered as "relevant" to whether a function contains bugs or not. Obviously, only (10 / 152 =) 6.7% of the scenarios can prove the "relevance". Moreover, the same metric has different correlation with bugs in different projects, so it has no universal significance. Compared to the metrics we proposed in this paper, these metrics are of no statistical significance.

TABLE 8. Ego network measures.

Logistic regression results		Nginx	Gedit	Nagioscore	Redis
Size	p-value	0.014	0.005	0.011	0.008
	Odds ratio	0.687	0.095	0.084	0.147
Ties	p-value	0.001	0.000	0.000	0.000
	Odds ratio	4.187614e+46	5.84756e+70	9.249639e+69	6.492157e+59
Pairs	p-value	0.049	0.017	0.006	0.076
	Odds ratio	1.479564e-45	4.218964e-46	7.917911e-34	8.12476e-30
Density	p-value	2.469	1.486	0.370	0.476
	Odds ratio	1.475628e-14	9.147625e-13	6.636574e-11	9.51476e-8
nWeakComp	p-value	0.014	0.001	0.002	0.000
	Odds ratio	0.147	0.045	0.012	0.009
pWeakComp	p-value	1.567	0.476	0.211	0.083
	Odds ratio	2.486	2.486	1.250	7.654
2StepReach	p-value	0.006	0.001	0.000	0.018
	Odds ratio	73.51	86.16	45.53	49.18
2StepPct	p-value	0.004	0.000	0.000	0.005
	Odds ratio	67.218	104.26	71.396	49.561
ReachEffic	p-value	0.752	1.458	0.900	0.834
	Odds ratio	0.416	0.176	0.746	0.218
Broker	p-value	1.426	1.249	0.661	1.357
	Odds ratio	0.0149	0.0005	0.0006	0.0012
nClosed	p-value	0.029	0.004	0.000	0.000
	Odds ratio	2.486194e+11	1.006549e+13	1.848833e+11	8.467599e+10
EgoBetween	p-value	0.018	0.000	0.001	0.009
	Odds ratio	19.16	35.18	21.37	42.19

TABLE 9. Network measures correlates to functions' fault-proneness.

Global network measure	Ego network measure
Degree, Out_Degree, Ripple_Degree, Closeness, Betweenness, Clus_Coef, K_Cores, Eigenvector, Efficiency, Constraint	Size, Ties, Pairs, nWeakComp, 2StepReach, 2StepPct, nClosed, EgoBetween

VII. THREATS TO VALIDITY

The first threat involves whether the software projects selected in this paper is representative enough to serve as the basis of a solid study. In our study, we analyzed more than 1,000 versions from four open source projects, which is far more than the numbers of versions used in other studies. For example, the authors of [34] selected 39 versions, which is far less than the number of versions we selected. In the next stage, we will include more projects to make the results more universal.

Second, the defects used in this study are extracted using the steps described in Section III B. However, in order to determine the exact locations of defects, manual inspection is required. As a result, there is a possibility that the locations of these defects are incorrect. However, we did our best to

TABLE 10. Traditional source code metrics used in this study for comparison analysis.

Lines-of-code Metrics (LOC):4	
LOCphy	number of physical lines
LOCbl	number of blank lines (a blank line inside a comment block is considered to be a comment line)
LOCpro	number of program lines (declarations, definitions, directives, and code)
LOCcom	number of comment lines
Halstead-Metrics:9	
n1	number of unique (distinct) operators
n2	number of unique (distinct) operands
N1	total number of operators
N2	total number of operands
V	the program volume (V) is the information contents of the program, measured in mathematical bits
D	the difficulty level or error proneness (D) of the program is proportional to the number of unique operatorsin the program
L	the program level (L) is the inverse of the error pronenessof the program
E	the effort to implement (E) or understand a program is proportional to the volumeand to the difficulty levelof the program
B	the number of delivered bugs (B) correlates with the overall complexity of the software
Maintainability Index Metrics:3	
MIwoc	Maintainability Index without comments
MIcw	Maintainability Index comment weight
MI	Maintainability Index = MIwoc + MIcw
McCabe's Cyclomatic Metrics:1	
v(G)	McCabe's Cyclomatic number v(G) shows the complexity of the flow of control through a piece of code. v(G) is the number of conditional branches in the flowchart.
Others:2	
Params	number of parameters
MaxND	Maximum nesting depth is somewhat related to the algorithmic complexity v(G) and is an indication on how deep the algorithmic nesting structure is in a function.

