Characterizing and Detecting Bugs in WeChat Mini-Programs

Tao Wang*
Qingxin Xu*
Xiaoning Chang
State Key Lab of Computer Sciences, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
{wangtao19,xuqingxin19,changxiaoning17}@otcaix.iscas.ac.cn

Wensheng Dou†‡
Jiaxin Zhu‡
State Key Lab of Computer Sciences, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
{wsdou,zhujiaxin}@otcaix.iscas.ac.cn

Jinhui Xie
Yuetang Deng
Jianbo Yang
Jiaheng Yang
Tencent, Inc.
Guangzhou, China
{hugoxie,yuetangdeng,xiaotuoyang,jiahengyang}@tencent.com

Jun Wei†
Tao Huang
State Key Lab of Computer Sciences, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
{wj,tao}@otcaix.iscas.ac.cn

ABSTRACT

Built on the WeChat social platform, WeChat Mini-Programs are widely used by more than 400 million users every day. Consequently, the reliability of Mini-Programs is particularly crucial. However, WeChat Mini-Programs suffer from various bugs related to execution environment, lifecycle management, asynchronous mechanism, etc. These bugs have seriously affected users' experience and caused serious impacts.

In this paper, we conduct the first empirical study on 83 WeChat Mini-Program bugs, and perform an in-depth analysis of their root causes, impacts and fixes. From this study, we obtain many interesting findings that can open up new research directions for combating WeChat Mini-Program bugs. Based on the bug patterns found in our study, we further develop WeDetector to detect WeChat Mini-Program bugs. Our evaluation on 25 real-world Mini-Programs has found 11 previously unknown bugs, and 7 of them have been confirmed by developers.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Software and its engineering → Software testing and debugging**.

*Tao Wang and Qingxin Xu contribute equally.
†Wensheng Dou and Jun Wei are the corresponding authors.
‡Wensheng Dou and Jiaxin Zhu are also affiliated with Nanjing Institute of Software Technology and University of Chinese Academy of Sciences, Nanjing, China.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510114

KEYWORDS

WeChat Mini-Programs, empirical study, bug detection

ACM Reference Format:
Tao Wang, Qingxin Xu, Xiaoning Chang, Wensheng Dou, Jiaxin Zhu, Jinhui Xie, Yuetang Deng, Jianbo Yang, Jiaheng Yang, Jun Wei, and Tao Huang. 2022. Characterizing and Detecting Bugs in WeChat Mini-Programs. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510114

1 INTRODUCTION

WeChat is one of the most popular social platforms, running on mobile and desktop operating systems, e.g., Android, iOS, Windows and macOS. A recent study has reported that WeChat is widely used in daily life, and there are more than one billion monthly active users [43].

WeChat Mini-Programs run on the WeChat platform, and can utilize various functions provided by WeChat, e.g., user management, file access, network access, location, camera, and payment. WeChat Mini-Programs do not have a palpable installation process as Android and iOS applications, but users can have an experience as smooth as native Android and iOS applications. WeChat Mini-Programs have been widely used in many scenarios, e.g., e-commerce, transaction, and education. Nowadays, there are more than 5.5 million Mini-Programs running on the WeChat platform, and there are more than 400 million WeChat users using Mini-Programs every day [57].

Based on the Mini-Program framework [15], developers can develop various WeChat Mini-Programs. The Mini-Program framework contains two layers: a render layer and a logic layer. The render layer is responsible for UI display. For the purpose, WXML (WeiXin Markup Language) and WXSS (WeiXin Style Sheets) are used as its description language, which are similar to HTML and CSS in traditional web applications. The logic layer is responsible for the program logic written in JavaScript. The two layers run on

two independent threads respectively and communicate with each other through an asynchronous and event-driven mechanism. The Mini-Program framework also provides rich components and APIs for developers to implement their own services. Most of these APIs are asynchronous, thus making the entire Mini-Program response smoothly.

Unfortunately, developing high-quality WeChat Mini-Programs is a challenging task. When developing WeChat Mini-Programs, developers need to properly handle asynchronous mechanism, platform differences, lifecycle management, etc. Otherwise, various bugs can occur in WeChat Mini-Programs, e.g., conventional JavaScript bugs, asynchronous related bugs, and compatibility bugs. We call WeChat Mini-Program Bugs as *WeBugs* for short. More than 70 million of errors caused by WeBugs are reported in the WeChat internal monitoring system every day. WeBugs can seriously affect users' experience and lead to serious consequences, e.g., economic losses and information leakages. To understand, detect and fix WeBugs is of great importance to WeChat Mini-Program developers.

Substantial empirical studies about software bugs have been conducted to enrich our knowledge on software reliability. Existing empirical study on mobile applications mainly focus on various bugs in Java-based Android applications, e.g., compatibility bugs [67, 88, 90], security-related bugs [68, 79, 106, 108]. Existing JavaScript-related empirical studies mainly focus on the client-side JavaScript applications [94, 95, 97] and server-side JavaScript applications [101–103]. These bugs are usually associated with dynamic JavaScript features and client-side JavaScript bugs are mostly DOM-related. However, WeChat Mini-Programs are developed based on the WeChat Mini-Program framework, which are greatly different from client/server-side JavaScript applications and Android applications. Thus, these studies cannot reflect the key characteristics of WeBugs. Existing works on WeChat Mini-Programs mainly focus on their development [76, 105], acceptance and usage [63, 86]. Therefore, there is little knowledge about WeBugs in literature. Understanding WeBugs is of significant interest to our researchers and practitioner community, and can be helpful in providing guidance on bug avoidance, detection, testing and fixing, etc.

In this paper, we conduct the first empirical study on WeChat Mini-Program bugs. We collect 83 WeBugs from three sources, i.e., open-source WeChat Mini-Programs in GitHub, QAs from the developer forum and online running Mini-Programs maintained by a WeChat development Company X. We study WeBugs from the following three research questions.

- **RQ1 (Root cause):** What are the root causes of WeChat Mini-Program bugs?
- **RQ2 (Fix strategy):** How do developers fix WeChat Mini-Program bugs? Are there common fix patterns?
- **RQ3 (Bug Impact):** What impacts do WeChat Mini-Program bugs have?

Through our in-depth analysis against the above three aspects, we obtain many interesting findings and useful lessons for further research and practice. For example, 14.2% WeBugs are caused by incompatible features across different devices, even though WeChat Mini-Programs are designed as cross-platform applications. Thus, developers should pay attention to compatibility-related WeBugs.

```
// index.wxml
1.  <view> Hello {{name}}! </view>
2.  <button bindtap="changeName">
3.          Click me! </button>
```

```
// index.wxss
4.   button{
5.     background-color: blue;
6.     width: 70%;
7.     margin-top: 70%;
8.  }
```

```
// index.js
9.  var helloData = {
10.   name: 'WeChat'
11. }
12. Page({
13.   data: helloData,
14.   changeName: function (e) {
15.     //sent data change to view
16.     this.setData({
17.       name: 'World'
18.     })
19.   }
20. })
```

(b)

(a) (c)

Figure 1: A simplified WeChat Mini-Program.

Based on our empirical study, we further design a static analysis tool, *WeDetector*, to detect WeBugs under three specific bug patterns, e.g., incorrect platform-dependent API usage, incomplete layout adaptation to specific devices, and non-specific handling for arguments passed to callbacks. We apply WeDetector on 25 popular WeChat Mini-Programs from GitHub and find 11 previously unknown WeBugs, in which 7 WeBugs have been confirmed by developers. We have made our studied WeBugs and WeDetector publicly available at https://github.com/tao2years/WeBug.

In summary, we make the following main contributions.

- We present the first empirical study on WeChat Mini-Program bugs from three aspects, i.e., root cause, bug impact and fix strategy. Our in-depth analysis reveals common vulnerabilities in WeChat Mini-Programs.
- Our empirical study obtains many findings that can open up new research directions. We hope our study can shed lights on combating WeChat Mini-Program bugs.
- We design a static WeBug detection tool, WeDetector, and find 11 new bugs in 25 popular WeChat Mini-Programs.

The rest of this paper is organized as follows. Section 2 presents an overview of WeChat Mini-Programs and its framework. Section 3 presents our study methods. Section 4 presents our empirical study results. Section 5 discusses lessons learned from our study. Based on our study results and lessons, we propose a WeBug detection tool in Section 6. Section 7 discusses related work and finally Section 8 concludes this paper.

