TNBDroid: Testing HTTP(S) Network-Related Behaviour of Android Applications

Anonymous Author(s)

ABSTRACT

For most Android apps, while the network plays a critical role in providing the app functionalities, the bugs of network usage account for a large proportion of app crashes. Since many network-related bugs can only be triggered in certain conditions (e.g., when the network speed is slow, the network response takes a long latency), it is difficult for existing general or GUI testing approaches to manifest them. To the best of our knowledge, there are few techniques dedicated to testing the network usage in apps. In this paper, we present an automatic traversal-based testing approach, TNBDroid, to effectively and efficiently explore the HTTP(S) network-related behaviour of apps. We have applied TNBDroid to 35 real-world Android apps, and the experimental results demonstrate that our approach achieves a high coverage (on average 77.37%) of the HTTP(S) network-related behaviour, and totally finds from the 35 apps 57 bugs that are relevant to the HTTP(S) network usage in 683 minutes.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Android app, network-related behaviour, testing

ACM Reference Format:

Anonymous Author(s). 2020. TNBDroid: Testing HTTP(S) Network-Related Behaviour of Android Applications. In *Proceedings of Proceedings of the 42nd International Conference on Software Engineering (ICSE'20).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many Android apps (apps for short) are built to replace the corresponding Web applications running on the desktop in the traditional client(browser)-server model. Therefore, these apps call for network to connect to the server to download and upload resources, and to update and share data to others. Since the network is diverse (e.g., 2G/3G/4G, WiFi, Bluetooth) and could be unstable, apps tend to crash due to network problems and exceptions [16, 21, 32]. Thus, app developers and testers should pay attention to the network invocations and the relevant behaviour of apps.

Although many techniques and tools have been proposed for testing apps, they are mostly general testing approaches, mainly focusing on testing the GUI of apps [10, 31]. When they (e.g., Monkey [12]) are used to test apps, much network-related behaviour cannot be explored effectively or efficiently. On the other hand, there are few approaches dedicated to the testing of network-related behaviour of apps. For example, Huang et al. employ the middleman technique to simulate different response data for detecting the network security threats to apps [16]. However, the middleman technique is forbidden by many apps.

In practice, app developers test network-related behaviour of apps in manual ways or with the help of different hardware and software simulators [16, 21, 32]. Specifically, they usually set different network environments to test the network-related behaviour of apps, and also validate the app when the network switches to another kind. Besides, they may also change the physical properties of the network (e.g., limit the speed of the network, increase the network response latency) to check whether their apps crash in such scenarios. Although this practice can manifest some bugs that are relevant to the network usage, it is resource-intensive and time-consuming. More importantly, this approach cannot test the network-related behaviour systematically because it is difficult to simulate all network environments, and thus some potential network-related bugs may not be found.

A study reports that in the app crash with the network call, about 88% of the network calls are successful, but returned unexpected data (dirty data); and, 10% of successful network calls that return no data[1]. Inspired by these, we propose in this paper a traversal-based testing approach, and an open-source tool TNBDroid (the URL is omitted for double-blind review), to systematically and automatically test the HTTP(S) network-related behaviour of apps. Our approach is based on the techniques of fuzzing, a technical to manifest program bugs by iteratively providing irregular data to the program [11, 16]. Our main idea is to use a software-defined way to instrument an app to define different possible outcomes (e.g., response data) of the HTTP(S) network request, and see how apps respond to the different outcomes (whether apps work well in the presence of different outcomes). The rationale behind lies in that many app crashes are not caused by the network itself or its properties (e.g., speed), but due to the fact that the app code does not handle the network exceptions or the response data in a proper way. As a consequence, we do not need to simulate different kinds of networks or their specific properties; instead we can directly simulate the possible responses of the HTTP(S) network request.

To implement our approach, we first conduct a static analysis of the app to locate the positions of all HTTP(S) network requests (i.e., network API invocations) based on Soot [30][2]. Meanwhile, we also determine the impact scope (in terms of a set of statements, called slice) of each network request. Based on the position of each HTTP(S) network API invocation, we then instrument the original

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE'20, May 23–29, 2020, Seoul, South Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]https://www.apteligent.com/research/network-crashes (accessed Aug, 2018)
[2]https://github.com/Sable/soot

app to add some code at the position to acquire the real response data based on which we can do fuzzing (i.e., define other possible response data) accordingly. Based on Robotium[3], we pre-explore the app to identify the activities that do not trigger network request directly. In the real traversal-based testing, these activities can be skipped to improve the exploration efficiency. Note that all simulated response data is supposed to be handled by the corresponding HTTP(S) network-related behaviour of the app. If the app does not handle the response data properly, a bug can be found.

The contributions of this work are summarized as follows:

(1) We develop a traversal-based testing approach and an open-source tool TNBDroid to systematically and automatically test the HTTP(S) network-related behaviour of apps. Based on a static scan and the following instrumentation, our approach traverses the app to first record the real network response data and then simulate and mutate the other possible response data accordingly with a set of fuzzing rules, which can test the network-related behaviour in an efficient and low-cost way.
(2) We apply TNBDroid to 25 apps from F-droid and 10 commercial apps from Google Play, the results of which demonstrate both the effectiveness and efficiency of our approach: it achieves a high code coverage (on average 81.74% for apps from F-droid, 66.43% for apps from Google Play) of the HTTP(S) network-related behaviour, and totally finds 57 bugs relevant to the network usage in the 35 apps.

The remainder of the paper is organized as follows. Section 2 positions our study in the related work. Section 3 presents our approach to testing network-related behaviour of apps. Section 4 reports on the experimental results on real-world apps. Section 5 discusses the limitations of our current work, and Section 6 concludes the paper.

2 RELATED WORK

Our work is related to both network behaviour analysis and automated testing of Android apps. In this section, we briefly review the existing work on these two aspects.

2.1 Network Behaviour Analysis

Network behaviour analysis. Nayam et al. [27] present a framework for identifying undesirable network behaviour of mobile apps. The proposed framework runs an app to obtain packet-level traces of the app's network behaviour, and then generates a set of attributes to group the app behaviour into three categories, including suspicious, potentially suspicious, and innocuous. Mostafa et al. [26] propose a method for statically summarizing network behaviour from the bytecode of Android apps, which can be used to determine the location of network-related code in the apps and to facilitate the maintenance of such code. Based on the string taint analysis, the method generates a summary of the network request by statically estimating the possible values of the network API parameters. The method is implemented as a static analysis tool NetDroid.

Network behaviour testing. To test the network behaviour of mobile apps, the main method is to simulate various network environments through hardware or software, and then run the app in the simulated network environment to verify whether errors will occur. For example, Caiipa [21] is based on the cloud service technology for testing mobile app over an expandable mobile context space. It simulates various contexts including various network environments at the hardware level. Some existing middle agent software, such as Fiddler[4] and Charles[5], can capture network data and simulate different network issues, such as long network latency and network low bandwidth. App developers often utilize these software to approximately simulate the bad network environment and use the captured network data to determine the success and failure of the network request.

