An Empirical Study of Regression Bug Chains in Linux

Guanping Xiao, *Student Member, IEEE*, Zheng Zheng, *Senior Member, IEEE*, Bo Jiang, and Yulei Sui

***Abstract*—Regression bugs are a type of bugs that cause a feature of software that worked correctly but stop working after a certain software commit. This paper presents a systematic study of *regression bug chains*, an important but unexplored phenomenon of regression bugs. Our paper is based on the observation that a commit $c1$, which fixes a regression bug $b1$, may accidentally introduce another regression bug $b2$. Likewise, commit $c2$ repairing $b2$ may cause another regression bug $b3$, resulting in a bug chain, i.e., $b1 \rightarrow c1 \rightarrow b2 \rightarrow c2 \rightarrow b3$. We have conducted a large-scale study by collecting 1579 regression bugs and 2630 commits from 57 Linux versions (from 2.6.12 to 4.9). The relationships between regression bugs and commits are modeled as a directed bipartite network. Our major contributions and findings are fourfold: 1) a novel concept of regression bug chains and their formulation; 2) compared to an isolated regression bug, a bug on a regression bug chain is much more difficult to repair, costing 2.4× more fixing time, involving 1.3× more developers and 2.8× more comments; 3) 85.8% of bugs on the chains in Linux reside in Drivers, ACPI, Platform Specific/Hardware, and Power Management; and 4) 83% of the chains affect only a single Linux subsystem, while 68% of the chains propagate across Linux versions.**

***Index Terms*—Bipartite network, bug-fixing commit (BFC), bug-introducing commit (BIC), Linux, regression bug, regression bug chain (RBC).**

I. Introduction

In SOFTWARE repositories, bug reports in bug tracking systems and commits in version control systems are widely utilized and investigated in software engineering research, since they provide valuable historical information of a software project. Mining bug reports and commits is very beneficial for evaluating and understanding software maintenance efforts, such as recovering links between bugs and commits [1]–[7], risk measurement [8]–[10], understanding [11]–[15], detecting [16]–[18], and predicting bugs [19], [20].

Regression bugs are a common type of bugs that lead to a feature of software that worked correctly but stop working after a certain software commit [22]. A regression bug can be caused by a commit fixing an existing bug or an implementation for a new system feature. For example, Linux regression bug ID-51881 was introduced by commit ID-65fe1f0f,[1] whose purpose is to implement a new feature for the SATA device. Previous studies found that regression bugs account for a significant proportion (50.1%) of all classified bugs in Linux [15]. In the Google Chromium project [23], regression bugs occupy about 51.1% of all labeled bugs.

The introduction of a regression bug has close relations with commits. Based on the relationships between bugs and commits, a commit can have one or more of the following three properties, i.e., *bug-fixing*, *bug-introducing*, and *bug-irrelevant*. A commit whose code changes repair a bug is called a *bug-fixing commit* (BFC), while a commit whose code changes inadvertently introduce a bug into the existing project is regarded as a *bug-introducing commit* (BIC). A *bug-irrelevant commit* does not fix or introduce any bugs. When inspecting Linux regression bug reports, we found an interesting type of commit, i.e., *hybrid commit*, which has both the bug-fixing and bug-introducing properties. For example, a hybrid commit $c1$ fixes a regression bug $b1$ but introduces another regression bug $b2$, likewise a new commit $c2$ that repairs $b2$ but also causes a new regression bug $b3$. These special commits together with the bugs can form a *regression bug chain* (RBC), i.e., $b1 \rightarrow c1 \rightarrow b2 \rightarrow c2 \rightarrow b3$. Note that an RBC contains at least two regression bugs and one hybrid commit.

Fig. 1 presents boxplots by comparing the maintenance cost of the bugs on RBCs with that of isolated regression bugs (not on chains) in Linux. All the isolated regression bugs and the first bugs on RBCs are extracted from Linux version 2.6.24. The average fixing time of a bug on an RBC is equal to the ratio of the time difference between the reported time of the first bug and the resolved time of the last bug to the number of related bugs on the RBC, while the fixing time of an isolated regression bug is defined as the time difference between the reported time and the resolved time of the bug.

Compared to an isolated regression bug, fixing a bug on RBCs is much more costly. The average fixing time of a bug on the

Manuscript received August 15, 2018; revised November 29, 2018 and February 8, 2019; accepted February 24, 2019. This work was supported in part by the National Natural Science Foundation of China under Grant 61772055 and Grant 61772056, in part by the Equipment Preliminary R&D Project of China under Grant 41402020102, and in part by the Australia Research Council under Grant DE170101081. Associate Editor: J. Maldonado. *(Corresponding author: Zheng Zheng.)*

G. Xiao and Z. Zheng 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).

B. Jiang is with the School of Computer Science and Engineering, Beihang University, Beijing 100191, China (e-mail: jiangbo@buaa.edu.cn).

Y. Sui is with the Centre for Artificial Intelligence and School of Software, University of Technology Sydney, Sydney, NSW 2007, Australia (e-mail: yulei.sui@uts.edu.au).

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.2019.2902171

[1]Following the common practice, we used the first eight digits to denote commit IDs in this paper.

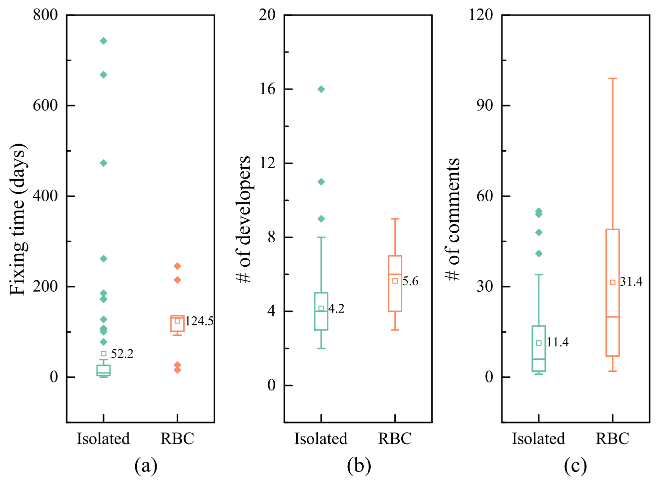


Fig. 1. Boxplots comparing the fixing efforts of the bugs on RBCs with isolated regression bugs. (a) Fixing time, (b) number of developers involved, and (c) number of comments made in bug reports when discovering and finding regression bugs. Note that the results are statistically significant (tested by the Mann–Whitney U test [21], $\alpha = 0.05$).

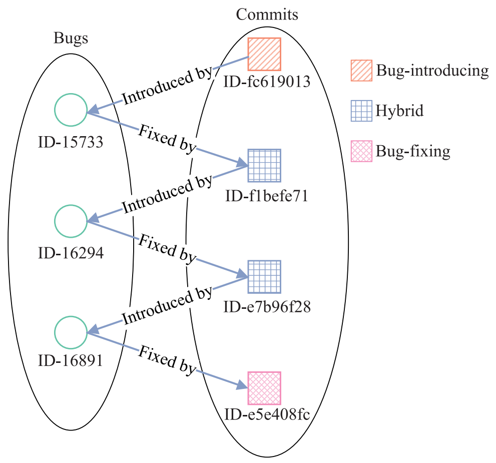


Fig. 2. Real-world example of the RBC in Linux.

RBCs is about 2.4 times longer than that of fixing an isolated regression bug. In addition, fixing a bug on the RBCs involves 1.3× more developers and 2.8× more comments than fixing an isolated regression bug. The efforts made in fixing RBCs significantly increase the cost of software maintenance. However, it is an unexplored research in discovering and understanding the RBCs. It is interesting to formulate, summarize, and understand the RBCs so that we can provide more useful insights for programmers to reduce the maintenance cost by fixing this type of bugs.

This paper proposes a new method to model the bug–commit relationships as a directed bipartite network for analyzing RBCs. The formal definition of the RBC and its related network parameters are given based on the network. The study is conducted using 1579 regression bugs of the Linux kernel from 57 versions and 2630 commits (i.e., BICs and BFCs) collected from the Linux kernel Git repository. This paper mainly focuses on answering the following three research questions.

*RQ1: How to discover and formulate the RBCs in large-scale software systems (*e.g., *Linux)?* In this research question, we investigate how to recover RBCs from bug reports and commits in a large-scale Linux system. More specifically, we will explore the severity and the number of the RBCs and their related bugs and commits.

RQ2: What are the characteristics of RBCs? To have a better understanding of RBCs, it is necessary to explore their characteristics. In this research question, the path lengths of RBCs and their distributions and the features of bugs on RBCs will be investigated. We will also study whether bugs in an RBC would propagate across Linux subsystems or versions.