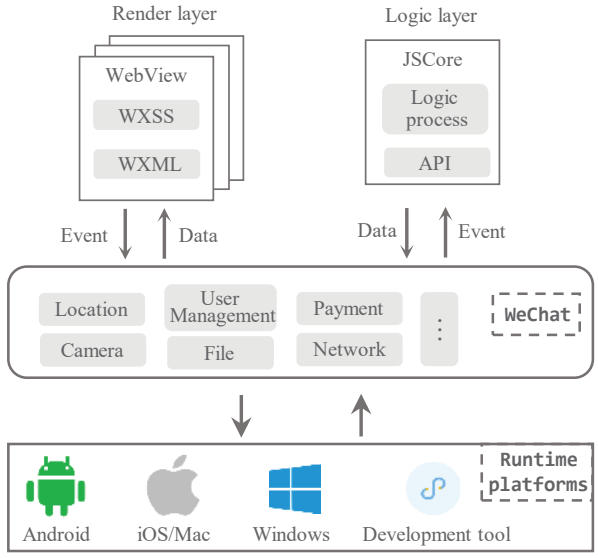


Figure 2: The WeChat Mini-Program framework.

2 WECHAT MINI-PROGRAMS

WeChat Mini-Programs run on the social WeChat platform, which have billions of users. On WeChat platform, users can easily find Mini-Programs by scanning related QR code or searching the names, and use them without downloading and installing process like Android and iOS applications. Benefited from the WeChat social platform, many Android and iOS applications also propose their corresponding WeChat Mini-Program versions, e.g., Amazon, Airbnb and KFC. Nowadays, WeChat Mini-Programs have been widely used in our daily life.

A WeChat Mini-Program. Figure 1 shows a simplified WeChat Mini-Program. Figure 1a shows the code while Figure 1b and Figure 1c show the display page of the program. This program contains three files, `index.wxml`, `index.wxss`, and `index.js`. File `index.wxml` builds the view of the page. File `index.wxss` manages the style of the components in page. File `index.js` decides the program logic written in JavaScript. From Figure 1a, we can see that the function `changeName` is bound on the button "Click me!" (Line 2-3), and property `name` (Line 1) is bound with the `name` in double curly brace (Line 10) in `index.wxml`. When the button in Figure 1b is clicked, the display turns to that in Figure 1c.

WeChat Mini-Program framework. Figure 2 shows the WeChat Mini-Program framework, which mainly contains a render layer and a logic layer. The render layer, e.g., `index.wxml` and `index.wxss` in Figure 1a, runs on the `WebView` thread, and the logic layer, e.g., `index.js` in Figure 1a, runs on JSCore thread to execute JavaScript code. Note that, similar to Android applications [2, 100], Mini-Programs and their pages have complicated lifecycle management, e.g., application launch, page load, and page show.

The logic layer and render layer communicate with each other through an asynchronous and event-driven mechanism. When users click the button "Click me!" on the page in Figure 1b, it will trigger the binding function `changeName` (Line 2) in the `index.wxml`. The `WebView` thread sends the event `changeName` through WeChat Mini-Program platform. The JSCore thread then executes the corresponding event handler (Line 14). When data changes occur in the logic layer, the data is passed from the logic layer to the WeChat Mini-Program platform via the `setData` method, and then forwarded to the render layer. When the render layer notices `name` has been changed, it then changes the page's display. The WeChat Mini-Program framework provides various functions, e.g., user management, file access, network access, location, camera, and payment. In order to make Mini-Programs respond smoothly, most of these APIs work in an asynchronous and event-driven manner.

Table 1: WeChat Mini-Program Execution Environments

Platform	Logic Layer	View Layer
iOS/Mac	JavaScriptCore [49]	WKWebView [58]
Android	V8 [54]	Tencent XWeb [53]
Development tool	NW.js [9]	Chromium Webview [5]
Windows	Chrome kernel [7]	Chrome kernel

Runtime platforms. WeChat Mini-Programs can run on different platforms. Mobile and desktop operating systems, including Android, iOS, Windows, and macOS are supported by the WeChat Mini-Program framework. These platforms have different implementations of the Mini-Program framework. Table 1 illustrates the different execution environments for logic layer and render layer on different platforms.

Although the WeChat Mini-Program framework intends to provide a cross-platform development and runtime environment, the underlying runtime platforms can cause intricate differences to the execution of Mini-Programs. For example, JavaScriptCore [49] for iOS9 and iOS10 is not fully compatible with the ECMAScript 6 standard [6], and Tencent XWeb [58] rewrites some components in Chrome kernel [7], e.g., video. Therefore, Mini-Programs can behave differently when incompatible features are used.

3 STUDY METHODOLOGY

In this section, we first present how to collect and analyze WeBugs, and then explain the threats to our empirical study.

3.1 Collecting WeBugs

We select WeBugs from three sources, i.e., open source GitHub Mini-Programs, official WeChat Mini-Program developer forum [56] and online running Mini-Programs maintained by a WeChat development Company X. These sources can provide various dimensions for WeBugs. Finally, we obtain 83 WeBugs from these sources. In the following, we explain how we collect WeBugs from them.

Open source Mini-Programs. Many open source WeChat Mini-Programs can be found in GitHub. To collect WeBugs from GitHub, we first collect GitHub repositories related to Mini-Programs. We use three keywords, "Mini-Program", "weapp" , "wxapp" to search GitHub repositories. We also include all open source repositories listed by the *awesome-wechat-weapp* project [45] in our study. Project *awesome-wechat-weapp* records most popular WeChat Mini-programs and development materials on GitHub, with over 35.8k

Table 2: Study Subjects from Github Mini-Programs.

Project	Category	Stars	Issues	Bugs
ColorUI [23]	Style	10.4k	296	3
iview-weapp [26]	Component	6k	389	3
wechat-app-mall [20]	Shopping	14.4k	189	9
wxParse [14]	Text parser	7.6k	327	12
winxin-watch-life [21]	Blog	2k	46	2
wx-calendar [22]	Calendar	1.9k	293	7
wechat-app-demo [12]	DevTool	1.7k	18	4
WeHalo [30]	Blog	1.1k	31	1
pinche-xcx [17]	Traveling	620	20	1
sculplus-wechat [27]	Daily life	554	509	4
Weapp-trending [29]	News	259	6	1
threejs-example [33]	Graph	106	22	1
CardOnePerson [38]	Notebook	27	3	1
Total				**49**

stars. Through this step, we obtain 13,365 repositories. We remove invalid repositories that do not have the entry file `app.js` for WeChat Mini-Programs, and further remove repositories with less than one issue, since they cannot contain valid information for our study. After that, we obtain 255 distinct WeChat Mini-Program repositories, and they contain more than 4,000 issues. We notice that most issues are about how to use or improve Mini-Programs. We ignore this kind of issues in our study and select WeBugs with these issues through the following steps. (1) We read the description of each issue report and select the issue if it has a clear description and available fix. (2) For issue reports having clear descriptions without fixes, we try to find out a fix solution. If we can find a fix solution, we also include them in the study. Finally, we obtain 49 WeBugs from 13 open source GitHub Mini-Programs, as shown in Table 2. These WeChat Mini-Programs cover a wide spectrum of categories, e.g., style libraries and daily services. We can also see that most of these WeChat Mini-Programs are popular, because they obtain many stargazers in GitHub, and contain hundreds of issues.

QAs in the official WeChat Mini-Program developer forum. WeChat Mini-Program developer forum [56] is a technical exchange community maintained by the official WeChat platform. Any questions related to Mini-Programs can be posted in the forum. On average, hundreds of questions are posted every day. Some of these questions can reflect WeBugs encountered by individual developers.

To extract WeBugs from the developer forum, we go through the Questions and Answers (QAs) posted from 2021.1.1 to 2021.3.4, and obtain more than 10,000 QAs. We find that many QAs are not related to WeBugs, e.g., developers ask how to use new features in the WeChat Mini-Program. We exclude these QAs from consideration. We consider a QA as a WeBug if it satisfies the following conditions. (1) The QA describes an issue when developers implement their WeChat Mini-Programs. (2) The QA contains code snippets or clearly describes the code related to the QA. Thus, we can reproduce the issues by ourselves. (3) The QA was solved, and the related WeBug was fixed. We obtain 16 WeBugs after the screening.

Online running Mini-Programs. During the collaboration with a WeChat development Company X, we have the opportunity to collect WeBugs from online running Mini-Programs. Online running Mini-Programs are similar to Android and iOS applications published on the market, respectively. For the issues provided by Company X, we compare their original and fixed versions, and analyze their error stack information. If we can fully understand a WeBug, we take it into consideration. Finally, we obtain 18 WeBugs from 16 online running Mini-Programs.

3.2 Analyzing WeBugs

We thoroughly studied these 83 WeChat Mini-Program bugs and tried to answer the research questions we raised before. All collected WeBugs were manually analyzed by three authors in this paper. We performed an in-depth analysis on the 83 WeBugs by investigating their source code, discussions, fix patterns, etc.