TABLE 11. Correlations between traditional source code metrics and function's defect-proneness LR results.

Lines-of-code Metrics		Nginx	Gedit	Nagioscore	Redis
LOCphy	p-value	0.584	0.531	0.809	0.074
	Odds ratio	1.304	0.740	1.149	2.184
LOCbl	p-value	0.64	0.707	0.812	0.128
	Odds ratio	0.796	1.229	0.870	0.513
LOCpro	p-value	0.564	0.537	0.806	0.121
	Odds ratio	0.751	1.306	0.867	0.508
LOCcom	p-value	0.874	0.463	0.667	0.07
	Odds ratio	0.923	1.450	0.780	0.450
Halstead-Metrics		**Nginx**	**Gedit**	**Nagioscore**	**Redis**
n1	p-value	0.268	0.208	0.261	0.008
	Odds ratio	1.247	0.572	1.101	1.539
n2	p-value	0.464	0.081	0.981	0.056
	Odds ratio	1.038	1.267	1.001	0.848
N1	p-value	0.024	0.95	0.478	0.952
	Odds ratio	1.061	1.008	1.009	0.997
N2	p-value	0.402	0.497	0.458	0.759
	Odds ratio	1.033	1.104	0.991	0.983
V	p-value	0.018	0.587	0.466	0.949
	Odds ratio	0.989	0.994	0.999	0.999
D	p-value	0.024	0.05	0.076	0.123
	Odds ratio	0.615	1.958	0.918	0.780
L(x1000)	p-value	0.875	0.221	0.431	0.092
	Odds ratio	1.001	0.972	0.990	1.012
E	p-value	0.249	0.024	0.165	0.817
	Odds ratio	0.999	1.000	0.999	0.999
B(x100)	p-value	0.026	0.049	0.133	0.457
	Odds ratio	1.319	0.310	1.051	1.201
Maintainability Index Metrics		**Nginx**	**Gedit**	**Nagioscore**	**Redis**
MIwoc	p-value	0.961	0.504	0.968	0.403
	Odds ratio	1.051	2.187	0.986	0.658
MIcw	p-value	0.955	0.502	0.655	0.475
	Odds ratio	0.942	2.205751	1.155	0.698853
MI	p-value	0.934	0.503	0.891	0.488
	Odds ratio	0.917	0.454	0.956	1.414
McCabe's Cyclomatic Metrics		**Nginx**	**Gedit**	**Nagioscore**	**Redis**
v(G)	p-value	0.692	0.479	0.023	0.988
	Odds ratio	0.969	0.094	1.088	1.002
Others		**Nginx**	**Gedit**	**Nagioscore**	**Redis**
Params	p-value	0.382	0.715	0.537	0.005
	Odds ratio	0.804	0.881	0.955	1.473
MaxND	p-value	0.403	0.36	0.36	0.06
	Odds ratio	1.331	1.766	0.675	0.951

prevent this from happening. For the bugs that cannot be reproduced, they will be dropped without further investigation. In addition, several engineers are included in this process. With respect to each bug, they should all agree on the actual location of the bug. If no agreement can be reached, one more engineer will be asked to investigate the bug and make the final decision.

Finally, the network measures used in this study may not be sufficient enough. There is the possibility that more attributes should be included, and some of the included attributes only apply to the selected projects. To make our study more representative of most software projects today, more open source projects as well as network measures will be included in the next step.

VIII. RELATED STUDIES

In the last few decades, several studies have been conducted to explore the empirical principles with respect to the distribution of faults in software systems ([16], [45]–[48]). The Pareto principle is the most frequently cited. It suggests that most of the faults in a software system lie in a small number of modules. Fenton and Ohlsson [45] not only verified the Pareto principle but also reported a "counter-intuitive relationship between pre- and postrelease faults." The least fault-prone modules in postrelease are the most fault-prone in prerelease, and the most fault-prone modules in postrelease are the least fault-prone in prerelease. The authors of [49] also observed the same with respect to a large industrial software system. Other studies ([16], [47], [48]) also verified the Pareto principle.

Aside from the Pareto principle, studies such as [47] have also been proposed to examine the quality assurance efforts spent on various software modules. Grbac *et al.* [47] suggested that defects are unevenly distributed in software systems, making quality assurance efforts spent on different parts of a software system also varied. This finding needs to be further validated by more empirical data.