There are also some work which simulates network failures by injecting network-related exceptions into the app. For example, Yang et al. [33] study the impact of weak response on the app, and detect whether the app can have an ANR (Application Not Responding) error by inserting a long delay into the typical time-consuming operation (including network access) in the app. Similarly, JazzDroid is an automated Android gray box testing tool that dynamically injects various environmental interference into the application to detect issues [32]. It contains a network issues injector that interferes with the app's HTTP requests and manipulates their responses to simulate network latency and connectivity.

In contrast to these researches, we do not simulate the network environments or the network issues such as network latency and connectivity. Instead, we directly simulate the response data that could be generated in those scenarios. Therefore, our approach is more easy to implement. Similar to us, Huang et al. [16] propose a network packet-based fuzzing method to test the network-related behaviour of Android apps. Based on Monkey and the middleman technique, this method first obtains the interaction data sent by servers to apps, and then adopts different mutation strategies to mutate the original data, and finally returns the mutated response data to apps. However, the middleman technique can only intercept the network data from the port in the WiFi environment, and thus it is difficult to know which app the data belongs to. The goal is to discover potential security threats of the apps. By contrast, our goal is to test network-related behaviour as comprehensively and efficiently as possible, not just to discover potential threats.

2.2 App Testing

Most existing app testing approaches aim at detecting general (UI) bugs that can be triggered from the app GUI [10, 31]. Their goal is to explore the app behaviour as comprehensively as possible. Generally speaking, there are three exploration strategies used in app testing[10]: *random testing* [12, 23, 28], *model-based testing* [2, 13, 29, 34], and *event sequence-based testing* [3, 24, 25].

Random testing. Monkey [12] provided together with the Android SDK is the most frequently-used testing approach for Android apps [10]. However, Monkey generates many redundant events in the testing. To this end, Dynodroid [23] is proposed to reduce the generation of redundant events. It triggers the next event based on the app reaction of the preceding triggered event. Since it is difficult to trigger certain events from the GUI, EHBDroid [28] is proposed

[3]https://github.com/RobotiumTech/robotium

[4]https://www.telerik.com/fiddler
[5]https://www.charlesproxy.com

to directly trigger the event callbacks (event handlers) based on the app instrumentation. Although EHBDroid bypasses the GUI and thus is more efficient, its random testing may generate false alarms because some bugs it detects do not occur in real executions. Besides these studies, the early Android app testing approaches mostly fall into the category of random testing [1, 2, 4, 15].

Model-based testing. Since redundant events are inevitably generated in random testing, a great deal of work propose to use model-based testing [2, 5, 6, 8, 9, 13, 29, 34]. These approaches rely on a model (e.g., a state machine) of the app GUI to generate test cases. The models are either obtained manually or can be generated through static and dynamic program analysis techniques. For example, GUIRipper [2] dynamically builds such a model based on which a depth-first exploration strategy is employed to test the app. GUIRipper is a black-box approach limited to UI event generation. ORBIT [34] first derives a model based on the app source code, and then utilizes the model to generate event for testing. Stoat [29] is a guided approach to performing stochastic model-based testing on apps. Recently, a new model-based testing tool, called Ape, is presented [13]. Instead of using a static model, Ape dynamically optimizes the model based on the generated information during testing. The effectiveness of these approaches significantly depends on the model. However, it is rather difficult to derive a accurate and complete model of the app.

Event sequence-based testing. Since many well-hidden bugs can only be manifested through specific event sequences. The last category approaches focus on generating such event sequences for app testing [3, 19, 24, 25]. For instance, ACTEve [3] employs the technique of symbolic execution to generate effective event sequences. EvoDroid [24] leverages the evolutionary algorithm to generate specific event sequences. Sapienz [25] uses a search-based algorithm to yield a set of shortest event sequences which can maximize the code coverage.

Recently, some work focuses on testing certain properties of apps, e.g., energy testing [7, 17, 18], service testing [22, 35]. However, little work concentrates on network-related behaviour testing.

3 TNBDROID

Figure 1 shows the workflow of our approach TNBDroid. The input of TNBDroid is an Android app, and the output is a crash (bug) report and a code coverage report for the HTTP(S) network-related behaviour. TNBDroid involves the following four major stages (modules), where the first two steps are based on Soot[6] if the source code is unavailable, and the last two are based on Robotium[7].

(1) **Static analysis**. It locates in the app the positions of all HTTP(S) network API invocations. Moreover, for each API invocation, it utilizes a program slicing tool to determine the slice (a set of statements) that could be impacted by the network API invocation.
(2) **Instrumentation**. This stage instruments the original app to obtain two new apps: the pre-explored app and the tested app. The exploring app is obtained as follows. For each HTTP(S) network API invocation, it instruments a logging statement to indicate whether this invocation is executed at runtime. The testing app is obtained in the following way. For each HTTP(S) network API invocation, it instruments some statements that records the response data for the invocation, and simulates and mutates some other possible response data.
(3) **Pre-exploration**. In this stage, the pre-explored app is employed to identify the activities that do not trigger HTTP(S) network API invocations. Specifically, it dynamically explores the app using a breadth-first traversal (details are in the following) and records the activities that are visited. It also labels each visited activity to indicate whether network API invocations can be triggered directly from the activity, which is helpful for reducing the space of the testing.
(4) **Testing**. Based on the output of the auto-exploration stage, this stage explores the tested app by a breadth-first traversal to trigger all HTTP(S) network API invocations and the following behaviour. It also substitutes the instrumented response data for the real response data to sufficiently test the relevant behaviour of the API invocations. Its output is a crash report and a code coverage report.

[6]https://github.com/Sable/soot
[7]https://github.com/RobotiumTech/robotium

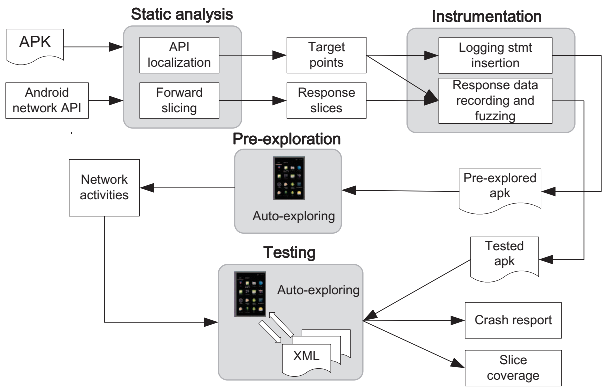

Figure 1: Framework of TNBDroid.

In the following, we elaborate on these four stages, respectively.

3.1 Static Analysis

The static analysis is performed on the source code of the app. If only the APK file is available, the static analysis can be applied to the Jimple code. This stage involves the following two steps.