RQ3: What are the patterns of bug–commit relationships for regression bugs? We will investigate the bug–commit relationships and the patterns of BICs and BFCs, which may reflect the complexity of bugs. For example, a bug can be introduced by one BIC and fixed by one BFC or caused by two BICs and solved by one BFC. We will also investigate the correlation between patterns and the complexity of regression bugs.

This paper makes the following main contributions.

1) To the best of our knowledge, it is the first work to explore the RBCs in Linux and also the first to model the relationships between bugs and commits as a directed bipartite network.
2) Compared to an isolated regression bug, a bug on an RBC is much more difficult to find and repair, costing 2.4× more fixing time, involving 1.3× more developers and 2.8× more comments for discussing and finding the bug.
3) For 71% of the RBCs, the first bug is the most difficult to be fixed.
4) For all the RBCs in Linux, 85.8% of bugs relate to *Drivers*, *ACPI*, *Platform Specific/Hardware*, and *Power Management*.
5) For the developers maintaining more than one subsystem, its proportion for fixing the bugs on RBCs is about 2.3× higher than that for fixing the isolated regression bugs.
6) 83% of RBCs affect only a single Linux subsystem. Bugs on 68% of RBCs are propagated across Linux versions.

The rest of this paper is organized as follows. Section II describes a motivating example of a real-world RBC. Section III presents the directed bipartite network approach for modeling RBCs. Section IV introduces data collection and aggregation. Section V provides the analytical results for three research questions. Section VI discusses the threats to validity, while Section VII introduces related work. Finally, Section VIII concludes this paper.

II. Motivating Example of RBCs

We show a real-world RBC in Linux as our motivating example. Fig. 2 depicts the RBC extracted from Linux version 2.6.32 to version 2.6.35. This RBC is related to the graphics translation table (GTT), which is an input–output memory management unit used by an accelerated graphics port. It took 290 days to eventually fix this complicated RBC (consisting of three bugs), since the first bug on the chain is reported in the Bugzilla of the Linux kernel. The fixing (i.e., commit ID-f1befe71) of

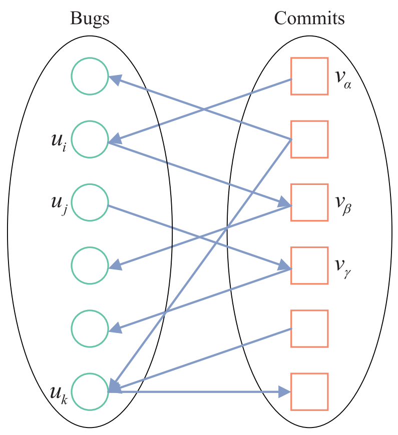


Fig. 3. Illustration of a bug–commit directed bipartite network.

a regression bug (ID-15733) introduces two more regression bugs, and nearly 97 days were spent to fix each bug on the chain. The following is the comment made by the main developer of the Intel-drm/i915 graphics kernel driver in the reports of bug ID-15733 and ID-16294:

> *"I'd like to avoid a regression fix for a regression fix for a regression fix."*
> – *Daniel Vetter*, main developer of Intel-drm driver

Linux regression bug ID-15733 ("Crash when accessing nonexistent GTT entries in i915") was introduced by commit ID-fc619013, which aims to fix the BIOS failures in order to correctly initialize the GTT. The bug was fixed by commit ID-f1befe71 through restricting GTT mapping to a valid range on Intel i915 and i945 chipsets. However, the BFC ID-f1befe71 introduced the regression bug (ID-16294) due to that the new commit fails to detect GTT size on Intel i830 chipsets. The bug was later fixed by commit ID-e7b96f28 ("agp/intel: Use the correct mask to detect i830 aperture size"). Unfortunately, commit ID-e7b96f28 again incorrectly introduced another regression bug (ID-16891), which was finally resolved by commit ID-e5e408fc ("intel-gtt: fix gtt_total_entries detection").

III. Bug–Commit Directed Bipartite Network

In this section, we first introduce our network modeling approach for representing and understanding the relations between bugs and commits on RBCs. Then, we describe the basic ideas and algorithms to analyze RBCs based on the directed bipartite network.

A. Network Modeling

Inspired by the real-world RBC in Fig. 2, we proposed the bug–commit directed bipartite network to model the relations between bugs and commits. As shown in Fig. 3, the relationships between bugs and commits are constructed as a directed bipartite network $G =< U, V, E >$, where U represents the set of bugs and V denotes the set of commits. Let $n = |U|$, $m = |V|$, and $l = |E|$. An edge $(u, v) \in E$ is established from bug u to commit v iff bug u was fixed by commit v. Otherwise, an edge $(v, u) \in E$ is established from commit v to bug u iff bug u was introduced by commit v. Note that there is no cycle in the directed bipartite network, since every commit is assigned to a unique ID based on the time sequence, e.g., a bug can only be connected to a new BFC even if this commit has the same solution as a previous one.

B. Basic Concepts

1) Degree: The degree of a node, denoted as k, in a network represents the number of edges connected to it. There are two types of degrees of a node in directed networks, i.e., out-degree k_{out} and in-degree k_{in}. For a bipartite network, the meanings of the out-degree and in-degree are different for nodes in different sets, i.e., bugs and commits. Given a bug u, the out-degree $k_{\text{out}}(u)$ represents the number of commits fixing u (BFCs), while the in-degree $k_{\text{in}}(u)$ denotes the number of commits introducing u (BICs). On the contrary, for a commit v, the out-degree $k_{\text{out}}(v)$ indicates the number of bugs it introduces, whereas in-degree $k_{\text{in}}(v)$ represents the number of bugs it fixes. For example, as depicted in Fig. 3, k_{out} and k_{in} for bug u_i are both 1, while k_{out} and k_{in} for commit v_α are 1 and 0, respectively.

2) Hybrid Node: Given a commit v, if it satisfies $k_{\text{in}}(v) > 0$ and $k_{\text{out}}(v) > 0$, the commit v is named a hybrid node. Let h denote the number of hybrid nodes in the network. For example, as shown in Fig. 3, commits v_β and v_γ are hybrid nodes. The hybrid node indicates that the commit fixes an existing bug and also introduces a new bug.

3) Path and Its Length: Given two nodes x, y ($x, y \in U \cup V$), a path $P(x, y)$ of them is defined as a sequence of directed edges, which connect a sequence of nodes from x to y. The length L of a path equals the number of edges traversed along the path. For example, as depicted in Fig. 3, the length L of path $P(v_\alpha, v_\beta)$ is 2.

4) Regression Bug Chain: Given two nodes x, y ($k_{\text{in}}(x) = 0$; $k_{\text{out}}(y) = 0$), if $L(P(x, y)) \geq 2$ ($x \in U$) or $L(P(x, y)) \geq 3$ ($x \in V$), the path from x to y is called an RBC. For example, as shown in Fig. 3, there are two RBCs starting from bug u_j and commit v_α, respectively. It is noted that an RBC contains at least one hybrid node and two bugs.

5) RBC Search Algorithm: Fig. 4 presents the RBC search algorithm, which is based on depth-first search [24]. The RBC search algorithm starts at a source node s, whose $k_{\text{in}}(s) = 0$, and records the reachable nodes of s as far as possible along each branch using the stack $Visited$. As shown in lines 14–20, if the reachable node w satisfies $k_{\text{out}} = 0$, and the path from s to w satisfies the length requirement of the RBC definition, we can obtain an RBC starting from the source node s to the reachable node w.

6) Motif: Given a bug u, the motif is defined as the pattern of the relationship between the bug and its corresponding commits (i.e., BIC and BFC). The motif for a bug can be determined by the combination of its in-degrees and out-degrees. For example, as shown in Fig. 3, bug u_i has the one–one relationship motif (i.e., introduced by one commit and fixed by one commit), while bug u_k has the two–one relationship motif (i.e., introduced by two commits and fixed by one commits).

```
Algorithm 1: Regression Bug Chain Search Algorithm
  Data: A directed bipartite network $G$
  Result: All regression bug chains
  foreach $(s \in U \cup V) and (k_{in}(s) = 0)$ do
      let $Stack$ be Stack
      let $Visited$ be Stack
      let $length \leftarrow 0$          //Record the number of nodes in $Visited$
      $Visited.push(s)$
      $Stack.push(s)$
      $length \leftarrow length + 1$
      while $Stack \neq \emptyset$ do
          $v \leftarrow Stack.pop()$
          for all neighbours $w$ of $v$ in network $G$ do
              if $w$ is not in $Visited$ then
                  $Visited.push(w)$
                  $length \leftarrow length + 1$
                  if $k_{out}(w) = 0$ then
                      if $(s \in U) and (length - 1 \geq 2)$ then
                          Output $Visited$
                      else if $(s \in V) and (length - 1 \geq 3)$ then
                          Output $Visited$
                      $Visited.pop(w)$
                      $length \leftarrow length - 1$
                  else
                      $Stack.push(w)$
```

Fig. 4. RBC search algorithm.