Various software metrics have also been proposed to help developers better predict possible locations of bugs. They can be divided into two types: product metrics and process metrics.

Product metrics can be further divided into two categories: object-oriented metrics and source code metrics. Many studies focus on object-oriented metrics for fault prediction, among which the CK metrics are most frequently referenced [50].

Cross-comparisons among different object-oriented metrics have also been conducted. Goel and Singh [51] compared ten class level metrics with respect to their abilities in fault prediction for object-oriented software. Gyimthy *et al.* [52] evaluated the performance of seven object-oriented metrics, including Weighted Methods per Class (WMC) and Depth of Inheritance Tree (DIT). Briand *et al.* [6] used 28 coupling measures, 10 cohesion measures, 11 inheritance measures, and a small selection of size measures to show that many coupling and inheritance measures are strongly related to the probability of fault detection in a class. Models derived by some of the coupling and inheritance measures can accurately predict most of the faults in some classes. Similar studies such as [53] were also reported. However, fault prediction based on object-oriented metrics can only help developers predict bugs at a class or file level. Moreover, they cannot be replicated across different environments and systems to obtain generalized conclusions.

Studies ([47], [54]–[56]) have shown the limitation of using source code metrics (e.g., lines of code) to predict fault-proneness. Zhou *et al.* [56] reported that many source code metrics have "moderate or almost moderate" ability to detect fault-prone classes. Xu *et al.* [54] examined the risk prediction abilities with respect to twelve metrics and showed that these metrics are not strong enough to establish a precise prediction model. Similar results can also be found in ([47], [55]).

Unlike product metrics focusing on the static aspects of software, process metrics emphasize the lack of considering aspects related to the development process. Graves *et al.* [57] conducted a study to predict fault incidence based on the change history of software. Illes-Seifert *et al.* [58] examined the relationship between a file's history and its corresponding fault-proneness. They also used open source programs to validate their findings. In recent years, software network measurements with good fault prediction ([59]–[61]) have attracted attention from both academia and industry.

The theory of complex networks has been widely used in the area of software engineering to help engineers analyze software systems, especially for systems with extremely large scale and complexity. Li *et al.* [62] considered software systems as complex networks and proved that the power consumption of software is non-linearly related to its network characteristics. Chong and Lee [63] used a weighted complex network to represent object-oriented software systems and to capture the structural characteristics of these systems. The maintainability and reliability of these systems were then carefully evaluated based on the networks. Later, they [64] provided an approach that can help practitioners automatically achieve the clustering constraints from the implicit structure of software systems based on graph theory. In [65], Myers discussed the relationships between several network topological measurements to software engineering practices. Qian *et al.* [66] applied the widely-used community detection algorithm in the clustering of software modules and demonstrated that the clustering performance is better than that of the Bunch method. In addition, Le and Panchal [67] proposed an analysis method for product structures at both the product level and module level. In [68], the authors presented an approach that models software packages as nodes and dependencies among them as edges. Then, methods in the field of complex networks were applied to explore properties of network models. In [69], community structure detection was proposed to evaluate the cohesion of object-oriented software programs.

In recent years, software network measurements have also been attracting practitioners from both academia and industry. Zimmermann and Nagappan [70] suggested that by analyzing the dependencies exist among various pieces of code, engineers could identify around 60% of the critical binaries, which is twice as many as the percentage identified by complexity metrics. Tosun *et al.* [60] indicated that the network measurements are closely related to defective modules, whereas the correlation is not significant for small-scale projects. T.H.D. Nguyen et al. [59] demonstrated that several of the dependency network measurements are strongly correlated with post-release failures, while other measurements are not. They also indicated that testing can benefit from bug predictions and reduce costs during the entire testing process. In [61], the authors proposed an MHCP model to help engineers analyze open source projects based on software complex network measures.

IX. CONCLUSIONS AND FUTURE WORK

In this paper, we first analyze the distribution of bugs from both *time* and *space* points of view. Different types of function-level modifications are proposed, and further investigations are performed to determine the correlation between these modifications and functions' fault-proneness. Other attributes related to functions as well as software networks are also utilized as possible indicators for fault prediction.