Network API localization. Android SDK provides a variety of network APIs. App developers can choose appropriate network APIs to make app access to the network. Additionally, certain apps may also use third-party frameworks to connect to the network. To systematically test all the network-related behaviour of an app, we first need to determine which network APIs are used in the app, and where they are used. The network APIs considered in this paper are summarized in Table 1, including the native APIs of Android SDK and the APIs of the three most popular third-party network libraries, where concrete methods of the APIs are not listed. Since the return data type of `java.net.Socket` is not known, we do not consider the APIs of socket communications. In the following, when we mention network requests and network API invocations,

Table 1: Android Network APIs

Category	Class
Android SDK	`java.lang.URL`
	`java.net.URLConnection` `java.net.HttpURLConnection` `javax.net.ssl.HttpsURLConnection`
	`org.apache.http.client.HttpClient`
	`java.net.Socket` (not considered)
Android network libraries	`Retrofit`
	`Volley`
	`OkHttp`

they all refer to HTTP(S) requests. When the source code of the app is not available, we use Soot to convert the app's bytecode into the Jimple code. The Jimple code is more readable than the bytecode and is more suitable for program analysis. We traverse the Jimple code to locate the positions (called target points) where the network APIs are used in the app.

Forward slicing. The behaviour of an app is very complicated, while the network-related behaviour is only a part of it. Since our goal is to manifest the bugs relevant to the network usage, we only focus on the app code that corresponds to the network-related behaviour. To this end, we use an open source tool, called Extractocol [20][8], to extract the program slice s_r (a set of statements) with respect to each network request and the corresponding response. Extractocol is based on Flowdroid [4][9]. Since the bugs of network usage are mainly caused by the network response data, we use the forward slicing technique of Extractocol to obtain all statements that could be affected by the network response data. The network-related behaviour is reflected and determined by the statements of network requests and the statements in the corresponding slices.

3.2 Instrumentation

This stage is to prepare two instrumented apps for the last two stages, respectively. Similar to the stage of static analysis, the instrumentation can be applied to the source code, or the Jimple code with the help of Soot.

Logging statement insertion. As aforesaid, by running the pre-explored app at the third stage, we can determine which activities can directly trigger network requests and which cannot. To obtain the pre-explored app, we need to instrument the original app by adding a log statement at each of the target points identified in the first stage. By running the pre-explored app, once the app initiates a network request, the corresponding log statement will be printed. The printed log is captured in real-time with the help of the Android debugging tool ADB[10]. At the third stage (pre-exploration) of our approach, if during the lifecycle of an activity, there is a log captured by ADB, it indicates that the activity can directly trigger network request and referred to as the network activity.

Response data recording and fuzzing. As aforesaid, by running the tested app at the four stage, we can validate whether the app can properly handle different response data of the network request. To obtain the tested app, we instrument the app to implement the function of automatically acquiring network response data and the follow-up fuzzing. Meanwhile, we also instrument an log statement for each statement in the slice s_r (returned in the first stage) of each network request, in order to calculate the code coverage of our approach. More details are as follows.

Instrumentation point. Since an app may use multiple network APIs, we need to instrument the data acquisition and fuzzing code in all of the app's network API calls to test the network-related behaviour of the app as comprehensively as possible. For native network APIs (Android SDK), the instrumentation is performed at the corresponding target point (strictly speaking, at the position directly follows the target point). Since the `Volley`[11] library encapsulates the Android SDK for the network request, the instrumentation point is determined in the way similar to the native network APIs. For apps that use the libraries `Retrofit`[12] and `OkHttp`[13], since both libraries provide synchronous and asynchronous APIs for network request, and it is somewhat difficult to obtain the response object for the asynchronous request, the target point identified in the first stage cannot be the correct instrumentation point. We choose to instrument inside the library. Since `Retrofit` uses `OkHttp` for the network request, we can only instrument `OkHttp` accordingly. In `OkHttp`, no matter whether synchronous or asynchronous network request is used, the `getResponseWithInterceptorChain()` method is finally invoked, in which we choose the statement that invokes the `proceed()` method as the instrumentation point, because it returns the response object for the network request.

Response data acquisition. Our method is mainly for the network request based on the HTTP(S) protocol. The response of HTTP(S) contains not only the response data, but also the status code and response header[14]. We determine whether the network request is successful according to the status code, and judge whether the response data is empty according to whether "Content-Length" in the response header is zero. From the "Content-Type" field in the response header, we know the type of response data. Based on this information, we can parse the response data correctly and perform the follow-up data fuzzing. For the socket communication, since only the data stream is returned, it is not easy to perform fuzzing. Therefore, we do not consider APIs of socket communication.

Response data fuzzing. Since most crashes relevant to the network usage occur when the network calls are successful, we mainly fuzz the response data in this case (status code 2XX)[15]. When the status code is 1XX (indicating that the server is processing the request) or 4XX/5XX (only contains some error information), we do not perform the fuzzing. We also consider redirect status code 3XX if the response content is not empty. We design the following fuzzing rules in our approach:

- `NULL`: Replace the original data with null to test whether the app can correctly handle the response data as null;
- `EMPTY`: The original data is replaced with an empty data to test whether the app can correctly handle the response data length of 0;

[8]https://github.com/kaist-ina/Extractocol_public
[9]https://github.com/secure-software-engineering/FlowDroid
[10]https://developer.android.com/studio/command-line/adb

[11]https://github.com/google/volley
[12]https://square.github.io/retrofit
[13]https://square.github.io/okhttp
[14]https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[15]https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Table 2: Content-Types and Their Fuzzing Rules

Content-Types	Fuzzing rules
application/json, application/xml, text/xml, text/plain (json type)	NULL, EMPTY, DEL, VALUE_NULL, VALUE_BOUND, VALUE_RAND, CODE404, CODE503
image/gif, image/jpeg, image/png, text/html	NULL, CODE404, CODE503
text/plain (basic types)	NULL, EMPTY, VALUE_BOUND, VALUE_RAND, CODE404, CODE503

- DEL: For the original data of the key-value form, i.e, JSON and XML, some fields are randomly deleted to test whether the app can correctly handle the data format error;
- VALUE_NULL: For the original data of the key-value form, i.e, JSON and XML, the values of some fields are randomly set to null to test whether the app can correctly handle the case where the field value is null;
- VALUE_BOUND: For the original data, if its type is int, float, or long, etc., the data is set to the corresponding boundary values to validate whether the app can correctly handle the boundary value;
- VALUE_RAND: The original data is set to a random value of the same type and the same length to check how the app responds to different data of the same types.

To test the network-related behaviour more comprehensively, we also simulate the failure response for the app request:

- CODE404: Set the response status code to 404 and set the corresponding client error information to validate whether the app can correctly process the status code 404;
- CODE503: Set the response status code to 503, and set the corresponding server error information to validate whether the app can correctly process the status code 503.

Table 2 summarizes the fuzzing rules that can be applied to different Content-Types. All fuzzing rules are applied to the JSON and XML type data. Except DEL and VALUE_NULL, the rest fuzzing rules are applied to the basic types (i.e., int, float, long, string, boolean). For the network response which contains image or HTML data, we only fuzz response data as NULL and request failure, respectively. We do not fuzz the response for the other response data types (e.g., PDF, octet-stream), for they are difficult to parse.