TABLE I
DATA SOURCE

Name	Description
Linux regression bug reports	Obtained from [15], including 2020 regression bug reports whose status are fixed and closed.
Linux Git repository commits	Git clone from Linux kernel source tree (upstream tree) on Jan. 8, 2018.

IV. Data Collection and Aggregation

This section presents the details of our data collection and aggregation, including the data source and the data processing procedure.

A. Data Source

As shown in Table I, we utilized two types of data, including Linux regression bug reports and Linux Git repository commits.[2] The Linux regression bugs are obtained from [15]. In this work, among the 4035 classified bugs, there are 2020 regression bugs. Note that we only investigated bug reports with the version numbers starting from 2.6.12. The information of code changes is hard to be obtained before that version because developers only utilized Git to track code changes since version 2.6.12 [25]. As a result, 1907 regression bugs are selected, which account for 94.4% of all regression bugs. The Linux Git repository is downloaded from the Linux kernel source tree (also called *upstream tree*) using the *git clone* command. The changes conducted in Linux development are recorded as commits. In the following section, we will elaborate on the data processing procedure, i.e., the extraction of BICs and BFCs.

[2][Online]. Available: git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

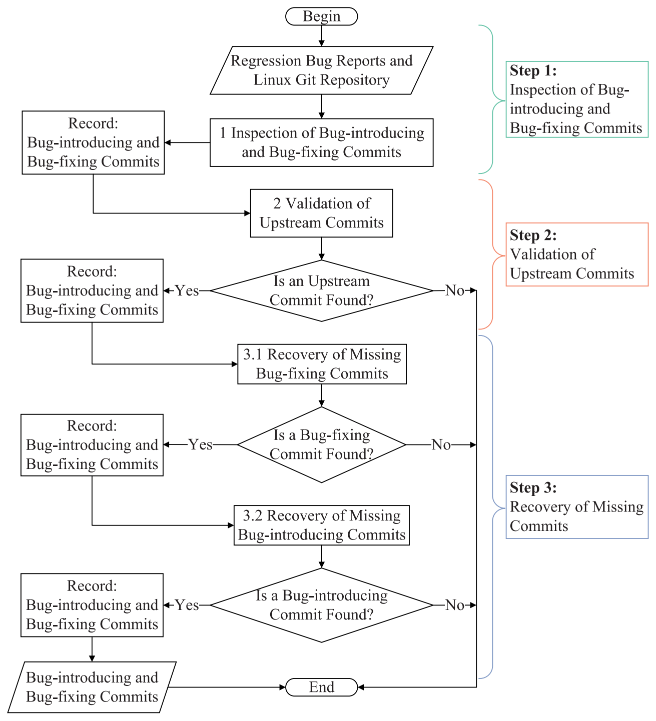

Begin
Regression Bug Reports and Linux Git Repository
1 Inspection of Bug-introducing and Bug-fixing Commits
Record: Bug-introducing and Bug-fixing Commits
2 Validation of Upstream Commits
Is an Upstream Commit Found?
Yes
No
3.1 Recovery of Missing Bug-fixing Commits
Is a Bug-fixing Commit Found?
3.2 Recovery of Missing Bug-introducing Commits
Is a Bug-introducing Commit Found?
Bug-introducing and Bug-fixing Commits
End
Step 1: Inspection of Bug-introducing and Bug-fixing Commits
Step 2: Validation of Upstream Commits
Step 3: Recovery of Missing Commits

Fig. 5. Extraction procedure of BICs and BFCs. Step 1: Inspection of BIC and BFC. Step 2: Validation of upstream commits. Step 3: Recovery of missing commits.

TABLE II
EXAMPLES OF KEYWORDS FOR THE DETERMINATION OF BICS AND BFCS

Commit Types	Keywords
Bug-introducing	first bad commit; offending commit; culprit commit; guilty commit; caused by; introduced by; problem exposed by; the commit breaks;
Bug-fixing	fixed by; fixed with; patch upstreamed; fix has been upstreamed; fixed at; fix has been merged; commit merged in;

B. Data Processing Procedure

To explore the relationships between bugs and commits, we first extract BICs and BFCs. As depicted in Fig. 5, the extraction procedure consists of the following three steps.

1) Step 1. Inspection of BICs and BFCs: As shown in Fig. 5, BICs and BFCs are first inspected in regression bug reports. We manually performed the inspection through several keywords, as illustrated in Table II. For regression bug reports, bug reporters or maintainers tend to describe a BIC as the first bad commit, the offending commit, or the culprit commit. Under certain situations, their descriptions may contain keywords “caused by” or “introduced by” to explain which commit(s) may introduce this bug. By inspecting these keywords or phrases in bug reports, we can obtain BICs. Similarly, there also exist several keywords related to BFCs, such as “fixed by” or “patch upstreamed,” as shown in Table II. A BFC is usually provided at the last comment of a bug report. It is worth noting that there still exist several bugs that we cannot determine their BICs or BFCs after conducting the manual inspection, since these reports did not provide complete information. The missing commit recovery process is handled in step 3.

2) Step 2. Validation of Upstream Commits: BICs and BFCs extracted from bug reports are provided by reporters or maintainers. The provided commits may reside in maintainers trees (i.e., the developer branches of the Linux kernel source tree), but not the upstream tree. This causes a problem that the commit IDs are different though the contents are identical, since they belong to different development trees. For example, the BIC ID of bug ID-59491 is cd7b304d (*x86, range: fix missing merge during add range*). However, the commit ID for the identical content (i.e., *x86, range: fix missing merge during add range*) in the upstream tree is fbe06b7b. Therefore, we should unify the commit IDs by validating them in the upstream tree to eliminate the inconsistency of different commit IDs. We wrote a script based on the *git show* command to automatically check whether commits are in the upstream tree. For the development commits not in the upstream tree, we need to know whether the code changes of these commits are accepted by the upstream or not. We searched these commits using their IDs in Google and tried to find their corresponding code changes, and then, we used the *git log* command together with the *grep* command (with option *-B*) to inspect whether these code changes are really committed in the upstream tree. If the changes of a development commit cannot be found, we discarded this commit from our collected dataset. After step 2, all commits are confirmed as upstream commits, and they can be utilized as input data for Step 3.

3) Step 3. Recovery of Missing Commits: The recovery of missing commits consists of two subphases, i.e., recovery of the incomplete BFCs and recovery of the BICs. Because the recovery of BICs relies on BFCs, the recovery of missing BFCs should be processed first.

a) Step 3.1. Recovery of Missing BFCs: Missing BFCs are recovered through searching their bug IDs in Linux Git repository. For example, the BFC that fixes bug ID-22672 cannot be found in its report from step 1. Therefore, we used the *git log* command together with the *grep* command (with option *-B*) to search the bug ID in the upstream tree to determine which commit (i.e., ID-47356eb6) fixed this bug. Worse, several bug reports specify fixing patches without providing fixing commit IDs. In this case, we searched the patch message in the upstream tree to obtain their commit IDs. For example, a fixing patch entitled "*NFS: Fix a hang/infinite loop in nfs_wb_page()*" was provided in bug ID-29062. We used the *git log* command together with the *grep* command (with option *-B*) to search the patch message. Finally, we obtained the BFC ID-b8413f98 that fixes this bug.

b) Step 3.2. Recovery of Missing BICs: We recovered missing BICs based on a popular approach for identifying bug-introducing changes, i.e., the SZZ approach [1], which was proposed by Śliwerski, Zimmermann, and Zeller in 2005. The SZZ approach first inspects BFCs by searching for the bug IDs in the logs of version control systems (e.g., Git, CVS, and SVN). Once the BFCs are obtained, the changed lines of code for fixing the bug are identified. SZZ traces back based on the code history of

```
Commit:  4b00e4b3···
Author: John Stanley <jpsinthemix@verizon.net>
Message: savagedb: Fix typo causing regression in savage4 series video chip detection
- - - a/drivers/video/savage/savagefb.h
+++b/drivers/video/savage/savagefb.h
@@ -55,7 +55,7 @@
-#define S3_SAVAGE4_SERIES(chip)   ((chip>=S3_SAVAGE4) || (chip<=S3_PROSAVAGEDDR))
+#define S3_SAVAGE4_SERIES(chip)   ((chip>=S3_SAVAGE4) && (chip<=S3_PROSAVAGEDDR))
```

(a)

```
Commit: cc406341···
Author: Tormod Volden <debian.tormod@gmail.com>
Message: savagefb: New S3_TWISTER and S3_PROSAVAGEDDR chip families
@@ -52,14 +51,15 @@
- - - a/drivers/video/savage/savagefb.h
+++b/drivers/video/savage/savagefb.h
...
-#define S3_SAVAGE4_SERIES(chip)   ((chip==S3_SAVAGE4) || (chip==S3_PROSAVAGE))
+#define S3_SAVAGE4_SERIES(chip)   ((chip>=S3_SAVAGE4) || (chip<=S3_PROSAVAGEDDR))
```

(b)

Fig. 6. Commits of bug ID-39842. (a) BFC ID-4b00e4b3. (b) BIC ID-cc406341.