To build the taxonomies, we adopt open card sorting approach. To reduce bias, WeBugs are grouped and labeled by three authors independently. Next, we discuss their results to reach a consensus. We sort WeBugs using the following approaches.

- **Root cause.** We try to answer the question "what caused this bug at the code level". (1) Based on the stack trace and messages in issue report, we locate the buggy code and figure out what caused the bug. Take the message `result is not defined` as an example, we find the locations where the *result* is used, where the *result* is passed from, and why it is not defined. (2) If the issue report does not include the stack trace, we manually reproduce this bug to obtain it. (3) If we cannot reproduce the bug, we compare the buggy code and corresponding fix patch.
- **Fix strategy.** The fix patch is obtained from associated pull request. If there does not exist associated pull request, we search and analyze commit log of the project to obtain the fix code. Finally, we manually extract the common patterns by abstracting the patches.
- **Bug impact.** The qualitative impact of most WeBugs can be inferred by the issue reports straightforwardly. If developers do not provide enough information, we reproduce the bugs to check their results.

In our study process, we also referred to some studies which classified the software bugs [60, 94, 95, 110] for the initial classifications of root causes. With further analysis, we gradually refined these root causes, e.g., removed the unmatched and added new ones.

During our study, we also reproduced some WeBugs if related information are available.

3.3 Threats to Validity

Internal validity. Our empirical study's design and planning follow the structure suggested by Wohlin [64]. We carefully study each WeBug's source code, discussion and fix patch to have a deeper understanding. When any categories change during the study, all WeBugs need to be re-analyzed again. All the collected WeBugs and analysis results have been discussed and confirmed by at least three authors of this paper. For bugs that we do not agree on, we turn to developers and Company X for help. If a bug cannot be fully understood by us, we do not take it into consideration. Thus, we

believe that all studied WeBugs are true positives and have been thoroughly studied.

External validity. We may miss some known WeBugs in the selection process, and the selected WeBugs may not cover all kinds of WeBugs. However, our dataset has covered the investigated space of the WeChat Mini-Programs and has good representativeness. From Figure 2, we can observe that bugs can occur in some situations, i.e., the adaptation of different operating systems, original WeChat functionalities, regular JavaScript code and asynchronous mechanism. We have studied more than 4000 issues and 10000 QAs, covering most of the popular open source projects. Our dataset does cover these bugs.

4 WEBUG STUDY RESULTS

In this section, we present the detailed results of our empirical study on 83 WeBugs from three aspects, i.e., root cause, bug fix and bug impact.

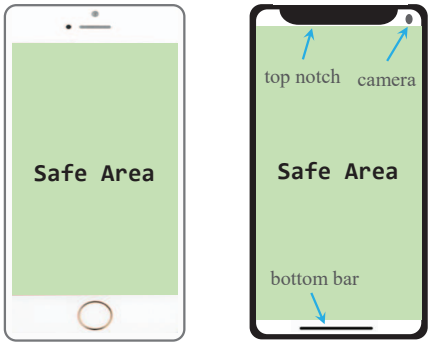
4.1 Root Cause

As shown in Table 3, we identify six types of root causes.

Differences in execution environments. Mini-Programs are designed to be cross-platform, and it has an internal mechanism to handle platform differences when interpreting the code of render layer and logic layer. However, 18 (21.7%) WeBugs are caused by the discrepancies among platform execution environments. These bugs are related to platform-dependent API usage, inconsistent rendering among different devices, and different versions of the Mini-Program framework.

Platform-dependent APIs. Some APIs in the Mini-Program framework are OS-specific, e.g., they do not support all OS or have different behaviors on different OS. Compatibility bugs occur when invoking the same API results in inconsistent behavior on different device models. For example, *wx.onShareTimeline* only works in Android. *Date* returns different values between Android and iOS. The issue wechat-app-mall#237 [52] reports that users cannot access a page on iOS devices. This WeBug is caused by the inconsistent time format between Android and iOS. Developers invoke API *Date* to obtain the system time. The return time on Android is split with "-" (e.g., 2021-1-1), while "/" (e.g., 2021/1/1) on iOS. Developers do not normalize the value into the same format, and introduce this bug. Nine WeBugs are related to platform-dependent APIs.

Note that, Android applications also suffer from compatibility issues caused by platform-dependent APIs [106]. However, platform-dependent APIs in WeChat Mini-Programs have different characteristics from those in Android. Platform-dependent APIs in WeChat Mini-Programs are not only related to hardware but also related to software platforms. We have summarized 13 platform-dependent APIs in Table 4, in which six APIs are related to hardware, e.g., bluetooth, NFC, and video. Four APIs have certain restrictions on different platforms, e.g., different parameters, different return values and different data format. For example, the type of return value of API *wx.onCompassChange* on iOS device is *Number* while on Android device is *String*. If these APIs are handled properly, similar compatibility bugs can be avoided.



(a) Normal screen (b) Anomalous screen

Figure 3: Safe areas of normal screen and anomalous screen.

> **Finding 1**: *Some APIs in the Mini-Program framework are platform-dependent. If developers do not handle these APIs properly, compatibility bugs can occur.*

Inconsistent rendering among different devices. Although the layout of Mini-Programs is self-adaptive, there are also three abnormal rendering bugs. These bugs arise in specific devices due to their anomalous screens. For some devices, e.g., Figure 3a, all their screen can be used to render applications. However, for some devices, e.g., Figure 3b, parts of their screen are occupied by some components, e.g, camera. Thus, rendering applications on the restricted areas can cause abnormal displays. The issue colorUI#50 [40] reports that the height of the side drawer is incorrect on iPhoneX, in which there is a bottom bar on the screen and the bottom bar occupies certain position. Developers should adapt the layout settings based on `safeArea` of the screen in different devices.

Note that, previous rendering issue studies in Android [81, 106] observe that rendering issues are mainly caused by different screen sizes. WeChat Mini-Programs have been able to adjust the UI adaptively according to screen sizes. Our study finds that WeChat Mini-Programs should pay more attention to *safeArea* in mobile phones.

> **Finding 2**: *Although the layout of Mini-Programs is self-adaptive in rendering, developers still need to adjust them to fit the safe areas on some devices with anomalous screens.*

Inconsistent behaviors due to different versions of the Mini-Program framework. There have been two major releases of the Mini-Program framework, and the APIs are changing across different versions. For example, when the Mini-Program framework was updated from 1.0 to 2.0, the property `shape` is no longer supported for `i-button` [18]. It is difficult for developers to notice the changes in the Mini-Program framework and to modify their programs accordingly. Additionally, some features are not correctly supported by certain framework versions, e.g., some settings in component `textarea` [47] and API `wx.previewMedia` [44] cannot work properly. Six WeBugs belong to this case.

Table 3: Root Causes of 83 WeChat Mini-Program Bugs

Root cause		WeBugs	Ratio
Differences in execution environments	APIs have restricted usages on specific OS, e.g., Android and iOS.	9	10.8%
	Inconsistent rendering among different devices.	3	3.6%
	Inconsistent behaviors due to different versions of Mini-Program framework.	6	7.2%
	Subtotal	**18**	**21.7%**
API misuse	Using inappropriate APIs.	4	4.8%
	Using inappropriate API parameters.	2	2.4%
	Using improper style priority settings.	4	4.8%
	Subtotal	**10**	**12.0%**
Syntax errors	Spelling mistakes in JavaScript code.	4	4.8
	Forgetting to define necessary variables and functions.	11	13.3%
	Subtotal	**15**	**18.1%**
No verification of data validity	Forgetting to check the validity of return data from external requests.	16	19.3%
Asynchronous errors	Concurrency errors caused by asynchronous mechanism and lifecycle events.	3	3.6%
Logic errors	Incorrect or incomplete implementation logic.	21	25.3%
Total		**83**	**100%**

Finding 3: *Different versions of the Mini-Program framework can introduce discrepancies, and thus cause WeBugs.*

API misuse. Developers may misunderstand the APIs / components of the Mini-Program framework, and introduce errors due to incorrect usage. 10 (12%) WeBugs are caused by API misuse, including inappropriate API selection, inappropriate API parameters and incorrect settings of style priority.

Inappropriate API selection. Developers may have wrong understanding of the API functionality provided by the Mini-Program framework. Thus, using inappropriate APIs cannot satisfy developers' expectations, and introduces bugs. Four WeBugs belong to this case. WeBug CardOnePerson#2 [36] is such an example. In a textarea, developers expect that the page automatically updates after the user modifies the page content. However, they bind a `contentChange` function by using `bindblur`, which is triggered when the focus of the page changes. Thus, when users click `confirm update` directly after their modifications, the focus of the textarea does not change, and the page does not update accordingly. In this case, developers should use `bindinput` instead of `bindblur`.