Fuzzing data storage. The fuzzing (simulated) response data of each network request is stored in a local XML file. To ensure that the network request can read the right fuzzing data, we name the XML file of the fuzzing data with the MD5 value of the URL in the request to distinguish the fuzzing data from responses of different requests. If the app triggers the same HTTP(S) request again, our instrumented code will read the new response data directly from the corresponding XML file. If an app does not have the permission to access the external storage, we grant it the permission by inserting the permission into the Android Manifest XML file and turning on the corresponding switch for the app.

3.3 Pre-exploration

If we have a state machine (i.e., activity transition graph) of the app and we know how to trigger each HTTP(S) request from the main activity, the testing can be guided and thus fast. However, neither is obtained for free. As far as we know, it is difficult to obtain both especially when the source code is unavailable. To this end, we adopt a breadth-first traversal to comprehensively test the app behaviour relevant to the HTTP(S) request. To reduce the traversal space for our testing, we first pre-explore the app to identify which activities need exploring and which do not.

The pre-exploration is based on Robotium, an open-source automated testing framework. Robotium itself is a package based on the *Instrumentation* framework provided by Android, which can easily extracts and then performs respective operations (e.g., click, slide, longClick) on the components in the app interface. Based on Robotium, we implement an automated traversal strategy to explore the app. Since the *Instrumentation* framework runs the test program in the same process as the app, we can even monitor the activity lifecycle of the app to record the information (i.e., *Intent*) to start the activity.

Component classification. To cover as many HTTP(S) requests as possible, we need to click all the components on the app interface. Through our experiments, we find that getting all the components of the app's current interface and operating them one by one will cause many of the app's activities not to be traversed. The main reason is that some component clicks make certain components be overwritten or make the current activity's fragment switch. Since the original components are destroyed, they cannot be clicked any more. In response to the above problems, we use a component classification strategy. By classifying the components of the current interface and setting the corresponding priorities, the components with the highest priority are clicked first, and the components of the same priority are clicked sequentially. We mainly divide the window components into four groups from the highest to the lowest priorities: Pop-up components, tab bar components, view group components, and other components.

Figure 2 shows the main activity of the *Wikipedia* app. The components on this interface are divided into four groups according to our classification strategy. The label "1" in Figure 2 corresponds to a pop-up component (menu). By clicking on the menu, another pop-up component (sidebar) will be opened. When one of the components on the sidebar is clicked, the sidebar will automatically be closed, causing the remainder components on the sidebar to become invisible (thus unclickable). Moreover, clicking the component on the sidebar often triggers the activity jump or the change of the interface content. Therefore, if the pop-up component disappears and there are some components on the original pop-up component not traversed, the pop-up component needs to be re-opened. Notably, each time we click a component on the pop-up component, you need to judge whether the current interface content changes. If it changes, you need to re-traverse it once. Pop-up components also include Menu, Dialog, etc. The label "2" in Figure 2 is a tab switching bar. When any tab is clicked, the content of the current interface will change. Therefore, we need to re-acquire all the components of the current interface every time we click on a tab component, and re-traverse it once. The label "3" in Figure 2 is a

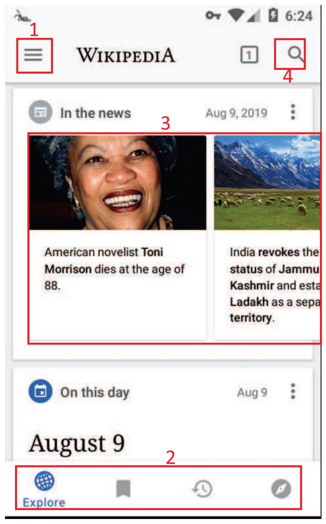
1
2
3
4
Wikipedia
In the news
Aug 9, 2019
American novelist Toni Morrison dies at the age of 88.
India revokes the status of Jammu Kashmir and esta Ladakh as a sepa territory.
On this day
Aug 9
August 9
Explore

Figure 2: The components on the main activity of *Wikipedia* that is classified into four groups (priorities) from "1" to "4".

view group component containing multiple items. We need to traverse all items in it. Components in this category include ListView, GridView, ScrollView and RecyclerView. The label "4" in Figure 2 is a normal component, we can just click it. Components of this kind include Button, TextView, etc.

Traversal strategy. To run Robotium, we need to provide it with the launch activities (including the main activity), which can be obtained from the Android Manifest XML file. We implement a breadth-first traversal strategy to automate the pre-exploration of the app. We start from the main activity, and maintain two activity queues, including *openedActivities*, a queue for the newly-found activities, and *exploredActivities*, a queue for the explored activities. For the main activity or each activity dequeued from *openedActivities*, if it exists in *exploredActivities*, this activity is skipped. Otherwise, all its components are classified and prioritized according to our component classification strategy. When a component is clicked, if the app jumps to a new activity, the new activity related information (activity name, Intent information) is recorded, and the new activity is put into *openedActivities*. Meanwhile, the app returns to the original activity, and the exploration continues. If no activity jump occurs, we determine whether new components are generated (including dialog). If so, the newly generated component is processed first. If no new component is created and all clickable components of the current activity have been clicked, the activity is put into *exploredActivities*. The above procedure is repeated until *openedActivities* becomes empty.

Network activities. During the traversal procedure, we record the activities that can directly trigger HTTP(S) network requests, and we call them network activities. In the real testing stage, if an activity is not a network activity, it will be skipped, which fastens the testing procedure. The method of determining network activities is as follows. In the implementation of the breadth-first traversal, we monitor each activity's lifecycle by registering `ActivityLifecycleCallbacks()`. Hence, if an HTTP(S) request is triggered at runtime, the instrumented log statement (cf. Section 3.2) will be outputted. Therefore, if during the span from an activity's `onResume()` method is called to the activity's `onPause()` is called, there is an instrumented log statement printed, then the activity is a network activity.

3.4 Testing

Finally, we explain the last stage of our approach, that is, the real testing. The testing is performed on another instrumented app, i.e., the tested app.

Traversal-based testing. Similar to the pre-exploration stage, Robotium is employed to start the tested app and to realize a breadth-first traversal of the app for the testing. To comprehensively test the HTTP(S) network requests, we need to traverse the app iteratively to trigger all network requests and simulate their responses comprehensively. More specifically, we first sign and install the tested app (e.g., in a simulator), and then run the breadth-first traversal. If the activity that is opened is the network activity, we traverse each component according to their priorities (cf. Section 3.3). Otherwise, we do nothing and return directly. In the process of traversing the component, when the app's network request is triggered, the testing procedure is summarized in Figure 3: Our instrumented code will automatically obtain the real response data and perform fuzzing. When the same network request is triggered repeatedly, the new fuzzing data can be read from the corresponding XML file to replace the original response data. The automated traversal iterates until either of the following two termination conditions is met:

(1) All fuzzing data have been used as the response data and no new HTTP(S) network requests can be generated.
(2) The app ends normally and the slice coverage reaches a fix point (the slice coverage cannot be increased any more).

During testing, if a new response is captured or new fuzzing data is read, the corresponding log statement (added in the instrumentation stage) will be outputted. Therefore, if there are no corresponding log statements outputted any more, we known all fuzzing data has been covered.

