IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 26, 2019, accepted May 1, 2019, date of publication May 7, 2019, date of current version May 20, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2915339

Fuzzing the Android Applications With HTTP/HTTPS Network Data

XINYUE HUANG[1], ANMIN ZHOU[1], PENG JIA[2], LUPING LIU[2], AND LIANG LIU[1]
[1]College of Cybersecurity, Sichuan University, Chengdu 610065, China
[2]College of Electronics and Information Engineering, Sichuan University, Chengdu 610065, China

Corresponding author: Liang Liu (liangzhai118@163.com)

ABSTRACT Nowadays, the number of mobile netizens continues to grow, mobile life continues to infiltrate people's lives. Mobile applications play an increasingly important role in major industries (financial consumption, travel, education, and entertainment). High dependence and complexity make network communication become an important attack surface of mobile applications. How to quickly and efficiently discover security threats in the process of network interaction has become an urgent problem. This paper proposed a test method based on network packets fuzzing for Android applications. The scheme uses middleman technology to obtain the interaction data sent by servers to applications, adopts different mutation strategies to mutate the original data of different types, returns the mutated response data to applications, uses log monitoring technology to monitor crash information, thereby discovers potential security threats. 10 popular applications were tested based on the proposed method, and four kinds of problems were discovered. The problems contain unresponsiveness, crashes caused by JSON data exception, HTML content replacement, and URL redirection. The results indicated that the proposed method was effective in exposing bugs of mobile applications in the process of network data interaction.

INDEX TERMS Android applications, Fuzzing, HTTP/HTTPS, network data.

I. INTRODUCTION

Nowadays, with the continuous development of the mobile network, Android applications face more and more security threats. These threats come from many sources [1]–[3], and irregular/unexpected input data is one of the main reasons. For Android applications, data coming from the servers, files saved in local storage and operations performed by users are three main types of input data [4]–[6].

At present, the main idea of mining and discovering the potential security threats based on input data, is to use fuzzing technology to mutate regular input data, and then check how the applications process the mutated data. Fuzzing is the technical means to discover applications' code/logical defects by continuously inputting irregular data to them. It is one of the most common methods used in software security test [7]–[9]. File-based fuzzing technology is mature, and many fuzzing frameworks are available on Linux/Windows platforms (afl [10], Peach [11], etc.). User-based fuzzing technology mainly for the Android platform, which automatically manipulates applications and monitors if the applications crash, has test frameworks such as Monkey [12] and MonkeyRunner [13]. Traditional network-based fuzzing is mainly used to test server-side applications (web applications, database applications, etc.), which is not effective and has limited support for massive mobile applications.

Considering the above problems, we proposed a fuzzing method based on network packets, to discover security threats and vulnerabilities of Android applications caused by the response data from servers. The main contributions of this paper are summarized as follows.

Novel: The main processing methods of Android applications on network data were summarized. Facing the plenty of data interaction between an Android application and its server, some valuable test objects were selected from a large number of HTTP/HTTPS response data, which reduced the difficulty of testing. At the same time, in order to adapt to the significantly different response data between various Android applications, a series of fuzzing strategies were developed. The proposed fuzzing method explores the potential security threats in Android applications from a new perspective.

Feasibility: In file-based fuzzing, mutated files can be repeatedly opened/loaded, different from this, mutated

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

response data sent to an application will not be effectively processed without operating the application according to a particular sequence. Additionally, before triggering a crash, for the same test input data, fuzzing technology needs to generate variant samples and perform multiple tests. Therefore, how to achieve reproducibility is the main problem need to be solved when performing network-based fuzzing, to improve test efficiency. The fuzzing method proposed in this paper solved this problem by selecting appropriate test objects and formulating general fuzzing strategies.

Effectiveness: Experimental results proved the effectiveness of the proposed method. A corresponding fuzzing framework AAHF (standing for Android App HTTP/HTTPS Fuzzing) were implemented, and 10 popular Android applications were tested based on it. The test duration of each app was limited to 1-3 hours, and 4 kinds of problems, containing no responding, crash caused by JSON data, HTML content replacement and URL redirection, were discovered. Each tested application contained at least one of these problems. The results indicated that in addition to users' private data leakage, incorrect network communication data also would bring security threats to Android applications.

The following content of this paper are organized as follows. Section II introduces the relevant research. Section III introduces the proposed network-based fuzzing method. The effectiveness of the method is evaluated based on the experiments in Section IV, and Section V discusses the limitations and future work.