Inappropriate API parameters. Developers may have wrong assumptions, and use inappropriate parameters for the APIs provided by the Mini-Program framework. Two WeBugs belong to this case. For example, in WeBug wechat-app-mall#203 [39], the developer invokes the payment API by executing `wxapi.payBill ({A, B})`, and only pass one parameter ({A, B}). However, the payment API `wxapi.payBill` requires two parameters `(A, B)` and should be used as `wxapi.payBill (A, B)`. The developer combines the two parameters into a single object, and thus makes the payment fail.

Incorrect setting of style priority. Developers may not have a fully understanding of style priority. Figure 4 explains style priority with a simplified example. Figure 4a shows the style setting code, and Figure 4b shows the display. We can see that the global style setting in `index.wxss` of button.background-color is blue while the button color is set green in `index.wxml`. However, the final color of the button is green, because inner settings have higher priority than global settings. Improper style priority settings can introduce unexpected rendering bugs, especially when third-party style libraries are used. Four WeBugs belong to this case.

```
// index.wxml
<button
  style= "background-color: green">
  Click me!
</button>
```

```
// index.wxss
 button{
  background-color: blue;
  width: 80%;
  margin-top: 50%;
}
```

(a)

(b)

Figure 4: A simplified example of style priority setting.

Finding 4: *Developers may not have a fully understanding of style priority and introduce some non-functional issues like unexpected rendering performance.*

Syntax errors. The Mini-Program framework adopts JavaScript as its programming language. Due to the dynamic feature of JavaScript, Mini-Programs can still run when encountering syntax errors. We observe that syntax errors are common in Mini-Programs, and 15 (18.1%) WeBugs belong to syntax errors. Syntax errors in Mini-Programs fall into two categories, i.e., spelling mistake (4/15), and forgetting to define variables and functions (11/15).

Finding 5: *Syntax errors are common (18.1%) in Mini-Programs.*

No verification of data validity. During the processing of data, especially dealing with return data from requests, developers often assume that the data structure meets their expectations. They may directly access properties or functions on the return data. However, the return data can be `undefined` or `null` due to various reasons,

```
// index.js
1. getList: function() {
2.   var that = this;
3.   util.req (url, page, function(data) {
4. +   if(data.list) {
5.       var list = data.list
6.       list.foreach (function(item)) {
7.           var li = {...}
8.       }
9. +   }
// util.js
10. function req (url, data, cb) {
11.   wx.request({
12.     url: rootDocment + url,
13.     success: function(res) {
14.       return typeof cb== 'function' && cb(res.data)
15.     },
16.     fail: function() {
17.       return typeof cb== 'function' && cb(false)
18.     }
19.   …
```

false.list.foreach

Figure 5: WeBug pinche#20 [50]. In this WeBug, the developer assumes that all return data from req() have the property *list*.

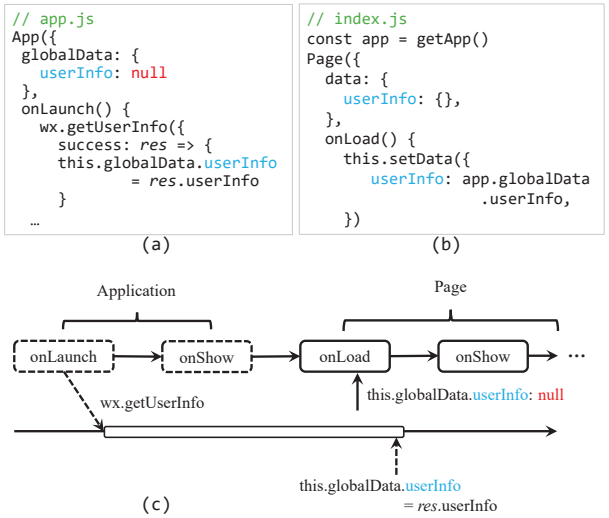


Figure 6: WeBug forumQAs [48]. During the initialization of the Mini-Program, the unexpected data processing orders result in the user information being set to *null*.

e.g., network failures and server crashes. Thus, they need to verify the validity of the data before use. Otherwise, a TypeError [72] due to access property of `undefined` / `null` object will occur. In total, 16 (19.3%) WeBugs are caused by no verification of data validity.

Take pinche#20 [50] in Figure 5 as an example. When a user clicks "mine" at the bottom of the index page, the function `getList` (Line 1) is triggered, and an error message "TypeError: Can not read property 'foreach' of undefined" appears. Function `getList` calls the function `req` (Line 3) to send a request. If the request fails, it triggers the `fail` callback function (Line 17), and returns a boolean data `false`. Since function `getList` does not check whether it is a valid value (e.g., Line 4), and operations on a boolean data `false` can cause a TypeError.

> **Finding 6**: *Developers often assume that return data from requests have expected properties, and forget to verify their validity, causing WeBugs.*

Asynchronous errors. As discussed in Section 2, the Mini-Program framework provides many asynchronous APIs, and Mini-Programs have complicated lifecycle management, which is event-driven. The asynchronous and event-driven mechanism introduces the risk that the data are processed and accessed in an unexpected order, thus causing asynchronous errors. Three WeBugs belong to this case.

Figure 6 shows an asynchronous error caused by lifecycle events in Mini-Programs [48]. Figure 6a and Figure 6b shows the application and page code, respectively. Figure 6c shows the process how this bug happens. When the Mini-Program starts, the application-level lifecycle events (`app.js`) are first triggered, e.g., onLaunch and onShow. Then the page-level lifecycle events (`index.js`) are triggered. The global variable `userinfo` in `app.js` is `null` by default. The `onLaunch()` function in `app.js` asynchronously sends a request to get user information, which is executed in parallel with the code in `onLoad` of `index.js`. If `onLoad` of `index.js` is executed before the request returns in `app.js`, `userInfo` in `index.js` will be set as `null`. This is unexpected.

> **Finding 7**: *Asynchronous mechanism and lifecycle management in Mini-Programs can introduce intricate concurrency errors.*

Logic errors. The logic implemented by developers may be incomplete or incorrect, resulting in some functions or displays that do not satisfy the requirements. 21 (25.3%) WeBugs fall into logic errors. For these WeBugs, most of them (19/21) are caused by incomplete logic. For example, in wxParse#365 [37], the parser cannot parse strings longer than 255 characters. In wechat-app-mall#255 [41], the balances less than 0.1 cannot be rounded.

4.2 Fix Strategy

We find that the fixes of WeBugs are highly related to their root causes. We summarize four common fix patterns for 44.6% of the studied WeBugs. The rest WeBugs are fixed case by case.

Layout adaptation to anomalous screens. Three WeBugs related to inconsistent rendering on specific devices can be fixed by a general approach. Developers can use API `wx.getSystemInfo` to get the `safeArea` of a user's device. With the `safeArea`, they can set up the layout as normal. However, developers may fix these bugs case by case. For example, developers set 34px at the screen bottom if they identify the device type is iPhoneX in WeBug colorUI#50 [40].

Fixing syntax errors. Four WeBugs, caused by spelling mistakes, can be directly fixed with a few grammar checkers, e.g., JSLint [1], and ESLint [4]. The rest 11 WeBugs caused by forgetting to define variables and functions can be detected by these grammar checkers. Base on detecting results, developers can easily fix these bugs.

Adding undefined / null check before use. All 16 WeBugs caused by no verification of data validity are fixed by adding the undefined / null check and exception handling before use. The fix code in Figure 5 is a typical example, in which the developer checks the validity of `data.list` before use.

Synchronization. Incorrect timing of asynchronous APIs and lifecycle events can lead to wrong shared data accessing orders. To fix asynchronous errors, developers can introduce ad-hoc synchronization for shared data. For example, in Figure 6, developers need to check whether `getUserInfo` has returned and `userInfo` has been set in `onLoad` of `index.js`. Three WeBugs caused by asynchronous mechanism are fixed under this approach.

Case-by-case fixes. The remaining bugs have diverse characteristics and require specific fixes. For bugs caused by using deprecated APIs, they have to be replaced by the new versions. There are some API changes detection work [91, 93] for other libraries and frameworks, which can be applied to WeChat Mini-Programs. The API misuse bugs have to be fixed according to the context and API specification. The fixes of bugs caused by logic error depends on the logic itself.

Finding 8: *44.6% WeBugs have common fixing patterns, such as considering device model before using some platform-dependent APIs and adding checks before using request data.*

4.3 Bug Impact

In our study, we first try to reproduce our studied WeBugs, and observe their impacts. If we cannot reproduce a WeBug, we infer its impact from its issue report and related discussion. Finally, we clearly obtain the impacts of 65 WeBugs. For these 65 WeBugs, their impacts are described as follows.

Application crash. Mini-Program bugs hardly cause severe consequences. In our study, we only observe one bug that can crash its application.