In our empirical analysis, we observe that certain apps always generate new HTTP(S) network requests, which causes the first termination condition not to be satisfied. We look into these apps and those HTTP(S) network requests. We find that the URLs of these new requests are almost the same with the carried parameters different; their response data is the same, as well as the corresponding code parsing the response data. In this case, the second termination condition applies.

Crash and coverage report. In the procedure of testing, we also implement the automatic capturing and recording of app crash information. By implementing the `UncaughtExceptionHandler` interface in our traversal program in Robotium, once the app crashes, the corresponding exception information will be captured. We parse the exception information and store it in a local file, which is the outputted crash report.

When the tested app is started, we start a thread in our program implemented in Robotium to record the executed statements in each slice into a file with the help of ADB. From the file, we can count the number of statements to generate the slice coverage report.

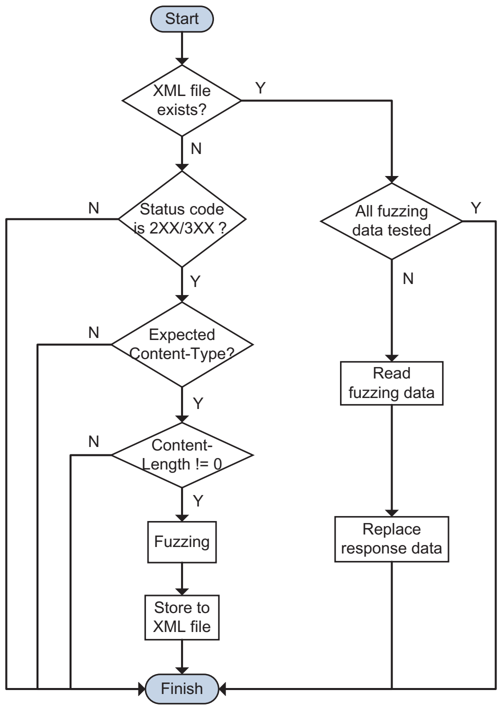

Figure 3: Obtain the real response data of the HTTP(S) network request and use fuzzing response data for the testing.

4 EVALUATION

We conduct an experimental evaluation on real-world apps to answer the following three research questions:

- **RQ1** - *Code coverage*: What is the code (statement) coverage of TNBDroid in exploring the HTTP(S) network-related behaviour?
- **RQ2** - *Fault detection ability*: Can TNBDroid effectively find bugs that are relevant to HTTP(S) network-related behaviour?
- **RQ3** - *Testing efficiency*: What is the reduction rate of activities in testing? And, how efficient is TNBDroid?

Benchmarks. Table 3 summarizes the real-world apps (the app versions are not listed, and we can provide the apps once the paper is accepted) employed in our evaluation, which are selected from F-droid and Google Play. The first 25 apps (from F-droid) are popular benchmarks with a wide variety of functionalities, with 11K lines of code (Jimple), 900 methods, and six activities on average for each app. The other 10 apps (from Google Play) are complex real-world commercial apps on the Android market, with 126K lines of code, 10K methods, and 19 activities on average.

All apps are analyzed with our open-source tool TNBDroid on a computer with an Intel Core i7 3.6GHz CPU and 32 GB of memory, running Windows 10, JDK 1.8, and Android 7.1.1.

4.1 Code Coverage

Coverage criterion. To quantitatively evaluate the effectiveness of TNBDroid, we define a novel coverage criterion (called *slice coverage*) based on the traditional statement coverage. Specifically, we first use the tool Extractocol to obtain the slice S_r (a set of statements) that could be impacted by the response data of an HTTP(S) network request. Errors usually occur to the statements in S_r when the response data is dirty. Thus, our goal is just to cover all statements in such slices. Assume that all such slices are summarized in a set S, and S_r' ($\subseteq S_r$) denotes the statements in S_r that are executed by a test. The coverage rate is defined as the proportion of statements in such slices that are covered by the test: $c = \frac{\sum_{S_r \in S} |S_r'|}{\sum_{S_r \in S} |S_r|}$.

Table 3: Benchmarks Used in Our Experiments

App name	LOC	#Class	#Method	#Activity
anarxiv	2,912	46	180	5
libreNews	1,882	39	107	2
simpleweather	7,034	145	520	6
archwikiviewer	994	25	106	2
forecastie	5,465	62	314	5
GoodWeather	6,116	77	426	6
Drinks	2,645	62	266	5
flym	16,938	271	1,264	5
openmanga	37,982	444	2,716	11
commons	42,142	1,044	4,305	15
openMensa	2,622	56	226	2
NWSWeather	3,183	56	313	5
OpenGappsDownloader	3,360	62	189	3
openHAB	16,103	238	1,314	5
NASAImages	2,247	40	146	1
AnkiDroid	39,012	461	2,630	21
DroidShow	11,768	99	614	6
Blitzortung	6,393	79	398	2
MaterialIOS	30,127	447	2,229	5
JustCarigslist	6,757	50	113	8
Goblim	4,363	96	301	5
kinolog	2,885	59	206	5
materialistic	25,676	560	2,690	23
giggity	10,000	111	599	5
movieDB	8,650	103	498	5
Average	**11,890**	**189**	**906**	**6**
10000NatureWallpapers	8,192	138	431	3
Freemeteo	46,243	922	5,621	19
AnimeMusic	16,295	328	1,701	3
TED	107,264	2,166	10,638	26
Pinterest	700,539	12,958	54,628	20
WEBTOON	194,820	3,285	19,092	56
4KWallpapers	13,873	273	903	6
wikipedia	95,721	2,007	10,128	34
translate	31,202	369	1,509	13
CBCNews	55,602	745	3,860	18
Average	**126,975**	**2,319**	**10,851**	**19**

Coverage rate. The second column of Table 4 summarizes the slice coverage of our approach (i.e., TNBDroid). Overall, the coverage rate is high: for the 25 apps from F-droid, the average coverage rate is 81.74%, and for the 10 apps from Google play, the average coverage rate is 66.43%. For 12 apps from F-droid, the coverage rate is over 90%, where for 6 of them (i.e., *libreNews*, *archwikiviewer*, *Drinks*, *OpenGappsDownloader*, *NASAImages*, *JustCarigslist*), the coverage rate is 100%. For commercial apps from Google play, the coverage rate is relatively lower; nevertheless, the highest coverage rate is still as much as 83.06%.

We expect the coverage rate is as high as possible. However, this is not reality. Here, we analyze the major reasons why the coverage rate is below our expectation:

(1) **Uncovered requests**. Since TNBDroid employs a breadth-first traversal for the testing, we cannot guarantee that all HTTP(S) network requests are covered. In fact, some network requests are difficult to trigger. Additionally, some apps also uses the other third-party libraries for the network request (e.g., *materialOS* uses `Bridge`), which is not completely covered in TNBDroid.
(2) **Uncovered exceptions**. There are some exceptions that are not easy to trigger by our fuzzing rules. Consequently, TNBDroid fails to reach the app code that deals with these exceptions.
(3) **Uncovered paths**. Even for the covered HTTP(S) network requests, there could be many branches and paths in the app code that handles the response data. Since the conditions of the branches and paths do not merely depend on the response data (also depend on other app variables), TNBDroid does not guarantee to cover all branches or paths.