TABLE III
COLLECTED DATA FOR LINUX FROM VERSIONS 2.6.12 TO 4.9

# of Bugs	# of Commits	# of BICs	# of BFCs
1579	2630	1148	1542

version control systems to find the time when the changed code was introduced. According to the SZZ approach, we used the *git log* command (with options *-p -M –follow –stat*) together with the *grep* command (with option *-B*) to search for BICs for the bugs whose BFCs are already available, but the commits that introduce those bugs are not directly available.

For example, as depicted in Fig. 6(a), the BFC ID-4b00e4b3 of bug ID-39842 deletes the code "*#define S3_SAVAGE4_SERIES(chip) ((chip > = S3_SAVAGE4) || (chip < = S3_PROSAVAGEDDR))*" in the file "*drivers/video/savage/savagefb.h*." After conducting SZZ search, it is found that the deleted line of code was introduced in commit ID-cc406341, as shown in Fig. 6(b). Note that the SZZ approach cannot recover a BIC by tracing a BFC, which contains newly added lines of code, since this code was introduced for the first time. To ensure the validity of the collected data, we only recover BICs for the bugs, whose BFCs only contain code changes in a single file. If the code changes of a BFC were conducted in multiple files, there may have several commits to trace back. It is difficult to determine which commit is the BIC because the result produced by the SZZ approach is not guaranteed to be sound [26]. Moreover, it is unnecessary to conduct step 2 for the recovered commits, since the recovery of missing commits is conducted on the upstream tree.

After the extraction of BICs and BFCs, we collected 2630 commits related to 1579 regression bugs for the Linux kernel from versions 2.6.12 to 4.9, as shown in Table III. Among the collected commits, there are 1148 BICs and 1542 BFCs. Note that each bug in the collected data possesses at least one commit. Finally, we have released our collected datasets found online.[3]

According to the network modeling approach described in Section III-A, we constructed a directed bipartite network based on the collected bugs and commits from Linux version 2.6.12 to

[3][Online]. Available: https://guanpingxiao.github.io/data/linux_rgbugs.xlsx

TABLE IV
NUMBERS AND PROPORTIONS OF RBCs

# of RBCs	# of Bugs (%)	# of Commits (%)
100	133 (8.4)	168 (6.4)

4.9 shown in Table III. There are 4029 nodes (i.e., 1579 bugs and 2630 commits) and 2872 edges. The average degrees of bugs $< k(u) >$ and commits $< k(v) >$ are 1.82 and 1.09, respectively. The result indicates that each bug has 1.82 commits, and each commit has 1.09 bugs on average. In addition, the number of hybrid nodes h is 60. Note that all the network calculations in this study are based on *NetworkX*,[4] a Python package for analyzing graphs and networks.

V. ANALYSIS

This section presents the analytical results of our three research questions in terms of findings and implications.

A. *RQ1: How to Discover and Formulate the RBCs in Large-Scale Software Systems (e.g., Linux)?*

After performing the RBC search algorithm provided in Section III on the constructed directed bipartite network, we obtained the numbers and the proportions for RBCs shown in Table IV. The number of hybrid nodes (i.e., a commit is both a bug-fixing and BIC) is 60. Although the hybrid nodes only account for 2.3% of all commits, they introduce 100 RBCs, and the number of related bugs and commits is 133 and 168, respectively. The proportions of the bugs and commits on the RBCs over all the regression bugs and commits are 8.4% and 6.4%, respectively.

In the Linux Bugzilla, a bug is reported by a user or developer through a custom drop-down field in the reporting page specifying whether a bug is a regression bug. A reported regression bug is then further confirmed by developers. By carefully analyzing the reports of the bugs on RBCs, most of the regression bugs are found after installing system updates, e.g., introducing new system features and/or new BFCs. An abnormal functionality of devices or failures occurred while the system using these new updates. In addition, to further analyze the causes of the bugs on RBCs, we manually examine the BFCs of RBC bugs (i.e., 133 bugs). It is found that 73.7% of the bugs on RBCs are functional bugs, while the rest of bugs are related to concurrency bugs (14.3%) and memory-related bugs (12.0%). For the functional bugs, i.e., the causes relate to the implementation of specific functionalities (e.g., device drivers), the high proportion is due to the lack of regression test cases to cover the code changes. Given limited test inputs, it is hard for developers to validate a new commit that can work correctly on all related hardware platforms. For example, the fixing commit ID-1a7c618a of bug ID-12302 adds the functionality to support a specific kind of BIOSes (Asus Laptops). For the concurrency bugs (e.g., data race: bug ID-15819, deadlock: bug ID-14924)

[4][Online]. Available: https://networkx.github.io/

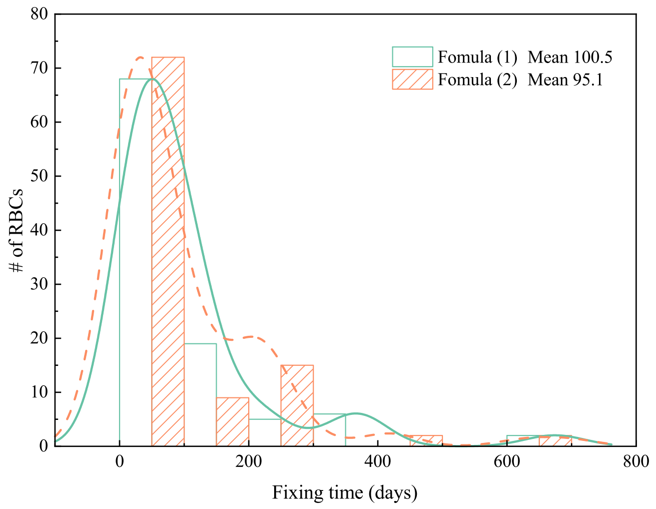


Fig. 7. Distributions of average fixing time of the bugs on RBCs using two calculation formulas.

and memory-related bugs (e.g., null pointer dereference: bug ID-14030, memory leak: bug ID-13518), it is useful to apply some static code analysis tools for detecting these types of bugs [27]–[29].

To understand the complexity of RBCs, we measured the average fixing time of the bugs on RBCs using two methods. The first method is calculated as follows:

$$< t_{\text{fixing}} >= (d^{b_n}_{\text{resolved}} - d^{b_1}_{\text{reported}})/n \tag{1}$$

where $d^{b_n}_{\text{resolved}}$ and $d^{b_1}_{\text{reported}}$ represent the resolved date of the last bug b_n and the reported date of the first bug b_1 on the chain, respectively, and n is the number of bugs on the chain. In this formula, the gaps between the resolved date of bug b_i and the reported date of bug b_{i+1} are included in the average fixing time. In order to analyze the impact of the gaps, we defined another calculation method

$$< t_{\text{fixing}} >= \sum_{i=1}^{n} (d^{b_i}_{\text{resolved}} - d^{b_i}_{\text{reported}})/n. \tag{2}$$

Fig. 7 shows the distributions of average fixing time of the bugs on RBCs using formulas (1) and (2). The result was tested using the *Mann–Whitney U* test [21] with a null hypothesis that the times calculated by the two formulas have similar values. After performing the test, we obtained $p = 0.159$, which is larger than the given significance level of $\alpha = 0.05$. Therefore, we cannot reject the hypothesis. The difference in the mean values in Fig. 7 is negligible, i.e., the mean value of the average fixing time calculated by formula (1) is only 5.7% longer than the mean value from formula (2). The average fixing time calculated by the two formulas is very similar. This is because that the Linux kernel is one of the most popular open-source software with a large community consisting of many active users and developers. The side effects of a BFC, i.e., introducing new bugs, are often quickly observed by users and/or developers. We used the first formula, which is more intuitive, to calculate the average fixing time of the bugs on RBCs in this study (e.g., Fig. 1).

TABLE V
LENGTHS OF RBCs

$< L >$	L_{min}	L_{max}	$L = 3$	$L = 4$	$L = 6$
			# of RBCs		
3.6	3	6	50	45	5

Although the proportion of the bugs on the RBCs is not very high, these types of bugs are very hard to repair. For example, the repairing time of RBCs occupies 43.6% of the total fixing time of all regression bugs in Linux 2.6.24, which significantly increases the total maintenance efforts.

Moreover, we have investigated the individual fixing time of each bug on an RBC to understand which bug is more "difficult" to be fixed. We defined the difference of the fixing time as $\Delta F = t_{\text{fixing}}^{b_{i+1}} - t_{\text{fixing}}^{b_i}$. It is found that for 71% of RBCs, the first bug is the most difficult to be fixed.