Our results indicate that: 1) bug distributions in both *time* and *space* conform to the power law distribution; 2) of the six types of function-level modifications, four of them (*AD*, *Deg-In*, *Deg-Out*, and *Deg-Ch*) fit the power law distribution, while *MC* and *OM* show no such pattern; 3) *AD*, *Deg-In*, *Deg-Out*, and *Deg-Ch* are all positively related to functions' fault-proneness, while *MC* and *OM*'s correlations with functions' fault proneness are not remarkable; 4) of the three functional attributes, two of them (*DiffCount* and *HisBug*) are positively related to functions' fault-proneness, while *Time* exhibits significant negative correlation with the possibility that a function contains bug(s); 5) several software network measures are also proven to be closely related to functions' fault proneness. Overall, we have identified a set of critical factors that can serve as valuable indicators for existing fault prediction approaches to help increase their prediction accuracy.

In the next step, analyses towards more software projects will be performed to achieve more in-depth knowledge with respect to factors that have an influence on functions' fault-proneness. We would also like to establish a novel fault prediction model based on not only the factors we have proposed in this paper but also the existing metrics to provide better precision in fault prediction. In addition, other software network measures will also be introduced into the area of software fault prediction so that engineers can achieve better fault prediction strength.

ACKNOWLEDGMENTS

The authors would like to thank the support from Technical Basis Funding: JSZL2017601C001.

REFERENCES

[1] M. R. Lyu, "Software reliability engineering: A roadmap," in *Proc. Future Softw. Eng. (FOSE)*, Minneapolis, MN, USA, May 2007, pp. 153–170.

[2] N. Report, "Software Errors Cost U.S. Economy 59.5 Billion Annually," NIST, Gaithersburg, MD, USA, Tech, Rep. 02-3, May 2002.

[3] N. Leveson, *Safeware: System Safety and Computers*. Boston, MA, USA: Addison-Wesley, Sep. 1995.

[4] W. E. Wong, X. Li, and P. A. Laplante, "Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures," *J. Syst. Softw.*, vol. 133, pp. 68–94, Nov. 2017.

[5] B. W. Boehm, "An experiment in small-scale application software engineering," *IEEE Trans. Softw. Eng.*, vols. SE–7, no. 5, pp. 482–493, Sep. 1981.

[6] L. C. Briand, J. Wüst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, May 2000.

[7] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, Szeged, Hungary, Sep. 2011, pp. 4–14.

[8] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel, "Early quality prediction: A case study in telecommunications," *IEEE Softw.*, vol. 13, no. 1, pp. 65–71, Jan. 1996.

[9] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005.

[10] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Proc. 15th Int. Symp. Softw. Rel. Eng.*, Saint-Malo, France, Nov. 2004, pp. 417–428.

[11] K. Magal. R and S. Gracia Jacob, "Improved random forest algorithm for software defect prediction through data mining techniques," *Int. J. Comput. Appl.*, vol. 117, no. 23, pp. 18–22, 2015.

[12] Ö. F. Arar and K. Ayan, "Software defect prediction using cost-sensitive neural network," *Appl. Soft Comput.*, vol. 33, pp. 263–277, Aug. 2015.

[13] R. Jindal, R. Malhotra, and A. Jain, "Software defect prediction using neural networks," in *Proc. 3rd Int. Conf. Rel., Infocom Technol. Optim.*, Noida, India, Oct. 2014, pp. 1–6.

[14] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, May 2008.

[15] X. Rong, F. Li, and Z. Cui, "A model for software defect prediction using support vector machine based on CBA," *Int. J. Intell. Syst. Technol. Appl.*, vol. 15, no. 1, p. 19, Apr. 2016.

[16] C. Andersson and P. Runeson, "A replicated quantitative analysis of fault distributions in complex software systems," *IEEE Trans. Softw. Eng.*, vol. 33, no. 5, pp. 273–286, May 2007.

[17] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012.

[18] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 402–419, Jun. 2007.

[19] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Inf. Softw. Technol.*, vol. 59, pp. 170–190, Mar. 2015.

[20] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[21] B. Turhan and A. Bener, "Analysis of naive Bayes' assumptions on software fault data: An empirical study," *Data Knowl. Eng.*, vol. 68, no. 2, pp. 278–290, Feb. 2009.

[22] J. Hernández-González, D. Rodriguez, I. Inza, R. Harrison, and J. A. Lozano, "Learning to classify software defects from crowds: A novel approach," *Appl. Soft Comput.*, vol. 62, pp. 579–591, Jan. 2018.