4.2 Fault Detection Ability

The third column of Table 4 lists the bugs found by TNBDroid. A total of 62 bugs are detected, where 57 of them are relevant to the HTTP(S) network request. The other 5 bugs are bugs relevant to the app GUI. Besides the network response fuzzing, TNBDroid also explores the components in each activity following a priority strategy. Therefore, it is not surprised that it also detects 5 bugs relevant to the app GUI. We manually check the source code of the apps to validate the bugs, and ensure that all the bugs detected are real. Together with the code coverage results, this demonstrates the effectiveness of TNBDroid.

TNBDroid finds totally 26 bugs from the 25 apps from F-droid, with on average 1.04 bugs found from an app. TNBDroid detects a total of 30 bugs from the 10 apps from Google play, with on average 3 bugs found from an app. Surprisingly, the commercial apps involve more bugs relevant to the network usage. This is because the behaviour relevant to the network usage of the commercial apps is usually more complicated.

We classify the 57 bugs relevant to the network usage into three main categories (cf. Table 5). For each category of bugs, we first explain the reason why they exist, and then use a real case from our benchmarks to illustrate this category of bugs.

Null or empty data exception. It follows from Table 5 that more than half (34/57 = 59.65%) of the bugs belong to this category of exceptions, which mainly occur when the fuzzing rules used are `NULL`, `EMPTY`, `VALUE_NULL`, and `DEL`. By analyzing the source code of the app or the decompiled app code, we find that for the code which parses the response, it only judges whether the response is successful (*status* = 200) and whether the entire response content format is the expected. It does not determine whether the response content is null or empty, or whether the value of each field of the data (JSON, XML) is null. This can easily cause a null pointer exception when the app uses the values of the data or fields. For example, as shown in Figure 4a, when the *materialOS* app invokes the `toLowerCase()` method to convert the value (a string) of the variable *wallpapaer.url* into the lower case representation, it does not check whether *wallpapaer.url* is null. As a consequence,*materialOS* may crash when the URL is null. It is very difficult for existing GUI testing approaches to find this bug. It can only be detected in specific scenarios, e.g., the network signal is very weak. However, our approach (i.e., TNBDroid) can easily detect this bug: Under the fuzzing rule `VALUE_NULL`, the *wallpapaer.url* is set to be null, which causes the app crash (cf. Figure 4b). To avoid this category of bugs, the developer ought to determine in the code whether the response data is empty or null. If so, app developers should take measures to handle this exception.

Data type exception. The second category of bugs are caused by the data type exceptions, which accounts for 11/57 = 19.30%. We

Table 4: Experimental Results of RQ1-RQ3

App name	Code coverage	#Bugs found	Activity reduction rate	Time (min.)
anarxiv	82.35%	2(2)	3/5 = 60.00%	6
libreNews	100%	0	0/2 = 0.00%	5
simpleweather	96.38%	3(3)	4/5 = 80.00%	8
archwikiviewer	100%	1(1)	1/2 = 50.00%	12
forecastie	59.00%	1(1)	4/5 = 80.00%	6
GoodWeather	65.96%	0	1/5 = 20.00%	10
Drinks	100%	3(3)	2/5 = 40.00%	7
flym	55.13%	0	3/5 = 60.00%	13
openmanga	61.13%	0	2/7 = 28.57%	25
commons	50.51%	3(2)	4/10 = 40.00%	20
openMensa	75.00%	0	0/2 = 0.00%	13
NWSWeather	92.98%	0	2/5 = 40.00%	6
OpenGappsDownloader	100%	2(2)	1/3 = 33.33%	5
openHAB	64.52%	1(1)	1/3 = 33.33%	13
NASAImages	100%	2(2)	0/1 = 0.00%	8
AnkiDroid	79.07%	0	9/11 = 81.81%	8
DroidShow	97.85%	0	4/6 = 66.67%	6
Blitzortung	55.70%	0	0/2 = 0.00%	15
MaterialIOS	68.85%	2(2)	2/4 = 50.00%	10
JustCarigslist	100%	3(3)	5/7 = 71.43%	9
Goblim	94.35%	0	1/5 = 20.00%	9
kinolog	80.00%	2(2)	2/4 = 50.00%	13
materialistic	83.49%	1(1)	7/21 = 33.33%	10
giggity	87.85%	0	3/5 = 60.00%	9
movieDB	93.47%	2(2)	1/4 = 25.00%	13
Average	81.74%	1.08(1.04)	40.94%	10.36
10000NatureWallpapers	70.50%	5(4)	0/3 = 0.00%	25
Freemeteo	62.43%	3(3)	9/19 = 47.37%	44
AnimeMusic	72.58%	6(6)	1/2 = 50.00%	50
TED	78.26%	2(1)	12/21 = 57.14%	56
Pinterest	55.14%	7(7)	3/6 = 50.00%	53
WEBTOON	58.64%	4(2)	12/41 = 29.27%	70
4KWallpapers	83.06%	0	2/6 = 33.33%	15
wikipedia	62.78%	2(2)	5/17 = 29.41%	45
translate	61.59%	0	4/8 = 50.00%	31
CBCNews	59.36%	6(5)	2/10 = 20.00%	35
Average	66.43%	3.50(3.00)	36.65%	42.40

Table 5: Category of Network Usage Bugs

Category	Type	#Bugs
Null or empty data exception	`java.lang.NullPointerException`	32
	`java.lang.ArrayIndexOutOfBoundsException`	2
Data type exception	`android.content.ActivityNotFoundException`	3
	`java.lang.NumberFormatException`	3
	`java.util.regex.PatternSyntaxException`	1
	`java.lang.UnsupportedOperationException`	1
	`java.net.URISyntaxException`	1
	`kotlin.TypeCastException`	1
	`org.json.JSONException`	1
Uncaught exception	`java.lang.IllegalArgumentException`	6
	`java.lang.IllegalStateException`	3
	`java.io.IOError`	2
	`rx.exceptions.OnErrorNotImplemented Exception`	1

```
public  static  void download(final  Activity  context,  Wallpaper wallpaper,
   final  boolean apply){
     mContextCache = context;
     mWallpaperCache = wallpaper;
     mApplyCache = apply;
     if  (Assent.isPermissionGranted(AssentBase.WRITE_EXTERNAL_STORAGE)
         || apply){
           File  saveFolder;
           String  name;
           String  extension  = wallpaper.url.toLowerCase(Locale.getDefault()).
          endsWith(".png") ? "png" : "jpg";
          ......
     }
}
```

(a)

```
java.lang.NullPointerException: Attempt to invoke  virtual  method
   'java.lang.String java.lang.String.toLowerCase(java.util.Locale)'
      on a  null  object  reference
   −at com.afollestad.polar.util.WallpaperUtils.
     download(WallpaperUtils.java:337)
   −at com.afollestad.polar.viewer.ViewerPageFragment.
     onOptionsItemSelected(ViewerPageFragment.java:81)
    ......
```