To discuss a regression bug's "latency," we have also collected the time from the moment a bug is reported until it receives its first comment. It is found that the average time of the first comment for RBC bugs is 6.0 days, while the average time of the first comment for isolated regression bugs is 5.8 days. However, the result is not statistically significant, which was tested by the *Mann–Whitney U* test [21] with a null hypothesis that the times of the first comments for RBCs bugs and isolated regression bugs have similar values.

Finding #1: *The numbers of RBCs, related bugs, and commits are 100, 133, and 168, respectively. The related bugs and commits account for 8.4% and 6.4% over all bugs and commits in the directed bipartite network.*

Finding #2: *For 71% of RBCs, the first bug is the most difficult to be fixed.*

Implication: *The efforts in fixing Linux RBCs are nonnegligible, and it is also interesting to investigate the characteristics of RBCs in other software systems. In addition, developers should pay more attention to the bugs that are difficult to fix, as their fixing commits are likely to introduce new bugs.*

B. RQ2: What are the Characteristics of RBCs?

We have investigated the characteristics of RBCs from four aspects, including the lengths of RBCs, the distribution of the bugs on RBCs across Linux subsystems, bug propagation across Linux subsystems and/or versions, and the largest weakly connected components in the directed bipartite network.

1) Lengths of RBCs: We have conducted a statistic of the lengths of RBCs, as shown in Table V.

It can be observed that the average length of RBCs is 3.6, while the shortest and the longest lengths of RBCs are 3 and 6, respectively. Note that if both the BIC and the BFC of a bug can be found, the length L of an RBC satisfies $L = 2n$, where n is the number of related bugs on the chain. In Table V, there are 50 out of 100 RBCs whose lengths are 3, and 45 RBCs whose lengths are 4. In addition, there are five chains whose lengths are 6. The longer an RBC is, the more efforts developers take when fixing the RBC.

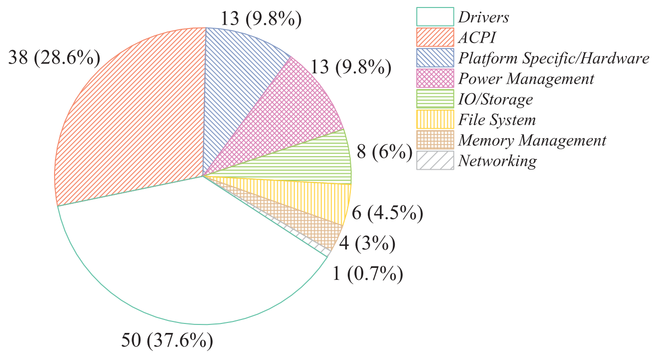


Fig. 8. Distribution of the bugs on RBCs across Linux subsystems.

Finding #3: *The length of RBCs is from 3 to 6. In addition, the lengths of 50% of RBCs are 3 and the lengths of 45% of RBCs are 4.*

Implication: *Understanding the relations between bugs and commits is helpful for understanding a software project and reducing its maintenance cost. Based on the proposed directed bipartite network, the lengths of RBCs can be utilized to measure the effectiveness and the maintenance efforts in bug-fixing processes. If the first bug on an RBC is fixed in a low quality, it is likely to produce a longer RBC.*

2) Distribution of Bugs on RBCs Across Linux Subsystems: We have investigated the distribution of the bugs on RBCs across Linux subsystems, as shown in Fig. 8.

We observed that bugs related to *Drivers*, *ACPI*, *Platform Specific/Hardware*, and *Power Management* occupy 85.8% of all bugs on RBCs. Note that these four subsystems are closely related to device drivers and architecture platforms. This indicates that bugs related to these subsystems are more likely to appear on RBCs. According to the study [30], the number of functions in the *drivers* directory of the Linux source code accounts for approximately half of all functions in the Linux source code. The fast growing of various devices and platforms requires frequent software development iterations and system updates. Updating code in a driver-related software component in Linux tends to be more error-prone, and many driver-related parts are likely written by less experienced software developers, making new patches that fix existing bugs but introducing new bugs. Therefore, for RBCs in Linux, the high proportion of bugs in these subsystems is reasonable.

To further validate the result of the distribution of the bugs on RBCs across Linux subsystems, we calculated statistics of the locations, where the bugs are fixed in the BFCs for RBCs, as shown in Fig. 9. It can be observed that 92.4% of the bugs on the chains are fixed in the *drivers* and *arch* directories. The source code of *Drivers*, *ACPI*, and *Power Management* mainly lo-

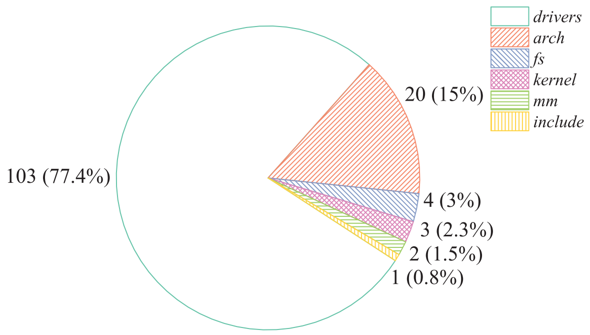


Fig. 9. Distribution of the bugs on RBCs across locations of the Linux source code.

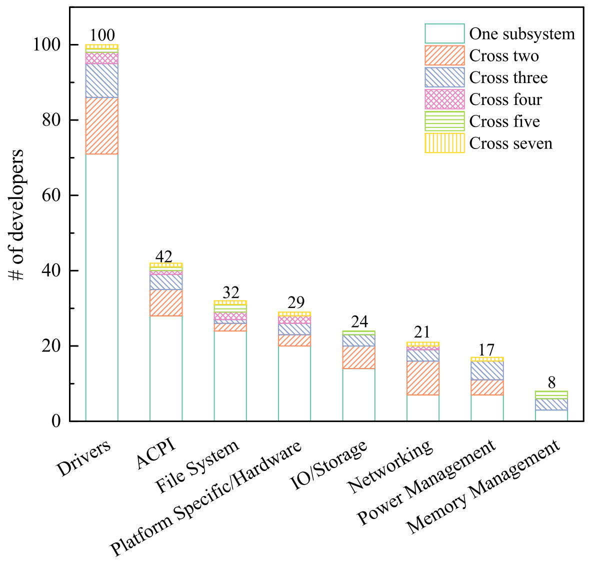


Fig. 10. Developer distributions of subsystems.

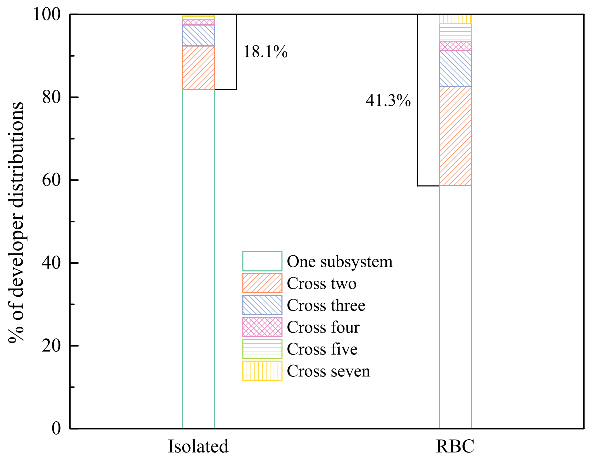


Fig. 11. Proportions of developer distributions for the isolated and RBC bugs.

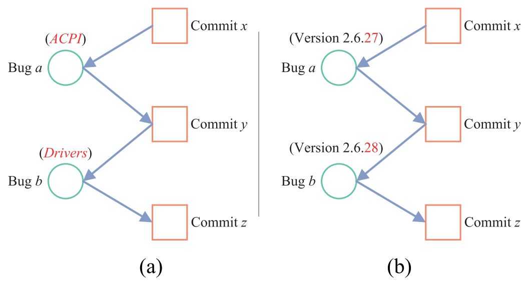


Fig. 12. Illustration of a bug propagating on RBCs. (a) Across Linux subsystems. (b) Across Linux versions: the major version numbers of the two bugs are different.

cates in the *drivers* directory, while the source code of *Platform Specific/Hardware* mainly locates in the *arch* directory. Thus, the distribution of locations where the bugs are fixed is consistent with the result in Fig. 8.

We have investigated the relationships between RBCs and the developers of these subsystems. To obtain distributions of developers of different subsystems, we have performed statistics of bug assignees, i.e., persons in charge of resolving bugs, based on 1579 collected bug reports. Fig. 10 shows the distributions of the developers of the eight subsystems containing RBCs. Note that the number of bugs in these subsystems accounts for 91.6% of all collected bugs. Since Linux subsystems have tight coupling relationships [31], it can be observed from Fig. 10 that around 25–65% of the total developers work on more than one subsystem with some experienced developers work cross seven subsystems.