Abnormal display. 30 WeBugs lead to abnormal display, e.g., the unexpected style of the view components in Figure 4, some inconsistent rendering among specific devices [40], and errors in style settings [19].

Functional error. In 26 WeBugs, the applications do not meet their requirements. The applications provide the incorrect or flawed functions. We also find that four WeBugs are sporadic. If the user re-enters or refreshes the page, these bugs may disappear. These bugs are usually caused by missing exception handling of network failures and incorrect asynchronous operations.

Silent error. Eight WeBugs neither affect the display of related pages nor cause functional errors. They only report related errors in the console.

Finding 9: *WeBugs can cause various consequences, e.g., applications crashes, abnormal displays, functional errors and salient errors.*

5 LESSONS LEARNED

WeBugs can lead to abnormal display, functional errors and even application crashes (Finding 9). Thus, resolving WeBugs is of great significance for the reliability of WeChat Mini-Programs. In this section, we discuss implications to existing practice and opportunities for research in avoiding, detecting and testing WeBugs in WeChat Mini-Programs.

5.1 Avoiding WeBugs

Syntax errors are common (18.1%) in practice (Finding 5). Developers can apply some tools to reduce the occurrence of syntax errors. For example, developers can use syntax checkers in the development process, e.g., ESLint [4] and JSLint [1], thus avoiding some syntax errors.

Although Mini-Programs are designed to run on multiple platforms, many (14.2%) WeBugs are still caused by the discrepancies among different platforms (Finding 1, 2). During our study, we also observe that the official document does not clearly state related APIs' restrictions. To avoid these WeBugs, these restrictions should be well documented in the official document. In the same while, a static checker tool, which can automatically identify related APIs in Mini-Programs and check whether their restrictions are satisfied, could be greatly helpful. From another point, the Mini-Program framework can shield some discrepancies among different platforms, e.g., inconsistent time format results from `Date`. Thus, related WeBugs can be avoided.

5.2 Detecting WeBugs

Existing JavaScript bug detection approaches, e.g., TAJS [78], XSS-SAFE [101] and SYNODE [102], mainly focus on detecting server-side and client-side JavaScript bugs. Our empirical study has reported many Mini-Program specific bugs, e.g., compatibility bugs due to incorrect platform-dependent API usages (Finding 1) and inconsistent rendering across specific devices (Finding 2), asynchronous errors due to Mini-Program lifecycle and asynchronous mechanism (Finding 7) and abnormal rendering results due to improper style priority settings (Finding 4). Existing approaches cannot work on these WeBugs properly. Our findings can be used to design new WeBug detection tools.

Compatibility bug detection. Many (14.2%) WeBugs are caused by the discrepancies among different platforms (Finding 1, 2). This is mainly caused by incompatibility APIs provided by the Mini-Platform framework. By summarizing incompatible APIs in the Mini-Program framework, we can develop a static analysis tool to automatically identify these WeBugs.

Return data guided bug detection. Developers often assume that they can obtain expected data from external requests or asynchronous tasks, and ignore checking the data validity (Finding 6). However, the return data may be wrong due to various reasons, e.g., network failures and asynchronous task failures. Thus, essential judgments on the return data are required, e.g., whether the return data can be used without triggering any errors.

Non-deterministic bug detection. The Mini-Program framework provides many asynchronous APIs, and Mini-Programs adopt event-driven lifecycle management. The asynchronous and event-driven mechanism introduces WeBugs (Finding 7). Different from non-deterministic bugs in client-side JavaScript applications and server-side JavaScript applications, the lifecycle management in

Table 4: Four Kinds of Platform-Dependent APIs

Android only	iOS only	Inconsistency on Android and iOS	Any OS
wx.makeBluetoothPair, wx.qy.startNFCReader, wx.qy.NFCReader, canvas.toDataURL, wx.onShareTimeline	BLEPeripheralServer.onCharateristicSubscribed, VideoContext.showStatusBar, BLEPeripheralServer.onCharasteristicUnsubscribed, wx.setBackgroundColor -a arg.backgroundColorTop -a arg.backgroundColorBottom	wx.onCompassChange - r type: Number (iOS) - r type: String (Android), wx.connectWifi - v WeChat version >11 (iOS) - v no limit (Android), Date - af split with - (iOS) - af split with / (Android)	console.dir *

* console.dir makes some devices crash [32] and it should not be used in Mini-Programs.
-a The arguments of the function | **-r** The return data of the function | **-v** The version of the WeChat | **-af** The data format of the argument.d

Mini-Programs makes non-deterministic bug detection more challenging. To automatically detect non-deterministic bugs in Mini-Programs, we need to build a clear model about the lifecycle management and asynchronous APIs in Mini-Programs.

5.3 Testing Mini-Programs

Testing is critical in exposing bugs before software release. Many testing techniques have been proposed for exposing Android and JavaScript applications [61, 70, 77, 79, 108**?**], but no technique is designed for WeChat Mini-Programs.

WeChat Mini-Programs usually have complicated interfaces and lifecycle management. It is critical to design a tool to automatically expose user interactions with Mini-Programs, thus examining their functionality. Further, Mini-Programs usually depend on various asynchronous tasks and external environments, and may encounter many adversarial conditions, e.g., network failures. Improperly handling these conditions can cause WeBugs (Finding 6). Researchers and developers can intentionally inject adversarial conditions, and validate whether Mini-Programs can behave correctly [59, 73, 89] under adversarial conditions.

6 DETECTING WEBUGS

In this section, we summarize three bug patterns from our empirical study on WeBugs, and further propose a static analysis tool *WeDetector* to detect WeBugs.

6.1 Bug Patterns

Our empirical study on WeBugs has revealed several potential patterns to detect WeBugs. Among them, we summarize three common bug patterns, i.e., incorrect invocations platform-dependent APIs, incomplete layout adaptation to anomalous screens, and improper handling of return data in invoking asynchronous APIs.

Pattern 1: Incorrect invocations of platform-dependent APIs. Some APIs in the Mini-Program framework are OS-specific (Finding 1). Invoking these platform-dependent APIs without considering different OS can introduce WeBugs. Therefore, developers should determine target operating system before using these APIs. We summarize four kinds of platform-dependent APIs in Table 4, which are collected from our empirical study and official Mini-Program development document [8]. The items beginning with - show some additional considerations. For example, "arg.backgroundColorTop" in column "iOS only" means that the argument can only be used on iOS devices when invoking function `wx.setBackgroundColor`. The column "Inconsistency on Android and iOS" includes some APIs that behave differently on different systems. For example, the *Date* format on Android is split with "/" while that on iOS is split with "-".

Pattern 2: Incomplete layout adaptation to anomalous screens. As discussed in Section 4.2, developers can use the API `wx.getSystemInfo` to get the `safeArea` of the phone model and avoid layout issues. If a Mini-Program does not apply this to anomalous screens, WeBugs related to abnormal rendering can occur.

Pattern 3: Improper handling of return data in invoking asynchronous APIs. In the WeChat Mini-Program framework, most APIs are used in an asynchronous way. The *success* and *fail* callbacks are defined to handle the return data for successful or failed invocations. However, the type or data structure for the return data may vary. If developers are unaware of their differences, WeBugs can occur. Figure 5 shows such an example.

6.2 Detection Approach

Detection of pattern 1. For WeBugs caused by incorrect invocations of platform-dependent APIs, we need to examine the usage of specific APIs that are listed in Table 4. In order to identify whether developers consider the system information, WeDetector checks the usage of `wx.getSystemInfo` that can obtain the target operating system information. If developers use the APIs on the list in Table 4 but do not consider the corresponding OS restrictions, we report a compatibility WeBug.

Detection of pattern 2. We examine whether developers have set up the layout based on `safeArea` or `windowHeight` and `windowWidth` through `wx.getSystemInfo` API. The `windowHeight` and `windowWidth` can also obtain the available area of the devices. Therefore, WeDetector checks whether a Mini-Program uses these properties. If not, we report an abnormal rendering WeBug.

Detection of pattern 3. To detect WeBugs in bug pattern 3, we check whether the return data from an asynchronous function (e.g, `wx.request`) is legally used, i.e., there are no type errors.

The main idea is as follows. First, we use Esprima [24] to parse all JavaScript files into abstract syntax trees (AST), and then identify all asynchronous function (e.g, `wx.request`) invocations in a Mini-Program. Second, for each asynchronous function invocation, we use AST to identify its two callbacks, `sucCallback` for its success return, and `failCallback` for its fail return. Third, for callback `sucCallback` and `failCallback`, we extract their return data from the asynchronous function, and further analyze their usage based on TAJS [78]. Fourth, for the return data usages, we check whether they can trigger TypeErrors. If yes, we report a WeBug.