(b)

Figure 4: A bug caused by "null or empty data": (a) The code snippet from *materialOS* (version 1.0.1), (b) Crash information of *materialOS* (Under the fuzzing rule `VALUE_NULL`, *wallpaper.url* is null, and a null pointer exception occurs when calling `toLowerCase()`).

first explain how the data type exceptions occur to the network response data. In the real-world implementation, the response data is transmitted in the form of a binary stream. To facilitate the use of the response data, the obtained data ought to be first parsed and converted into the respective types (i.e., structured data). For some types of data, if the converted data does not satisfy the required format, type conversion exceptions tend to occur. For instance, in Figure 5a, the *Drinks* app employs the *intent* object to start the browser, and the Uri parameter of *intent* is the data returned by the network request (i.e., $intent.\ setData(Uri.parse(this.drink.getWikipedia())))$. Under our fuzzing rule `VALUE_RAND`, the data is replaced with a random value, which does not meet the URI format requirement. This causes the browser to fail to open the expected URI and thus an exception is threw (cf. Figure 5b). To avoid such bugs, after app developers use these type conversion methods, they ought to first verify the converted data before using it.

Uncaught exception. The last category of bugs are caused by the uncaught exceptions, which accounts for 12/57 = 21.05%. We first explain why there are uncaught exceptions. Many methods in the app perform parameter validation to check whether the argument passed in is a non-empty or empty string. An exception will be threw directly if the argument is not valid. If the app does not catch the exception, it will probably crash. Let us illustrate this with an example. *Picasso* is a powerful image downloading and caching library for Android. It is frequently used in the commercial apps (e.g, *10000NatureWallpapers*). Figure 6a presents the source code of the `load()` method of the *Picasso* framework. If the argument of *path* passed to `load()` is an empty string (i.e., "", not null), *Picasso* will

```
void lambda$setupViews$0(View v){
   Intent  intent  = new Intent("android.intent.action.VIEW");
   intent.setData(Uri.parse(this.drink.getWikipedia()));
    startActivity  (intent);
     }
}
```

(a)

```
android.content.ActivityNotFoundException:
    No Activity  found to  handle  Intent
    {act=android.intent.action.VIEW dat=R$4Fwkz}Y/U0tQ}
       W[js]j]uUmUT|3*ThY9^Tq1UXP9.S]jCXK}
−at android.app.Instrumentation.
     checkStartActivityResult (Instrumentation.java:1798)
−at android.app.Instrumentation.
     execStartActivity (Instrumentation.java:1512)
   ......
```

(b)

Figure 5: A bug caused by "data type exception": (a) The code snippet from *Drinks* (version 2.0.8), (b) Crash information of *Drinks* (Under the fuzzing rule `VALUE_RAND`, the random data does not satisfy the URI format, and the intent uses the invalid URI to start other apps, resulting in `ActivityNotFoundException`.

throw an exception, i.e., "java.lang.IllegalArgumentException: Path must not be empty". The *10000NatureWallpapers* app loads images through `Picasso.load()` (cf. Figure 6b). When the parameter *path* is set to be an empty string ("") with our fuzzing rule `VALUE_BOUND`, the app will finally crash due to this, because it does not catch the exception. Accordingly, to avoid such bugs, app developers should know whether the method will throw an exception before calling it, and ensure that the parameters passed in are correct or expected. As an alternative, app developers can also implement the `UncaughtExceptionHandler` interface in the app to catch all unintended exceptions. In this way, the app will not crash due to these exceptions, and thus the user experience can be improved.

4.3 Activity Reduction Rate and Efficiency

Activity reduction rate. TNBDroid is a traversal-based testing technique. Hence, its performance significantly depends on the traversal strategy. In general, we employ a breadth-first traversal. To accelerate the automatic testing, we only focus on the activities that can directly trigger the HTTP(S) network request. To this end, in the third stage (i.e., pre-exploration) of TNBDroid, we first determine the so-called network activities. Then, in the last stage (i.e., testing) of TNBDroid, if an activity is not a network activity, it can be skipped without intensive testing. The activity reduction rate is exploited to evaluate the effectiveness of our pre-exploration, which is defined as the proportion of activities that can be skipped in the testing stage: $r = \frac{|A \setminus A_N|}{|A|}$, where A_N and A are the set of network activities and the set of activities that are pre-explored (have not intensively tested yet), respectively. It is worth mentioning that A does not necessarily include all the app activities, because certain activities can be missed by during our breadth-based traversal.

The fourth column of Table 4 summarizes the activity reduction rate of TNBDroid in the final traversal-based testing. It follows that

```
public RequestCreator load(String path){
        if (path == null){
            return new RequestCreator(this, null, 0);
        }
        if (path.trim().length() != 0){
            return load(Uri.parse(path));
        }
        throw new IllegalArgumentException("Path must not be empty.");
}
```

(a)

```
public View getView(int position, View convertView,
   ViewGroup parent){
     Picasso.with(this.context)
        .load((String) this.appStoreLinks.get(position))
        .placeholder((int) C0487R.drawable
        .progress_animation).noFade()
        .resize((int) this.scrWidth)/2, (int) this.newHeight)
        .centerCrop().into(calculatorImage);
      ......
}
```

(b)

Figure 6: A bug caused by "uncaught exception": (a) The code snippet from *Picasso*, (b) *10000NatureWallpapers* (version 3.13) uses *Picasso* (Under the fuzzing rule `VALUE_BOUND`, `this.appStoreLinks.get(position)` is "", an exception will be thrown in `Picasso.load(path)`, and the `getView()` method will not capture, causing the app crash.

the reduction is effective. The average activity reduction rate of the 25 apps from F-droid is 40.94%, and that of the 10 commercial apps from Google Play is 36.65%. For the *AnkiDroid* app, the activity reduction rate is as much as 9/11 = 81.81%. For the commercial app *TED*, the activity reduction rate reaches to 12/21 = 57.14%.

Efficiency. Usually, the traversal-based testing is time-consuming. Besides the activity reduction, our breadth-based traversal and the testing termination conditions together ensure the efficiency of the testing. The last column of Table 4 reports on the runtime cost of the last testing stage. It follows that TNBDroid is efficient. The average testing time of the 25 apps from F-droid is 10.36 minutes: The minimum and maximum testing time is 5 minutes (for apps *libreNews* and *OpenGappsDownloader*) and 25 minutes (for app *openmanga*), respectively. The average testing time of the 10 commercial apps from Google Play is 42.40 minutes: The minimum and maximum testing time is 15 minutes (for app *4KWallpapers*) and 70 minutes (for app *WEBTOON*), respectively. The total testing time of the 35 apps are 683 minutes.

4.4 Threats to Validity

Construct validity. In our experimental evaluation, we employ the open-source tool Extractocol to obtain the slice (a set of statements) that could be impacted by the response data of an HTTP(S) network request. The code coverage of TNBDroid is calculated based on the slice obtained. Hence, the slice returned by Extractocol significantly impact the code coverage of our approach. That is, if we substitute another tool for Extractocol, our experimental results on slice coverage could be changed.