Moreover, Fig. 11 depicts the proportions of developer distributions of isolated regression bugs and RBC bugs. For the developers maintaining more than one subsystem, its proportion for fixing RBC bugs is about 2.3× higher than that for fixing isolated regression bugs. The result indicates that the bugs on RBCs require more experienced developers, i.e., the ones who are familiar with more than one subsystem.

Finding #4: *Bugs related to Drivers, ACPI, Platform Specific/Hardware, and Power Management are likely to appear on RBCs. For all the RBCs of Linux, 85.8% of bugs relate to Drivers, ACPI, Platform Specific/Hardware, and Power Management. 92.4% of the bugs on the RBCs are fixed in the drivers and arch directories of the Linux source code.*

Finding #5: *For the developers maintaining more than one subsystem, its proportion for fixing RBC bugs is about 2.3× higher than that for fixing isolated regression bugs.*

Implication: *For bugs related to Drivers, ACPI, Platform Specific/Hardware, and Power Management, it is suggested to conduct more regression testing before releasing their fixes. For the bugs that have impacts on several subsystems, it is suggested to first analyze the closely related subsystems.*

3) RBC Bugs Propagation Across Linux Subsystems or Versions: We have investigated the propagation of the bugs on an RBC across different Linux subsystems or versions. As the example presented in Fig. 12(a), the affected subsystems of bugs are different, which clearly indicates that bugs on RBCs are propagated across subsystems. Similarly, as shown in Fig. 12(b), since the versions of bugs have different major version numbers, bugs on the RBC propagate across versions. According to the

TABLE VI
BUG PROPAGATING ACROSS LINUX SUBSYSTEMS OR VERSIONS ON RBCs

Subsystems	# of RBCs	Versions	# of RBCs
Yes	17	Yes	68
No	83	No	32
Total	100	Total	100

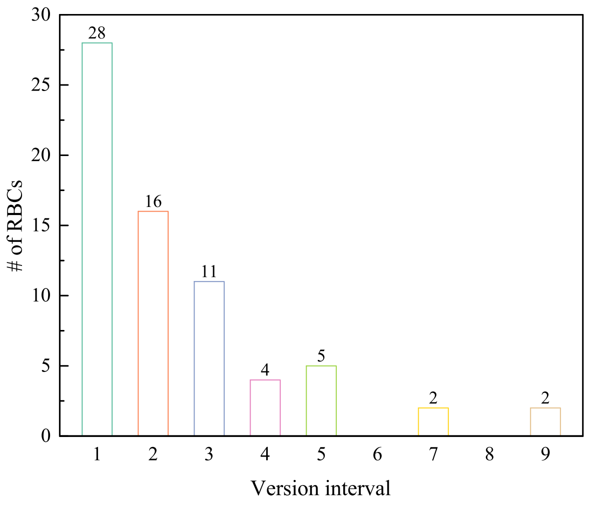


Fig. 13. Version interval of the bugs on RBCs having bug propagation.

Linux version numbering method [30], [32], the third digit represents the major version number from versions 2.6.11 to 2.6.39, while the second digit denotes the major version number from version 3.0. For example, version 2.6.28 is a major version, whereas version 2.6.28.7 is a minor version of 2.6.28. Table VI shows our investigation results.

There are 83 RBCs on which the bugs are not propagated across subsystems. We can draw a conclusion that most of the RBCs affect a single subsystem. However, there are 68 out of 100 RBCs, whose bugs propagate across versions. The result implies that more than two-thirds (i.e., 68%) of RBCs are exposed in later major Linux versions, whereas 32% of RBCs are exposed in the same major versions.

In order to investigate the interval between versions of the first bug and the last bug in an RBC, we further conducted statistics of version intervals of the RBCs, which have bug propagation, as depicted in Fig. 13. Note that the version interval is computed as the difference between major version numbers. As the example shown in Fig. 12(b), the version interval of the RBC is 1, since the major version numbers of bugs a and b are 27 and 28, respectively. It can be found from Fig. 13 that there are 55 out of 100 RBCs with version intervals 1, 2, or 3. This indicates that the last bugs in 55% of RBCs are exposed after no more than three major version intervals. However, there are still 13% of RBCs, in which the last bugs are exposed after no less than four major version intervals or even to be exposed after night major version intervals.

4) Largest Weakly Connected Component Related to RBCs: We analyzed the largest weakly connected component on the directed bipartite network to understand the severity of an RBC.

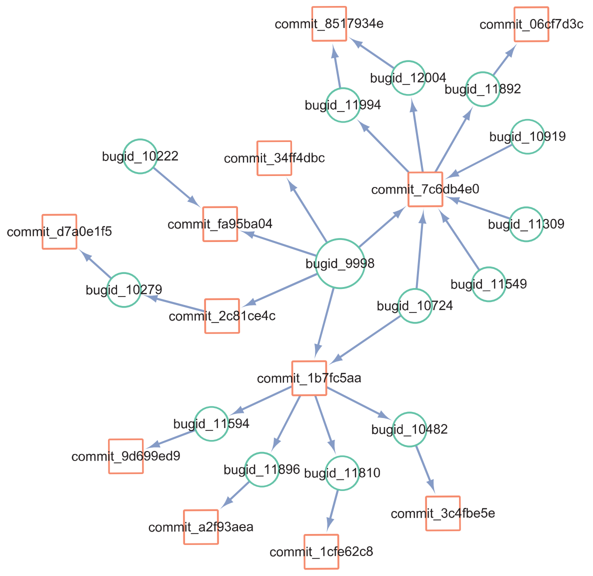


Fig. 14. Largest weakly connected component of the directed bipartite network.

Finding #6: *83% of RBCs affect only a single Linux subsystem. Bugs on 68% of RBCs are propagated across versions. Moreover, the last bugs in 13% of RBCs are exposed after no less than four major version intervals.*

Implication: *Regression bugs are very annoying to both developers and users. Even if a newly released version of Linux may offer new features and security enhancements, the users may not prefer to upgrade their operating systems if the release contains an RBC. It is more stable to keep using an older Linux version before the RBC is eventually fixed. Worse, more than two-thirds of RBCs propagate across versions. Releasing a stable version highly relies on the effective fixing of RBCs.*

The largest weakly connected component is the maximal subnetwork of the directed bipartite network G such that every pair of nodes (x, y) are connected to each other by some path, ignoring the direction of edges. As shown in Fig. 14, the largest weakly connected component consists of 14 bugs, 12 commits, and 24 RBCs. In the largest weakly connected component, eight RBCs were initially started from bug ID-9998, and the largest version interval of these RBCs is 4. Furthermore, all bugs in the largest weakly connected component are related to the *ACPI* subsystem.

C. RQ3: What are the Patterns of Bug–Commit Relationships for Regression Bugs?

We investigated the patterns (i.e., motifs described in Section III-B) found in the directed bipartite network. Since some bugs have only one type of commits (i.e., BFCs or BICs), we excluded these bugs to ensure result validity. Therefore, 1128 regression bugs are selected. After conducting the calculation, ten motifs were found. Fig. 15 gives the numbers and

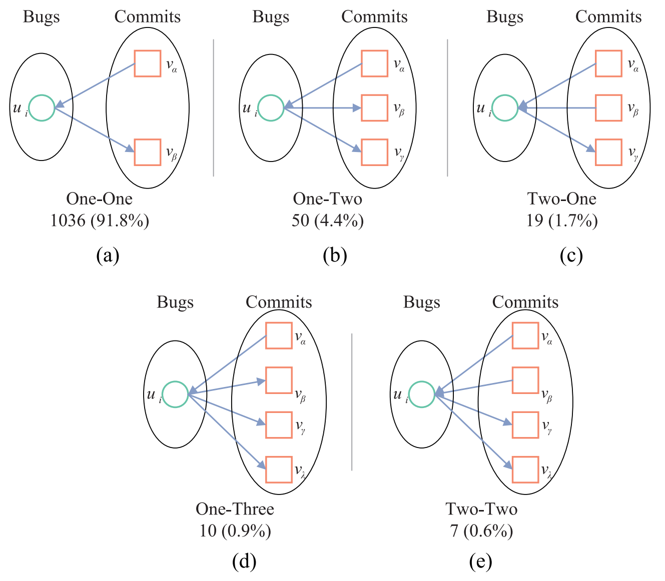


Fig. 15. Top five frequent motifs found in the directed bipartite network: (a) one–one, (b) one–two, (c) two–one, (d) one–three, and (e) two–two.