II. BACKGROUND

A. NETWORK-BASED FUZZING ON ANDROID APPLICATIONS

Currently, fuzzing on Android applications' network data is focused on searching for server-side security issues. The core idea of server-side fuzzing is to mutate the request packets sent by mobile applications, to exploit possible vulnerabilities on the server side. Zuo *et al.* [14] discovered authentication vulnerabilities existing on the server side by processing HTTP request data of mobile applications, using differential feature analysis to identify protocol domains, automatically replacing the domains and observing the server's response. In their earlier research [15], a test framework named AUTOFORGE was proposed, which included API hooking, lightweight reverse engineering on protocol, request message forgery and other black box test techniques. AUTOFORGE tests whether the server contains enough checks to ensure the security of users' accounts, by forging valid client HTTP request messages and sending them to the server. Ziqiang *et al.* [16] proposed an automation platform for mobile applications' security detection, which used a fuzzing method based on HTTP protocol, to detect server-side security vulnerabilities.

Compared with the above research, we focus on client-side fuzzing targeted on Android applications, fuzz Android applications by mutating response packets returned by servers, to explore the impact and harm of various mutated network communication data on them.

One similar work is IOTFUZZER [17], which automatically generates protocol-aware fuzzing messages to IoT devices from IoT apps, trys to discover memory corruption in firmware images. Our test objects and methods are different.

1) For IOTFUZZER, it targets the protocols between IoT devices and IoT apps. The communication protocol changes with different IoT devices and IoT apps, therefore, it needs to analyze the app to find the fields before testing. For our method, as the format of the HTTP protocol is publicly known, we do not need to analyze apps. Although it will lead to insufficient targeting of our test cases when considering the possible encryption and confusion in apps, this greatly improves test efficiency and reduces test difficulty.
2) IoT apps have more specific functions, and the protocol is not as complicated as the HTTP protocol, so IOTFUZZER performs data variation by hooking the key APIs and changes their parameters. For HTTP, much code involved in network processing in apps make it difficult to locate key hooking objects, and we found that it was easier to capture network packages by means of a middleman agent.

Another similar work is JazzDroid [18], an automated Android gray-box fuzzer that injects into applications various environmental interference on the fly to detect issues. It contains a network issues injector interposes on the HTTP requests from apps and manipulates their responses to simulate network delay and connectivity. In contrast, we focus on the impact that incorrect response data can have on applications, not just including network delay and connectivity.

B. NETWORK DATA ANALYSIS OF ANDROID APPLICATIONS

At present, researches on network data analysis of Android applications mainly focus on the following aspects: device information inference, application information inference, and privacy information leakage detection [19]–[21], etc.

Regarding device information inference, Malik *et al.* [22] used time interval of messages generated by mobile devices to infer operating systems running on devices, successfully distinguished between Android, iOS and Windows Phone. Application information inference includes application identification and malicious application detection. SAMPLES [23] is an adaptive application identification framework that abstracts the appearance pattern of application identifiers in HTTP traffic into a set of text rules containing HTTP fields, prefixes, and suffix strings of application identifiers. AppScanner [24] extracts 54 statistical features from receiving data, sending data, and bidirectional network flows, trains random forest classifiers to identify applications on networks, and evaluates the impact of several factors on detection rate. In the field of private information leakage detection, the method proposed by Continella *et al.* [25] utilizes black box difference analysis technology. They used