Take WeBug in Figure 5 as an example. We first parse `index.js` and `util.js` into abstract syntax trees. By traversing ASTs, we obtain all function definitions (e.g., `getList`) and function calls (e.g., `util.req`). We then examine all function definitions. If we get an function invocation (e.g., `util.req`), we continue to analyze this invoked function. If a function (e.g., `util.req`) invokes `wx.request`, we further obtain its two callbacks, e.g., success callback (Line 13-15) and failed callback (Line 16-18). Further, we can obtain that the return data `res.data` is passed into success callback (Line 14) and a constant `false` is passed into failed callback (Line 17).

In order to determine whether the return data is used legally, we adopt TAJS [78][1] to carry out a data flow analysis on the return data `data` (Line 3 in Figure 5). The result of TAJS is an intermediate representation, e.g., vardecl['list'], read-variable ['data', v0, v1], read-property [v0, 'list', v2], write-variable [v2, 'list'] for Line 5. Based on these intermediate representations, we can build the usage scenarios of the return data `data`. We find that `data.list.foreach` is the usage sequence on `data` (Line 5-6). If the failed callback is triggered, the return data will be `false` (Line 17), and the execution of `false.list.foreach` will throw an error message "TypeError: Can not read property 'foreach' of undefined". It is because that the first access returns a `undefined` object (`false.list` -> `undefined`) and the second access on `undefined` triggers a TypeError. Note that, if developers has already used if-condition to guard related accesses, we do not report a WeBug. For example, in the fixed version in Figure 5), "if (data.list)" in Line 4 is used to guard Line 5-6. So we do not report bugs for the fixed version.

6.3 Evaluation

In this section, we evaluate the effectiveness of WeDetector using GitHub open source projects, and answer the following research question: **Can WeDetector detect new WeBugs in real-world Mini-Programs?**

Experimental subjects. Our experimental subjects contain 25 WeChat Mini-Programs: 13 Mini-Programs are collected from the latest versions of the Mini-Programs in Table 2, and 12 Mini-Programs are collected from *awesome-wechat-weapp* [45] by their popularity (ordered by GitHub stars) in different categories. The first 12 rows show these 12 Mini-Programs.

Results. We run WeDetector on these 25 Mini-Programs, and detect 12 WeBugs, and confirm that 11 of them are true. Table 5 shows the detailed detection results. Note that, for the latest versions of 13 Mini-Programs in our empirical study (Table 2), we only detect one new WeBugs in pinche [17]. For Mini-Programs that we have not detected new WeBugs in Table 2, we do not list them in Table 5. We further reported all the 11 true WeBugs to their corresponding developers, and attached patches to help developers fix these WeBugs. For now, 7 of them have been confirmed by developers. The last column "Total (Conf.)" in Table 5 shows that details about reported and confirmed bugs.

[1]TAJS does not support new features provided by ECMAScript6. Therefore we use babel [46] to convert Mini-Projects into ECMAScript5.

Table 5: Detection Results on 25 Mini-Programs.

Project	Category	Detection Results				Total
		P1	P2	P3	FP	(Conf.)
wxchat-mail [35]	Email	0	1	0	0	1 (1)
HITMers [25]	Attendance	0	1	0	0	1 (1)
super9 [42]	Video	0	1	0	0	1 (1)
leantodo-weapp [16]	Notebook	0	1	0	0	1 (1)
taro-music [28]	Music	0	1	0	0	1 (1)
weapp-ssha [34]	News	2	0	0	0	2 (2)
weapp-mark [11]	Social	0	1	0	1	0 (0)
nideshop [31]	Shopping	2	0	0	0	2 (0)
Gitter [10]	Git	0	1	0	0	1 (0)
ShellBox [51]	Campus	0	0	0	0	0 (0)
douban [13]	Map	0	0	0	0	0 (0)
weapp-artand [55]	Art	0	0	0	0	0 (0)
pinche [17] *	Travelling	0	0	1	0	1 (0)
Total		**4**	**7**	**1**	**1**	**11 (7)**

* For 13 Mini-Programs in Table 2, we detect one new bug in pinche [17].

The column "Detection Results" in Table 5 shows different patterns of bugs found in these 25 Mini-Programs. We can observe that we can detect new WeBugs for all the three patterns. For bug pattern 1, two Mini-Programs use platform-dependent APIs without compatibility adaptation, and WeDetector reports four WeBugs. For bug pattern 2, among 25 Mini-Programs, 6 of them do not adapt the layout of views across different devices with anomalous screens. For bug pattern 3, WeDetector detects one WeBug in project pinche [17]. In this WeBug, when a `wx.request` fails, the program tries to invoke function on an undefined variable, which can cause a TypeError.

False positive. For the detected 12 WeBugs, we report one false positive, which belongs to bug pattern 2. In this false positive, the developer processes the adaptation to anomalous screens by using an uncommon approach, i.e., style setting for all possible devices. Thus, the Mini-Program does not handle anomalous screens using the common approach discussed in bug pattern 2. This results in the false positive.

Based on the above experiment and results, we draw the following conclusion: *WeDetector can effectively detect WeBugs in real-world WeChat Mini-Programs.*

6.4 Discussion

Comparison with existing bug detection approaches. Existing linter tools [1, 4, 72] can be used to detect incorrect code based on specific rules. However, they cannot analyze the bug patterns in Section 6.1. For the 11 true WeBugs that WeDetector detected, none of them can be detected by these linter tools. Some API usage detection approaches [92, 98, 106] can detect API usage, e.g., cryptographic API usage, APIs that perform additional actions. However,

these tools mainly focus on specific domain, e.g., trusted APIs in JavaScript. Without API specifications for WeBugs, these tools cannot be used to detect WeBugs. The most similar work is WeJalangi [83]. WeJalangi aims to detect NullPointException. It cannot be directly used to detect bug patterns in Section 6.1. In addition to this, WeJalangi is a dynamic analysis tool, and its effectiveness heavily depends on the provided test cases. It is challenging to construct test cases that trigger these WeBugs. However, WeDetector is a static analysis tool, which directly analyzes the code without running WeChat Mini-Programs.

Automated platform-dependent API extraction. In this paper, we manually investigate our collected WeBugs and extract platform-dependent APIs for bug detection. This is a labor-intensive process. It would be interesting to automatically extract these APIs and their API context from highly popular Mini-Programs. As discussed earlier, the most common pattern of fixing these WeBugs is to check device information before invoking certain APIs. Therefore, it is possible for us to mine the usage rules of these APIs. We will explore this in the future work.

7 RELATED WORK

In this section, we discuss related works that are close to our work.

WeChat Mini-Program studies. The WeChat Mini-Program framework is proposed in 2016, and has been widely used in practice. However, only few research works have been done for WeChat Mini-Programs. Hao et al. [76] analyze the development of WeChat Mini-Program and regard it as the next generation of mobile Internet platform. Some works investigate Mini-Programs from usage [63], acceptance [85]. Liu et al. [83] modify Jalangi [99] and use it for dynamic analysis on WeChat Mini-Programs. In our work, we conduct the first empirical study on WeChat Mini-Program bugs, and obtain a better understanding of these bugs. Based on our findings, we also propose a static analysis tool, WeDetector, and detect more bugs in the Mini-Programs. Our study should be able to promote new research in improving the quality of Mini-Programs.

Empirical bug studies. Empirical studies have played an important role in understanding a field, especially for the aspect of software reliability. These studies can often help to characterize problems and provide guidance for following research. Lu et al. [84] conduct the first empirical study on real world concurrency bugs. Their work provides precious information to help improve software reliability, such as concurrency bug detection [66, 107], bug fixing [80], crash recovery [71], etc. Ocariza et al. [94] conduct an empirical study on the client-side JavaScript bugs and find that bugs are usually DOM-related. This promotes some research on DOM [97, 104]. Wang et al. [103] conduct an empirical study on concurrency bugs in Node.js applications. This study reveals that most concurrency bugs in Node.js applications contend against shared memory and external resource. These findings enlighten subsequent works in Node.js bug detection as well. However, no empirical studies are performed on Mini-Program bugs.

Platform fragmentation studies. Different vendors produce a variety of Android phones and make the Android ecosystem suffer from fragmentation problems [3]. Han et at. [75] carry out a topic analysis of vendor-specific bugs to understand the Android fragmentation. They reveal that hardware-based fragmentation in Android is evident. Wei et at. [106] first characterize and detect **c**ompatibility issues **i**nduced by Android **f**ragmentation (FIC issues). They observe that device-specific FIC issues are caused by problematic driver implementation, OS customization, and peculiar hardware composition and the non-device-specific FIC issues are caused by API evolution and original Android system bugs. The high-frequency fragmentation bugs on Android devices are hardware-related, and they do not have much in-depth analysis for some non-functional requirements (such as inconsistent rendering).