Finding #7: *In the worst case, improperly handling fixes of one bug can introduce at least eight RBCs. In addition, the largest weakly connected component related to RBCs consists of at least 14 bugs, 12 commits, and 24 RBCs.*

Implication: *Similar bugs retrieval and developer recommendation are important tasks in the automated bug report management process [33]. The directed bipartite network is useful for analyzing similar bugs and recommending these bugs to developers. Combining with metadata in bug reports and commits, it can further utilize heterogeneous information network techniques (e.g., HIN2Vec [34]) to train a deep learning model for the prediction of similar bugs and developer recommendations.*

proportions of the top five frequent motifs. Note that the number of bugs in the top five motifs accounts for 99.4% of the selected bugs. Most of the regression bugs (91.8%) have the one–one relationship motif (i.e., introduced by one commit and fixed by one commit). The second and the third frequent motifs are one–two relationship (i.e., introduced by one commit and fixed by two commits) and two–one relationship (i.e., introduced by two commits and fixed by one commit), which accounts for 4.4% and 1.7% of the selected bugs, respectively.

In order to analyze the relation between motifs and the complexity of regression bugs, we calculated the average fixing time of the bugs in each motif. To ensure the validity of the results, all the motifs with the two–two relationship are excluded from our analysis, since the number of bugs is less than 10. The fixing time of each regression bug is estimated as the time difference between the reported time and the resolved time (i.e., resolution marked as CODE_FIX). Boxplots in Fig. 16 compare the fixing times of bugs in each motif. The result was tested by the *Kruskal–Wallis* test [35], with a null hypothesis that the

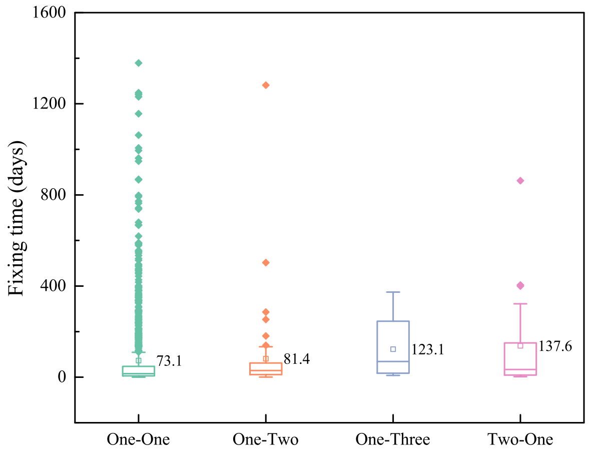


Fig. 16. Boxplots comparing the fixing times of bugs in each motif.

motifs have similar fixing times. For a significance level of $\alpha = 0.05$, we obtained $p = 0.008$, which indicates that we can reject the null hypothesis. The average fixing time of regression bugs increases when the number of BFCs increases. Compared to regression bugs that have one BIC, the average fixing time of regression bugs, which have two BICs, is significantly longer. The result indicates that motifs are good factors to reflect the complexity of regression bugs.

Finding #8: *Most of the regression bugs (91.8%) have the one–one relationship motif, and the proportions of regression bugs, which have one–two and two–one relationship motifs, are 4.4% and 1.7%, respectively. The average fixing time of regression bugs with more BFCs is longer than those with fewer BFCs. Likewise, a bug with more BICs is also much more costly to be fixed compared to the one with fewer BICs.*

Implication: *Motif can be utilized to measure the complexity of regression bugs. For example, a bug that has a two–one motif (i.e., caused by two BICs) is likely to be more complicated than a bug that has a one–two motif (i.e., caused by one BIC).*

VI. Threats to Validity

A. Internal Threats

Threats to internal validity come from experiments, i.e., manual inspection and recovering a bug's introducing and fixing commits in Linux bug reports and its Git repository. We have carefully examined the reports and commits from Linux version 2.6.12 to 4.9 using around three-month time with two persons.

The second threat is about the correctness of data information. Since bug reports are reported by users and developers, the correctness of the provided information (e.g., subsystems and versions) may have an impact on the results of relevant analyses in this paper.

The last threat comes from the calculation of bug fixing time. The fixing time of a bug is estimated as the time when the bug's report opens until it is resolved. It does not reflect the actual time a developer spent in fixing the bug. In addition, we used the bug fixing time obtained from the bug reports as the criteria for evaluating the complexity of a bug. However, two developers may have different capabilities and levels of experience in repairing the same bug, i.e., spending different bug fixing times.

B. External Threats

Threats to external validity come from the generalization of our results. Linux is one of the most important open-source projects in the world. We believe that this paper is representative. In addition, we have conducted the study based on 1579 regression bugs and 2630 commits from 57 Linux versions (from 2.6.12 to 4.9). However, we do not try to claim our findings or conclusions reflect all software. The prevalence and characteristics of RBCs are interesting to be explored in any other software projects. The proposed methodology is applicable for any project with version control and bug tracking systems.

VII. Related Work

A. Bug–Commit Relationships

Links between bugs in bug tracking systems and commits in version control systems can provide valuable information for software maintenance. Analyzing the relationships between bugs and commits is widely conducted over the past ten years. Related studies can be roughly classified as three categories, including recovering links between bugs and BFCs, recovering links between bugs and BICs, and studying characteristics of bug-fixing/introducing commits. These kinds of literature are discussed in the following paragraphs.

Bachmann *et al.* [3] found that developers do not always describe which commits conduct bug fixings, and it was reported that only 46% of bugs in Apache project are linked with bug fixes. Traditional heuristics methods for collecting links between bugs and commits are conducted through searching for keywords and bug IDs. Wu *et al.* [4] developed an automatic link recovery based on the criteria of features from explicit links to recover missing links. They obtained a better result than the traditional heuristics. Along this line of research, several improved algorithms are proposed [5]–[7].

The recovery of BICs relies on BFCs. In 2005, Śliwerski *et al.* [1] proposed SZZ, an approach for identifying bug-introducing changes. The proposed approach recovers BICs by tracing back the changed code in BFCs through code history to find its introduction commits. Based on the approach in [1], Kim *et al.* [2] presented algorithms to automatically and accurately identify BICs. In addition, several empirical studies related to BICs in software projects are presented for investigating Android [36] and Google Chromium project [23]. Moreover, the evaluation of the SZZ approach is a challenging task, since the ground truth is not readily available. To address the problem, researchers in [26] proposed a framework to evaluate the results of alternative SZZ implementations. The framework can provide a systematic way to evaluate the data generated by a given SZZ implementation.

Besides, there are several studies focusing on the characteristic analysis of BFCs and BICs. Shihab *et al.* [8] studied the risk of software changes in a large enterprise. The findings showed that the criteria for determining risky changes are different from developers and teams. Eyolfson *et al.* [9] studied the correlation between a commit's time-based characteristics and its "bugginess" in three open-source projects: the Linux kernel, PostgreSQL, and the Xorg server. It was found that commits between midnight and 4 A.M. are significantly buggier. To understand how the erroneous tendency of software developer changes across time, Li *et al.* [10] investigated the bug-introducing tendency of developers. They found that the BIC rates of developers tend to increase first before decreasing.

Most of the existing literature focuses on separate bug–commit links and rarely analyzes the connections between bug–commit links (i.e., the relationship among bugs, BICs, and BFCs). Compared to the existing work, we analyzed the bug–commit relationships by modeling bugs and commits as a directed bipartite network.

B. Regression Bugs

Nir *et al.* [22] found that regression bugs were usually introduced by bug fixes. They developed a tool for assisting the programmer to locate the lines of code causing a given regression bug. Khattar *et al.* [23] investigated regression bugs and identified the code changes introducing the regression bugs in Google Chromium project. It was found that 51.1% of labeled bugs are regression bugs. In addition, more than half of regression bugs possess high priorities. Recently, Xiao *et al.* [15] have reported that 50.1% of classified bugs in Linux are regression bugs, and regression bugs are more likely to be bohrbugs, i.e., bugs that can be consistently reproduced under a well-defined set of conditions since their activation and/or error propagation are simple. To the best of our knowledge, our study is the first work to investigate RBCs in Linux and their characteristics based on the proposed directed bipartite network.

C. Mining Software Repositories Based on Bipartite Networks

The bipartite networks are appropriate for modeling the relationships between two disjoint entities. The authors of [37] and [38] modeled the relations between developers and software modules (binaries) as a contribution bipartite network. Based on the network, it was found that central modules are more failure prone than modules located in surrounding parts of the contribution network. Dittrich *et al.* [39] described the ownership between authors and source files in Audacity project as a bipartite network for identifying key authors and subject matter experts. Schall [40] modeled the user repository as a directed bipartite network to introduce an approach for recommending relevant users to follow in large-scale online development communities. The approach was tested using a GitHub-based dataset and obtained excellent results regarding context-sensitive following recommendations.