External validity. TNBDroid requires instrumenting the app for the testing. If the source code of the app is available, TNBDroid works well. Otherwise, TNBDroid relies on Soot to instrument and repackage the app. Since the developers of many commercial apps take measures to prevent others from intruding their apps, e.g., signature, obfuscation [14], reinforcement, Soot may fail to repackage the instrumented apps and make them runnable. Therefore, our experimental results on commercial apps may not be generalized to these protected apps.

5 LIMITATIONS

In this section, we discuss some relevant issues that are not considered in our current work.

We discuss the issues TNBDroid does not consider.

Targeted test cases. Our aim is to test all HTTP(S) network-related behaviour of apps. However, TNBDroid is traversal-based testing approach, which inevitably takes many detours, though activity reduction is considered. Ideally, one may argue that we can design targeted test cases, each of which can start from the main activity to directly trigger a network request. To achieve this, a precise and complete state machine (or activity transition graph) of the app is usually required, which is also challenging.

Socket requests. TNBDroid only takes HTTP(S) network requests into consideration, because the HTTP(S) response data contains sufficient information that is necessary for applying our fuzzing rules. Socket requests are not covered in our approach. This is because the return data types of socket requests (i.e., APIs of `java.net.Socket`) is not known easily, and thus our fuzzing rules are not directly applicable.

6 CONCLUSIONS

Bugs relevant to the network usage accounts for a large proportion of app crashes. This paper proposes a novel approach and an open-source tool TNBDroid to systematically test HTTP(S) network-related behaviour of Android apps. Based on the static analysis, our approach first finds appropriate positions in the app code to parse the real response data of the HTTP(S) network request and then instrument accordingly the fuzzing data for intensively test the behaviour relevant to the network usage. At runtime, the instrumented app is automatically explored to trigger the HTTP(S) requests and our fuzzing response data validates whether the app can properly handle different responses. The experiment on 35 real-world apps demonstrates that TNBDroid covers on average 77.37% of the HTTP(S) network-related behaviour, and discovers from the 35 apps a total of 57 bugs related to the HTTP(S) network usage in 683 minutes.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. 2011. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *the Fourth IEEE International Conference on Software Testing, Verification and Validation, Berlin, Germany, 21-25 March, Workshop Proceedings*. 252–261.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7*. 258–261.

[3] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE'12, Cary, NC, USA - November 11 - 16*. 59:1–59:11.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, Edinburgh, United Kingdom - June 09 - 11*. 29.

[5] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13, part of SPLASH'13, Indianapolis, IN, USA, October 26-31*. 641–660.

[6] Young Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE'16, Singapore, September 3-7*. 238–249.

[7] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. 2018. EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps. *IEEE Trans. Software Eng.* 44, 5 (2018), 470–490.

[8] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13, part of SPLASH'13, Indianapolis, IN, USA, October 26-31*. 623–640.

[9] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13, part of SPLASH'13, Indianapolis, IN, USA, October 26-31*. 623–640.

[10] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE'15, Lincoln, NE, USA, November 9-13*. 429–440.

[11] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS'17, Dallas, TX, USA, October 30 - November 03*. 2123–2138.

[12] Google. 2015. The Monkey UI Android testing tool. http://developer.android.com/tools/help/monkey.html.

[13] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering, ICSE'19, Montreal, QC, Canada, May 25-31*. 269–280.

[14] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products. In *Proceedings of the 40th International Conference on Software Engineering, ICSE'18, Gothenburg, Sweden, May 27 - June 03*. 421–431.

[15] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST'11, Waikiki, Honolulu, HI, USA, May 23-24*. 77–83.

[16] Xinyue Huang, Anmin Zhou, Peng Jia, Luping Liu, and Liang Liu. 2019. Fuzzing the Android Applications With HTTP/HTTPS Network Data. *IEEE Access* 7 (2019), 59951–59962.

[17] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek. 2019. Search-based energy testing of Android. In *Proceedings of the 41st International Conference on Software Engineering, ICSE'19, Montreal, QC, Canada, May 25-31*. 1119–1130.

[18] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2016. Energy-aware test-suite minimization for Android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'016, Saarbrücken, Germany, July 18-20*. 425–436.

[19] Casper Svenning Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20*. 67–77.

[20] Jeongmin Kim, Hyunwoo Choi, Hun Namkung, Woohyun Choi, Byungkwon Choi, Hyunwook Hong, Yongdae Kim, Jonghyup Lee, and Dongsu Han. 2016. Enabling Automatic Protocol Behavior Analysis for Android Applications. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies, CoNEXT'16, Irvine, California, USA, December 12-15*. 281–295.

[21] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, Li Zhang, Börje F. Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, and Feng Zhao. 2014. Caiipa: automated large-scale mobile app testing through contextual fuzzing. In *The 20th Annual International Conference on Mobile Computing and Networking, MobiCom'14, Maui, HI, USA, September 7-11*. 519–530.

[22] Jun Ma, Shaocong Liu, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. 2018. LESdroid: a tool for detecting exported service leaks of Android applications. In *Proceedings of the 26th Conference on Program Comprehension, ICPC '18, Gothenburg, Sweden, May 27-28*. 244–254.

[23] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26*. 224–234.

[24] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE'14, Hong Kong, China, November 16 - 22*. 599–609.

[25] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'16, Saarbrücken, Germany, July 18-20*. 94–105.

[26] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. NetDroid: summarizing network behavior of Android apps for network code maintenance. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC'17, Buenos Aires, Argentina, May 22-237*. 165–175.

[27] Wipawee Nayam, Arguy Laolee, Luck Charoenwatana, and Kunwadee Sripanidkulchai. 2016. An analysis of mobile application network behavior. In *Proceedings of the 12th Asian Internet Engineering Conference, AINTEC '16, Bangkok, Thailand, November 30 - December 2*. 9–16.

[28] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: beyond GUI testing for Android applications. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE'17, Urbana, IL, USA, October 30 - November 03*. 27–37.

[29] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'17, Paderborn, Germany, September 4-8*. 245–256.

[30] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, Mississauga, Ontario, Canada*. 13.

[31] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE'18, Montpellier, France, September 3-7*. 738–748.

[32] Weilun Xiong, Shihao Chen, Yuning Zhang, Mingyuan Xia, and Zhengwei Qi. 2018. Reproducible Interference-Aware Mobile Testing. In *IEEE International Conference on Software Maintenance and Evolution, ICSME'18, Madrid, Spain, September 23-29*. 36–47.

[33] Shengqian Yang, Dacong Yan, and Atanas Rountev. 2013. Testing for poor responsiveness in android applications. In *The 1st International Workshop on the Engineering of Mobile-Enabled Systems, San Francisco, CA, USA, 25-25 May 2013*. 1–6.

[34] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE'13, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS'13, Rome, Italy, March 16-24. Proceedings*. 250–265.

[35] Li Lyna Zhang, Chieh-Jan Mike Liang, Yunxin Liu, and Enhong Chen. 2017. Systematically testing background services of mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE'17, Urbana, IL, USA, October 30 - November 03*. 4–15.