[23] D. P. P. Mesquita, L. S. Rocha, J. P. P. Gomes, and A. R. Rocha Neto, "Classification with reject option for software defect prediction," *Appl. Soft Comput.*, vol. 49, pp. 1085–1093, Dec. 2016.

[24] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "MULTI: Multi-objective effort-aware just-in-time software defect prediction," *Inf. Softw. Technol.*, vol. 93, pp. 1–13, Jan. 2018.

[25] D. Radjenović, M. Herčiko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1397–1418, Aug. 2013.

[26] *Nginx*. Accessed: May 20, 2019. [Online]. Available: http://nginx.org/en

[27] *Gedit*. Accessed: May 20, 2019. [Online]. Available: https://github.com/GNOME/gedit

[28] *Nagioscore*. Accessed: May 20, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Nagios

[29] *Redis*. Accessed: May 20, 2019. [Online]. Available: https://redis.io

[30] *Doxygen*. Accessed: May 20, 2019. [Online]. Available: http://www.doxygen.org

[31] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proc. Int. Conf. Softw. Maintenance ICSM*, 2000, pp. 120–130.

[32] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, p. 1, Jul. 2005.

[33] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. FSE*, Hong Kong, 2014, pp. 654–665.

[34] W. Ma, L. Chen, Y. Yang, Y. Zhou, and B. Xu, "Empirical analysis of network measures for effort-aware fault-proneness prediction," *Inf. Softw. Technol.*, vol. 69, pp. 50–70, Jan. 2016.

[35] *NetworkX*. Accessed: May 20, 2019. [Online]. Available: http://networkx.github.io