Bug detection in JavaScript applications. The approaches used in bug detection are decided by the target systems and bugs. There are various methods for bug detection, e.g., static analysis, dynamic analysis or combination of both. Guarnieri et al. [74] conduct one of the first pointer analyses for JavaScript. They first manage to model some features on JavaScript. TAJS [78] is a type analyzer for JavaScript applications. Our work utilizes the data flow analysis provided by TAJS. There are a group of work to detect concurrency issues in Node.js applications, e.g, NodeAV [61], NRace [62], NodeRacer [69], Node.fz [65], and Madsen et al. [87]. There are also some client-side JavaScript analysis work, e.g., RClassify [109], AutoFLox [96], and SymJS [82]. However, these works cannot detect some kinds of WeBugs in Mini-Programs, e.g., compatibility issues.

8 CONCLUSION

Millions of WeChat Mini-Programs are running on the WeChat social platform, and more than 400 million WeChat users are using Mini-Programs every day. However, WeChat Mini-Programs still suffer from various quality issues related to execution environment, lifecycle management, asynchronous mechanism, etc. A comprehensive understanding of these issues is required to improve WeChat Mini-Programs.

In this paper, we propose the first empirical study on the bugs in Mini-Programs. We collect 83 bugs from real-world Mini-Programs, and perform an in-depth empirical study on Mini-Program bugs from root causes, fix strategies and impacts. We further propose WeDetector to detect WeBugs of three bug patterns. The evaluation on real-world Mini-Programs indicates that WeDetector is effective. We hope our study can inspire more researchers and practitioners to combat Mini-Program bugs.

ACKNOWLEDGEMENTS

This work was partially supported by National Natural Science Foundation of China (61732019, U20A6003, 62072444, 61802378), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences (2018142).

REFERENCES

[1] 2002. *JSLint: The JavaScript Code Quality Tool.* Retrieved Apr 5, 2021 from http://www.jslint.com/
[2] 2008. *Android Document: Understand the Activity Lifecycle.* Retrieved Apr 43, 2021 from https://developer.android.com/guide/components/activities/activity-lifecycle
[3] 2012. *An Analysis Of Android Fragmentation.* Retrieved Aug 10, 2021 from http://www.tech-thoughts.net/2012/03/analysis-of-android-fragmentation.html#.YRoA64gzZPY
[4] 2013. *ESLint: Find and fix problems in your JavaScript code.* Retrieved Apr 5, 2021 from https://eslint.org/

[5] 2014. *Getting Started: WebView-based Applications for Web Developers.* Retrieved Apr 21, 2021 from https://developer.chrome.com/docs/multidevice/webview/gettingstarted/
[6] 2015. *ECMAScript 2015 Language Specification.* Retrieved Apr 5, 2021 from https://262.ecma-international.org/6.0/
[7] 2015. *Kernel Design.* Retrieved Apr 21, 2021 from https://www.chromium.org/chromium-os/chromiumos-design-docs/chromium-os-kernel
[8] 2015. *Mini-Program development document.* Retrieved Apr 10, 2021 from https://developers.weixin.qq.com/miniprogram/en/dev/api/
[9] 2015. *NW.js.* Retrieved Apr 21, 2021 from https://nwjs.io/
[10] 2016. *Gitter.* Retrieved Apr 23, 2021 from https://github.com/kokohuang/Gitter
[11] 2016. *weapp-mark.* Retrieved Apr 23, 2021 from https://github.com/Honye/weapp-mark
[12] 2016. *wechat-app-demo.* Retrieved Apr 23, 2021 from https://github.com/xwartz/wechat-app-demo
[13] 2016. *wechat-webapp-douban-location.* Retrieved Apr 23, 2021 from https://github.com/bruintong/wechat-webapp-douban-location
[14] 2016. *wxParse.* Retrieved Apr 23, 2021 from https://github.com/icindy/wxParse
[15] 2017. *The framework of WeChat Mini-Programs.* Retrieved Apr 18, 2021 from https://developers.weixin.qq.com/miniprogram/en/dev/framework/MINA.html
[16] 2017. *leantodo-weapp.* Retrieved Apr 23, 2021 from https://github.com/leancloud/leantodo-weapp
[17] 2017. *pinche-xcx.* Retrieved Apr 23, 2021 from https://github.com/vincenth520/pinche_xcx
[18] 2017. *Shape of i-button our of wrok.* Retrieved Apr 23, 2021 from https://github.com/TalkingData/iview-weapp/issues/398
[19] 2017. *Syntax Errors lead to abnormal display.* Retrieved Apr 23, 2021 from https://github.com/EastWorld/wechat-app-mall/issues/126
[20] 2017. *Wechat-app-mall.* Retrieved Apr 23, 2021 from https://github.com/EastWorld/wechat-app-mall
[21] 2017. *winxin-watch-life.* Retrieved Apr 23, 2021 from https://github.com/iamxjb/winxin-app-watch-life.net
[22] 2017. *wx-calendar.* Retrieved Apr 23, 2021 from https://github.com/treadpit/wx_calendar
[23] 2018. *ColorUI.* Retrieved Apr 23, 2021 from https://github.com/weilanwl/ColorUI
[24] 2018. *Esprima.* Retrieved Apr 23, 2021 from https://www.npmjs.com/package/esprima
[25] 2018. *HITMers.* Retrieved Apr 23, 2021 from https://github.com/upupming/HITMers
[26] 2018. *iview-weapp.* Retrieved Apr 23, 2021 from https://github.com/TalkingData/iview-weappI
[27] 2018. *scuplus-wechat.* Retrieved Apr 23, 2021 from https://github.com/mohuishou/scuplus-wechat
[28] 2018. *taro-music.* Retrieved Apr 23, 2021 from https://github.com/lsqy/taro-music
[29] 2018. *Weapp trending.* Retrieved Apr 23, 2021 from https://github.com/jae-jae/weapp-github-trending
[30] 2018. *WeHalo.* Retrieved Apr 23, 2021 from https://github.com/aquanlerou/WeHalo
[31] 2019. *nideshop-mini-program.* Retrieved Apr 23, 2021 from https://github.com/tumobi/nideshop-mini-program
[32] 2019. *Problems in using console.dir.* Retrieved Apr 20, 2021 from https://github.com/icindy/wxParse/issues/250
[33] 2019. *threejs-example.* Retrieved Apr 23, 2021 from https://github.com/yannliao/threejs-example-for-miniprogram
[34] 2019. *weapp-ssha.* Retrieved Apr 23, 2021 from https://github.com/yaoshanliang/weapp-ssha
[35] 2019. *wxchat-mail.* Retrieved Apr 23, 2021 from https://github.com/wk989898/wxchat-mail
[36] 2020. *A bug caused by API misunderstanding.* Retrieved Apr 12, 2021 from https://github.com/DoFind/CardOnePerson/issues/2
[37] 2020. *Can not handle too long attribute.* Retrieved Apr 5, 2021 from https://github.com/icindy/wxParse/issues/356
[38] 2020. *CardOnePerson.* Retrieved Apr 12, 2021 from https://github.com/DoFind/CardOnePerson
[39] 2020. *Inappropriate parameters.* Retrieved Jan 5, 2021 from https://github.com/EastWorld/wechat-app-mall/issues/203
[40] 2020. *iPhoneXvirtual.* Retrieved Feb 20, 2021 from https://github.com/weilanwl/ColorUI/issues/50
[41] 2020. *Money is not rounded.* Retrieved Apr 5, 2021 from https://github.com/EastWorld/wechat-app-mall/issues/255
[42] 2020. *super9.* Retrieved Apr 23, 2021 from https://github.com/terryso/super9
[43] 2020. *WeChat revenue and usage statistics in 2020.* Retrieved Apr 12, 2021 from https://www.businessofapps.com/data/wechat-statistics/
[44] 2021. *API wx.previewMedia fails.* Retrieved Apr 5, 2021 from https://developers.weixin.qq.com/community/develop/doc/000aec872908d80896cb6f0335b400
[45] 2021. *awesome-wechat-weapp.* Retrieved Feb 19, 2021 from https://github.com/justjavac/awesome-wechat-weapp
[46] 2021. *Babel documents.* Retrieved Apr 10, 2021 from https://babeljs.io/docs/en/
[47] 2021. *Component textarea fails.* Retrieved Apr 5, 2021 from https://developers.weixin.qq.com/community/develop/doc/00060e8c5340e0742acbeede951000
[48] 2021. *Error in render layers.* Retrieved Apr 5, 2021 from https://developers.weixin.qq.com/community/develop/doc/00046460eb4bf81380cbfb9ab56800
[49] 2021. *JavaScriptCore Tutorial for iOS.* Retrieved Apr 21, 2021 from https://www.raywenderlich.com/1227-javascriptcore-tutorial-for-ios-getting-started
[50] 2021. *No verification of data validity.* Retrieved Apr 21, 2021 from https://github.com/vincenth520/pinche_xcx/issues/20
[51] 2021. *ShellBox.* Retrieved Apr 23, 2021 from https://github.com/Airmole/ShellBox
[52] 2021. *System-specific API usage.* Retrieved Feb 21, 2021 from https://github.com/EastWorld/wechat-app-mall/issues/237
[53] 2021. *Tencent X5 Webview API.* Retrieved Apr 21, 2021 from https://x5.tencent.com/docs/tbsapi/reference/com/tencent/smtt/sdk/WebView.html
[54] 2021. *V8.* Retrieved Apr 21, 2021 from https://v8.dev/
[55] 2021. *weapp-artand.* Retrieved Apr 23, 2021 from https://github.com/SuperKieran/weapp-artand
[56] 2021. *WeChat developer forum.* Retrieved Apr 17, 2021 from https://developers.weixin.qq.com/community/develop/question
[57] 2021. *WeChat Mini-Program daily active users.* Retrieved Apr 10, 2021 from https://www.chinaz.com/news/1218295.shtml
[58] 2021. *WKWebView.* Retrieved Apr 21, 2021 from https://developer.apple.com/documentation/webkit/wkwebview
[59] Cyrille Artho, Armin Biere, and Shinichi Honiden. 2006. Enforcer – Efficient Failure Injection. In *Proceedings of International Conference on Formal Methods (FM)*. 412–427.
[60] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not All Bugs are the Same: Understanding, Characterizing, and Classifying Bug Types. *Journal of Systems and Software* 152 (2019), 165–181.
[61] Xiaoning Chang, Wensheng Dou, Yu Gao, Jie Wang, Jun Wei, and Tao Huang. 2019. Detecting Atomicity Violations for Event-Driven Node.js Applications. In *Proceedings of International Conference on Software Engineering (ICSE)*. 631–642.
[62] Xiaoning Chang, Wensheng Dou, Jun Wei, Tao Huang, Jinhui Xie, Yuetang Deng, Jianbo Yang, and Jiaheng Yang. 2021. Race Detection for Event-Driven Node.js Applications. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. 480–491.
[63] Ao Cheng, Gang Ren, Taeho Hong, Kichan Nam, and Chulmo Koo. 2019. An Exploratory Analysis of Travel-Related WeChat Mini Program Usage: Affordance Theory Perspective. In *Proceedings of Information and Communication Technologies*. 333–343.
[64] Wohlin Claes, Per Runeson, Martin Höst, Ohlsson Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Science & Business Media.
[65] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 145–160.
[66] Dongdong Deng, Wei Zhang, and Shan Lu. 2013. Efficient Concurrency Bug Detection across Inputs. *Acm Sigplan Notices* 48, 10 (2013), 785–802.
[67] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2187–2200.
[68] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security. In *Proceedings of USENIX Conference on Security (Security)*. 1–21.
[69] André Takeshi Endo and Anders Møller. 2020. NodeRacer: Event Race Detection for Node.js Applications. In *Proceedings of International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 120–130.
[70] Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. 2018. Jast: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 303–325.
[71] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 539–550.
[72] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.
[73] Francisco Gortazár, Micael Gallego, Boni García, Giuseppe Antonio Carella, Michael Pauls, and Ilie-Daniel Gheorghe-Pop. 2017. Elastest — An Open Source Project for Testing Distributed Applications with Failure Injection. In *Proceedings of IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 1–2.