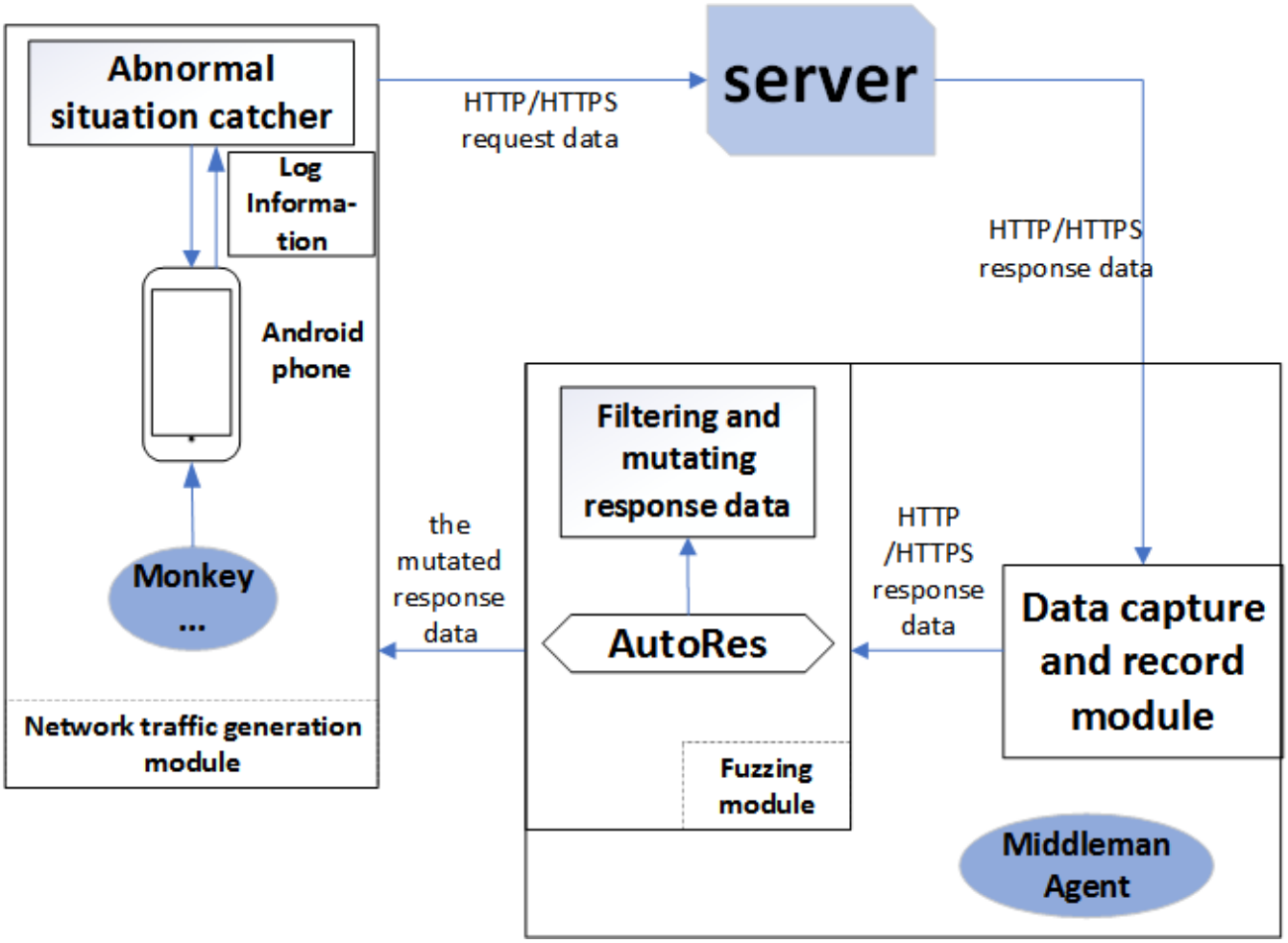


FIGURE 1. Schematic. *(Monkey is used to click on the application to trigger its network data. The possible security risks is identified according to the recorded server response data and the application log information.)*

differential analysis to detect data changes in corresponding network traffic after the PII (Personally identifiable information) data changed, to detect leaked PII information (even if it has been encrypted or confused).

III. FUZZING APPS WITH MUTATED NETWORK DATA

In this section, the main principle and processing flow of the proposed fuzzing method are summarized in Section A, the method is described in detail in Section B. Before explaining how to achieve the repeatability of fuzzing, we first introduce the determination of fuzzing objects and the specific content of the general fuzzing strategies.

A. OVERVIEW

In order to implement fuzzing on HTTP/HTTPS response data received by an Android application, we need to operate the application to trigger various network operations. The server generates and returns corresponding response data after receiving HTTP/HTTPS requests sent by the application. The method first intercepts response data by means of a middleman agent, mutates the data, then returns it to the application to test its ability on processing response data beyond expectations. The specific processing flow of the method is shown in Fig.1.

Scope and Assumptions: The Fuzzing method proposed in this paper focuses on testing Android applications using the HTTP/HTTPS protocol. For HTTPS, the content is encrypted, we use Fiddler (a free web debugging proxy) [26] to capture and decrypt the network data of applications, mutate the successfully decrypted plaintext data. The scheme mutates packets to discover the following security risks in applications: whether the HTTP/HTTPS response data are validated, whether there is a correct handling mechanism for various malformed data. Currently, the communication channel between the target application and the middleman agent is Wi-Fi, in the future our framework could be extended to other channels with some additional efforts.