[36] *Plfit*. Accessed: May 20, 2019. [Online]. Available: https://pypi.python.org/pypi/plfit
[37] M. L. Goldstein, S. A. Morris, and G. G. Yen, "Problems with fitting to the power-law distribution," *Eur. Phys. J. B*, vol. 41, no. 2, pp. 255–258, Sep. 2004.
[38] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 356–370, May 2011.
[39] Y. Tan and J. Wu, "Network structure entropy and its application to scale-free networks," *Syst. Eng. Theory Pract.*, vol. 24, no. 6, pp. 1–3, 2004.
[40] S. Zhang, J. Ai, and X. Li, "Correlation between the distribution of software bugs and network motifs," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Vienna, Austria, Aug. 2016, pp. 202–213.
[41] D. A. Belsley, E. Kuh, and R. E. Welsch, *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*. Hoboken, NJ, USA: Wiley, 2005.
[42] R. D. Cook and R. Dennis, "Detection of Influential Observations in Linear Regression," *Technometrics*, vol. 19, no. 1, pp. 15–18, 1977.
[43] J. Juran, *Quality Control Handbook*. New York, NY, USA: McGraw-Hill, 1974.
[44] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. Softw. Eng.*, vol. 22, no. 12, pp. 886–894, Dec. 1996.
[45] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000.
[46] T. G. Grbac and D. Huljenić, "On the probability distribution of faults in complex software systems," *Inf. Softw. Technol.*, vol. 58, pp. 250–258, Feb. 2015.
[47] T. G. Grbac, P. Runeson, and D. Huljenic, "A second replicated quantitative analysis of fault distributions in complex software systems," *IEEE Trans. Softw. Eng.*, vol. 39, no. 4, pp. 462–476, Apr. 2013.
[48] K. H. Moller and D. J. Paulish, "An empirical investigation of software fault distribution," in *Proc. IEEE 1st Int. Softw. Metrics Symp.*, May 1993, pp. 82–90.
[49] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," *ACM SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, p. 55, Jul. 2002.
[50] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
[51] B. Goel and Y. Singh, "Empirical Investigation of Metrics for Fault Prediction on Object-Oriented Software," *Comput. Inf. Sci.*, pp. 255–265, 2008.
[52] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005.
[53] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *J. Syst. Softw.*, vol. 56, no. 1, pp. 63–75, Feb. 2001.
[54] Z. Xu, X. Zheng, and P. Guo, "Empirically Validating Software Metrics for Risk Prediction Based on Intelligent Methods," *Digital Information Research Foundation*, vol. 5, no. 3, pp. 1049–1054, 2007.
[55] H. Zhang, "An investigation of the relationships between lines of code and defects," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2009, pp. 274–283.
[56] Y. Zhou, B. Xu, and H. Leung, "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems," *J. Syst. Softw.*, vol. 83, no. 4, pp. 660–674, Apr. 2010.
[57] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.
[58] T. Illes-Seifert and B. Paech, "Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs," *Inf. Softw. Technol.*, vol. 52, no. 5, pp. 539–558, May 2010.
[59] T. H. D. Nguyen, B. Adams, and A. E. Hassan, "Studying the impact of dependency network measures on software quality," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.
[60] A. Tosun, B. Turhan, and A. Bener, "Validation of network measures as indicators of defective modules in software systems," in *Proc. 5th Int. Conf. Predictor Models Softw. Eng.*, 2009, pp. 1–9.
[61] Y. Yang, J. Ai, X. Li, and W. E. Wong, "MHCP model for quality evaluation for software structure based on software complex network," in *Proc. IEEE 27th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Ottawa, ON, Canada, Oct. 2016, pp. 298–308.
[62] D. Li, B. Guo, Y. Shen, J. Li, and Y. Huang, "Software power modeling method at architecture level based on complex networks," *Sustain. Comput., Informat. Syst.*, vol. 12, pp. 34–42, Dec. 2016.
[63] C. Y. Chong and S. P. Lee, "Analyzing maintainability and reliability of object-oriented software using weighted complex network," *J. Syst. Softw.*, vol. 110, pp. 28–53, Dec. 2015.
[64] C. Y. Chong and S. P. Lee, "Automatic clustering constraints derivation from object-oriented software using weighted complex network with graph theory analysis," *J. Syst. Softw.*, vol. 133, pp. 28–53, Nov. 2017.
[65] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 68, no. 4, pp. 46–116, Oct. 2003.
[66] Q. Gunqun, Z. Lin, and Z. Li, "Applying complex network method to software clustering," in *Proc. Int. Conf. Comput. Sci. Softw. Eng.*, Dec. 2008, pp. 310–316.
[67] Q. Le and J. H. Panchal, "Network-based analysis of the structure and evolution of an open source software product," in *Proc. 45th Hawaii Int. Conf. Syst. Sci.*, Jan. 2012, pp. 3436–3445.
[68] X. Zheng, D. Zeng, H. Li, and F. Wang, "Analyzing open-source software systems as complex networks," *Phys. A, Stat. Mech. Appl.*, vol. 387, no. 24, pp. 6190–6200, Oct. 2008.
[69] Y. Qu, X. Guan, Q. Zheng, T. Liu, L. Wang, Y. Hou, and Z. Yang, "Exploring community structure of software call graph and its applications in class cohesion measurement," *J. Syst. Softw.*, vol. 108, pp. 193–210, Oct. 2015.
[70] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. 13th Int. Conf. Softw. Eng. (ICSE)*, May 2008, pp. 531–540.

JUN AI is currently an Associate Professor with the School of Reliability and Systems, Beihang University, Beijing, China. His research interests include software reliability, fault prediction, and software evaluation.

HAORAN GUO was born in Shandong, China, in 1995. She received the B.E. degree in safety engineering from the China University of Mining and Technology, Beijing, and the M.A.Eng. degree in software reliability from Beihang University. Her research interests include software complex networks, software defect prediction based on software complex networks, and software reuse code detection based on software networks.

W. ERIC WONG received the M.S. and Ph.D. degrees in computer science from Purdue University. He is currently a Full Professor and the Founding Director of the Advanced Research Center for Software Testing and Quality Assurance in Computer Science, The University of Texas at Dallas (UTD). He also has an appointment as a Guest Researcher with the National Institute of Standards and Technology (NIST), an agency of the U.S. Department of Commerce. Prior to joining UTD, he was with Telcordia Technologies (formerly Bellcore) as a Senior Research Scientist and the Project Manager in charge of Dependable Telecom Software Development. In 2014, he was named the IEEE Reliability Society Engineer of the Year. His research interests include helping practitioners improve the quality of software while reducing the cost of production. In particular, he is working on software testing, debugging, risk analysis/metrics, safety, and reliability. He has very strong experience developing real-life industry applications of his research results. He is the Editor-in-Chief of the IEEE TRANSACTIONS ON RELIABILITY. He is also the Founding Steering Committee Chair of the IEEE International Conference on Software Quality, Reliability, and Security (QRS) and the IEEE International Workshop on Debugging and Repair (IDEAR).