[74] Salvatore Guarnieri and Benjamin Livshits. 2009. Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Proceedings of Conference on USENIX Security Symposium (Security)*. 151–168.

[75] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. 2012. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In *Proceedings of Working Conference on Reverse Engineering (WCRE)*. 83–92.

[76] Lei Hao, Fucheng Wan, Ning Ma, and Yicheng Wang. 2018. Analysis of the Development of WeChat Mini Program. 1087, 6 (2018), 062040.

[77] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and Detecting Callback Compatibility Issues for Android Applications. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 532–542.

[78] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proceedings of International Symposium on Static Analysis (SAS)*. 238–255.

[79] Hao Jiang, Hongli Yang, Shengchao Qin, Zhendong Su, Jian Zhang, and Jun Yan. 2017. Detecting Energy Bugs in Android Apps using Static Analysis. In *Proceedings of International Conference on Formal Engineering Methods*. 192–208.

[80] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated Concurrency-Bug Fixing. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 221–236.

[81] Hammad Khalid, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. 2014. Prioritizing the Devices to Test Your App on: A Case Study of Android Game Apps. In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 610–620.

[82] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 449–459.

[83] Yi Liu, Jinhui Xie, Jianbo Yang, Shiyu Guo, Yuetang Deng, Shuqing Li, Yechang Wu, and Yepang Liu. 2020. Industry Practice of JavaScript Dynamic Analysis on WeChat Mini-Programs. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. 1189–1193.

[84] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 329–339.

[85] Lijun Ma, Lan Wang, and Entao Jiang. 2018. Empirical Study on the WeChat Mini Program Acceptance based on UTA UT Model Take the Pearl River Delta as An Example. In *Proceedings of International Conference on Service Systems and Service Management (ICSSSM)*. 1–6.

[86] Xiaojuan Ma. 2019. App Store Killer? The Storm of WeChat Mini Programs Swept over the Mobile App Ecosystem. *Masters of Media* 15 (2019).

[87] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static Analysis of Event-Driven Node.Js JavaScript Applications. In *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 505–519.

[88] Mehran Mahmoudi and Sarah Nadi. 2018. The Android Update Problem: An Empirical Study. In *Proceedings of International Conference on Mining Software Repositories (MSR)*. 220–230.

[89] Paul D. Marinescu and George Candea. 2011. Efficient Testing of Recovery Code Using Fault Injection. *ACM Transactions on Computer System (TOCS)* 29, 4 (2011), 1–38.

[90] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM)*. 70–79.

[91] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. 1–24.

[92] Duncan Mitchell and Johannes Kinder. 2019. A Formal Model for Checking Cryptographic API Usage in JavaScript. In *Proceedings of European Symposium on Research in Computer Security*. 341–360.

[93] Anders Møller and Martin Toldam Torp. 2019. Model-Based Testing of Breaking Changes in Node.js Libraries. In *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 409–419.

[94] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An Empirical Study of Client-Side JavaScript Bugs. In *Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 55–64.

[95] Frolin S Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2016. A Study of Causes and Consequences of Client-Side JavaScript Bugs. *IEEE Transactions on Software Engineering (TSE)* 43, 2 (2016), 128–144.

[96] Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. 2012. AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 31–40.

[97] Jinkun Pan and Xiaoguang Mao. 2017. Detecting DOM-Sourced Cross-Site Scripting in Browser Extensions. In *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 24–34.

[98] Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. 2020. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In *Proceedings of ACM Internet Measurement Conference (IMC)*. 648–661.

[99] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 488–498.

[100] Yuru Shao, Ruowen Wang, Xun Chen, Ahemd M. Azab, and Z. Morley Mao. 2019. A Lightweight Framework for Fine-Grained Lifecycle Control of Android Applications. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 14–31.

[101] Gupta Shashank and Brij Bhooshan Gupta. 2016. XSS-SAFE: A Server-Side Approach to Detect and Mitigate Cross-Site Scripting (XSS) Attacks in JavaScript Code. *Arabian Journal for Science and Engineering (ARAB)* 41, 3 (2016), 897–920.

[102] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*.

[103] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.js. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. 520–531.

[104] Ran Wang, Guangquan Xu, Xianjiao Zeng, Xiaohong Li, and Zhiyong Feng. 2018. TT-XSS: A Novel Taint Tracking based Dynamic Detection Framework for DOM Cross-Site Scripting. *J. Parallel and Distrib. Comput.* 118 (2018), 100–106.

[105] Xijie Wang, Minyong Shi, and Chunfang Li. 2019. Implementation of Elementary Chinese Language Learning Application in WeChat Mini Programs. In *Proceedings of IEEE International Conference on Big Data Analytics*. 394–398.

[106] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. 226–237.

[107] Zhendong Wu, Kai Lu, Xiaoping Wang, and Xu Zhou. 2015. Collaborative Technique for Concurrency Bug Detection. *International Journal of Parallel Programming* 43, 2 (2015), 260–285.

[108] Zhemin Yang and Min Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *Proceedings of World Congress on Software Engineering (WCSE)*. 101–104.

[109] Lu Zhang and Chao Wang. 2017. RClassify: Classifying Race Conditions in Web Applications via Deterministic Replay. In *Proceedings of International Conference on Software Engineering (ICSE)*. 278–288.

[110] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *Proceedings of International Conference on Software Engineering (ICSE)*. 1159–1170.