VIII. Conclusion

In this paper, we presented a large-scale empirical study of RBCs in the Linux kernel based on 1579 regression bugs and 2630 commits from a bipartite network perspective. First, we proposed the modeling of the bug–commit relationships as a directed bipartite network and introduced a novel concept of the RBC based on the network. Then, we introduced the data source and data processing procedure. The analysis was performed on three aspects: the prevalence of RBCs in Linux, the characteristics of RBCs, and the patterns of the bug–commit relationships. Along with eight findings and their implications, our results provided useful insights into the software maintenance process for large-scale real-world software systems.

Acknowledgment

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

References

[1] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. Int. Workshop Mining Softw. Repositories*, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[2] S. Kim *et al.*, "Automatic identification of bug-introducing changes," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2006, pp. 81–90. [Online]. Available: https://doi.org/10.1109/ASE.2006.23

[3] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 97–106. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882308

[4] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: Recovering links between bugs and changes," in *Proc. 19th ACM SIGSOFT Symp./13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 15–25. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025120

[5] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 63. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393671

[6] T.-D. B. Le, M. Linares-Vásquez, D. Lo, and D. Poshyvanyk, "RCLinker: Automated linking of issue reports and commits leveraging rich contextual information," in *Proc. IEEE 23rd Int. Conf. Program Comprehension*, 2015, pp. 36–47. [Online]. Available: https://doi.org/10.1109/ICPC.2015.13

[7] Y. Sun, Q. Wang, and Y. Yang, "FRLink: Improving the recovery of missing issue-commit links by revisiting file relevance," *Inf. Softw. Technol.*, vol. 84, pp. 33–47, 2017. [Online]. Available: https://doi.org/10.1016/j.infsof.2016.11.010

[8] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 62. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393670

[9] J. Eyolfson, L. Tan, and P. Lam, "Correlations between bugginess and time-based commit characteristics," *Empirical Softw. Eng.*, vol. 19, no. 4, pp. 1009–1039, 2014. [Online]. Available: https://doi.org/10.1007/s10664–013-9245-0

[10] Y. Li, D. Li, F. Huang, S.-Y. Lee, and J. Ai, "An exploratory analysis on software developers' bug-introducing tendency over time," in *Proc. Int. Conf. Softw. Anal., Test. Evol.*, 2016, pp. 12–17. [Online]. Available: https://doi.org/10.1109/SATE.2016.9

[11] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. 13th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2008, pp. 329–339. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346323

[12] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng.*, 2013, pp. 178–187. [Online]. Available: https://doi.org/10.1109/ISSRE.2013.6698917

[13] 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, 2014. [Online]. Available: https://doi.org/10.1007/s10664–013-9258-8

[14] 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, 2016. [Online]. Available: https://doi.org/10.1016/j.jss.2015.11.021

[15] 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 28th Int. Symp. Softw. Rel. Eng.*, 2017, pp. 101–111. [Online]. Available: https://doi.org/10.1109/ISSRE.2017.21

[16] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proc. Int. Symp. Softw. Test. Anal.*, 2012, pp. 254–264. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336784

[17] Y. Sui, D. Ye, Y. Su, and J. Xue, "Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions," *IEEE Trans. Rel.*, vol. 65, no. 4, pp. 1682–1699, Dec. 2016. [Online]. Available: https://doi.org/10.1109/TR.2016.2570538

[18] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 327–337. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180178

[19] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proc. 7th IEEE Work. Conf. Min. Softw. Repositories*, 2010, pp. 31–41. [Online]. Available: https://doi.org/10.1109/MSR.2010.5463279

[20] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proc. 11th Work. Conf. Min. Softw. Repositories*, 2014, pp. 182–191. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597078

[21] E. A. Gehan, "A generalized Wilcoxon test for comparing arbitrarily singly-censored samples," *Biometrika*, vol. 52, nos. 1–2, pp. 203–224, 1965. [Online]. Available: https://doi.org/10.1093/biomet/52.1–2.203

[22] D. Nir, S. Tyszberowicz, and A. Yehudai, "Locating regression bugs," in *Proc. 3rd Int. Haifa Verification Conf.*, 2007, pp. 218–234. [Online]. Available: https://doi.org/10.1007/978-3-540-77966-7_18

[23] 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. 8th India Softw. Eng. Conf.*, 2015, pp. 50–59. [Online]. Available: http://doi.acm.org/10.1145/2723742.2723747

[24] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972. [Online]. Available: https://doi.org/10.1137/0201010

[25] 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. [Online]. Available: https://doi.org/10.1007/978-3-642-15579-6_10

[26] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, Jul. 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2616306

[27] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Trans. Softw. Eng.*, vol. 40, no. 2, pp. 107–122, Feb. 2014. [Online]. Available: https://doi.org/10.1109/TSE.2014.2302311

[28] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proc. 25th Int. Conf. Compiler Constr.*, 2016, pp. 265–266. [Online]. Available: http://doi.acm.org/10.1145/2892208.2892235

[29] Y. Sui and J. Xue, "Value-flow-based demand-driven pointer analysis for C and C++," *IEEE Trans. Softw. Eng.*, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2018.2869336

[30] G. Xiao, Z. Zheng, and H. Wang, "Evolution of Linux operating system network," *Phys. A: Statist. Mech. Appl.*, vol. 466, pp. 249–258, 2017. [Online]. Available: https://doi.org/10.1016/j.physa.2016.09.021

[31] H. Wang, Z. Chen, G. Xiao, and Z. Zheng, "Network of networks in Linux operating system," *Phys. A: Statist. Mech. Appl.*, vol. 447, pp. 520–526, 2016. [Online]. Available: https://doi.org/10.1016/j.physa.2015.12.084

[32] 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, 2010. [Online]. Available: https://doi.org/10.1016/j.jss.2009.09.042

[33] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Trans. Softw. Eng.*, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2018.2870414

[34] T.-Y. Fu, W.-C. Lee, and Z. Lei, "HIN2Vec: Explore meta-paths in heterogeneous information networks for representation learning," in *Proc. ACM Conf. Inf. Knowl. Manage.*, 2017, pp. 1797–1806. [Online]. Available: http://doi.acm.org/10.1145/3132847.3132953

[35] N. Breslow, "A generalized Kruskal-Wallis test for comparing k samples subject to unequal patterns of censorship," *Biometrika*, vol. 57, no. 3, pp. 579–594, 1970. [Online]. Available: https://doi.org/10.1093/biomet/57.3.579

[36] M. Asaduzzaman, M. C. Bullock, C. K. Roy, and K. A. Schneider, "Bug introducing changes: A case study with android," in *Proc. 9th IEEE Working Conf. Min. Softw. Repositories*, 2012, pp. 116–119. [Online]. Available: https://doi.org/10.1109/MSR.2012.6224267

[37] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 2–12. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453105

[38] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, "Putting it all together: Using socio-technical networks to predict failures," in *Proc. 20th Int. Symp. Softw. Rel. Eng.*, 2009, pp. 109–119. [Online]. Available: https://doi.org/10.1109/ISSRE.2009.17

[39] A. Dittrich, M. H. Gunes, and S. Dascalu, "Network analysis of software repositories: Identifying subject matter experts," in *Complex Networks*. New York, NY, USA: Springer, 2013, pp. 187–198. [Online]. Available: https://doi.org/10.1007/978–3-642-30287-9_20

[40] D. Schall, "Who to follow recommendation in large-scale online development communities," *Inf. Softw. Technol.*, vol. 56, no. 12, pp. 1543–1555, 2014. [Online]. Available: https://doi.org/10.1016/j.infsof.2013.12.003

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

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

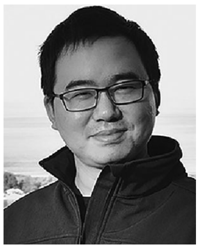
Zheng Zheng (SM'18) received the Ph.D. degree in computer software and theory from the Chinese Academy of Sciences, Beijing, China, in 2006.

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

Bo Jiang received the Ph.D. degree from the University of Hong Kong, Hong Kong, in 2011.

He is currently an Associate Professor in computer science with the School of Computer Science and Engineering, Beihang University, Beijing, China. His research has been reported in leading journals and conferences such as ASE, FSE, ICWS, QRS, TSC, TRel, JSS, IST, and SPE. He serves as a program committee member for many conferences. His current research interests include software testing, debugging, and blockchain technology.

Dr. Jiang is a Guest Editor for the *Journal of Systems and Software*. He received four Best Paper Awards from his conference publications.



Yulei Sui received the Ph.D. degree in computer science from the University of New South Wales, Sydney, Australia, in 2014.

He is a Faculty Member in computer science with the University of Technology Sydney, Sydney, NSW, Australia. He is broadly interested in the research field of software engineering and programming languages, in particular in static and dynamic program analysis for software bug detection and compiler optimizations.

Dr. Sui is a recipient of an ICSE Distinguished Paper Award, a CGO Best Paper Award, and an Australian Discovery Early Career Researcher Award 2017–2019.