B. FUZZING

1) FUZZING OBJECTS

The input data of Android applications, which may cause security issues, can be divided into the following three categories.

TABLE 1. Fuzzing objects. *(The test objects listed in this table are the common response data fields of HTTP and HTTPS. We do not study HTTPS' different fields from HTTP.)*

Test Objects	
Response Header	Content-Length Location
Status Code	2XX 3XX
Response Body	application/octet-stream application/json text/plain text/javascript text/html image/png image/jpeg image/gif image/icon image/webp

TABLE 2. The proportion of various status codes in the response data of the app *com.tencent.mobileqq. (We selected response data belonging to 2XX and 3XX as test objects, which covered a large part of the response data of an application.)*

Status Code	Proportion(%)
2XX	97.15
3XX	0.73
4XX(5XX)	2.12
1XX	0

1) Users' operations.
2) Files with various formats.
3) Various data from network packets.

Research in this paper focuses on HTTP/HTTPS response data from servers. Android applications receive numerous response data, but some of them have no meaning for testing. According to the specific structure of HTTP response data (status line, response header, blank line, response body), we studied and determined appropriate test objects. Finalized test objects are organized in Table 1.

Response data containing informational status code 1XX (indicating that the server is processing requests) and error status code 4XX/5XX do not have an additional impact on the operation of Android applications (they will only perform simple judgment on such response data). Therefore, we limited test objects to response data containing success status code 2XX (indicating that the server has processed requests and returned data) and redirect status code 3XX (requires applications to perform additional operations to complete requests).

As shown in Table 2, when operating an application, 97.15% of the response packets belong to returned data after successful processing, and 0.73% are redirected data. For the former, we focused on *the response body specified by response headers "Content-Type" and "Content-Length"*. For the latter, we focused on *the data containing the response header "Location"*. The "Location" header contains the redirect URL. If there is no legality judgment on this field, attackers may induce the application to access malicious address by replacing the URL. We finally determined test objects as follows.

1) *The response data with status code 2XX and "Content-Length" not equaling 0.*
2) *The response data with status code 3XX and containing "Location" in the response header.*

2) FUZZING STRATEGIES

Considering the significant differences on the structure and content of network data between different applications, we summarized the general processing procedures and methods used by Android applications on network data, formulated specific strategies based on these methods, to develop suitable general fuzzing strategies.

Nowadays, general processing methods on network data are as follows.

1) *Content display*, applications display content such as HTML and image in response packets.
2) *Download file resources*, some response data contain links pointing to the data required by applications, the application will open the links to download required data.
3) *Control the execution flow of applications*, the response data in JSON format returned by the server have data that may affect the execution flow of an application, the application executes different code according to different data.

In previous test, we captured the response data by running applications (*com.ifeng.newvideo*, *com.culiukeji.huanletao*) and counted the proportion of each type. We noticed that JSON data widely existed in various response data. Since JSON data have a key-value pair format of *{key:value}*, we extracted key-value pairs from the response data, mutated the value of common data type (*int, float, boolean, string, null*) by adopting the following heuristic mutation rules.

1) *Changing the length and content of strings for stack-based or heap-based overflow and out-of-bound access.* In our implementation, the original strings are replaced by random strings of the same length or appended with a variable number of special characters such as "//"," ", to construct malformed messages.
2) *Changing the integer or float values for integer overflow and out-of-bound access.* We mutated the original values into boundary cases and large values. Also, to trigger the cases of miscounting of boundary conditions, we also generate the off-by-one values.
3) *Changing the types, or providing empty values for uninitialized variable vulnerability.*

For the other data with higher frequency (image, HTML, etc.) shown in Fig.2, we refered to the idea of file-based fuzzing, generated a large number of test cases in advance using corresponding open source fuzzing tools (zzuf [27], domato [28]).

The general fuzzing strategy is finally formulated as shown in Fig.3. After intercepting the response data from the server, the status code is first obtained. If the code is outside the range (2XX/3XX), no further processing will be performed. Response data with status code of 2XX/3XX will be mutated according to the following strategies.

1) When status code is 302, get the value of "Location" field in response header, replace it with other URLs (such as Baidu, Google).

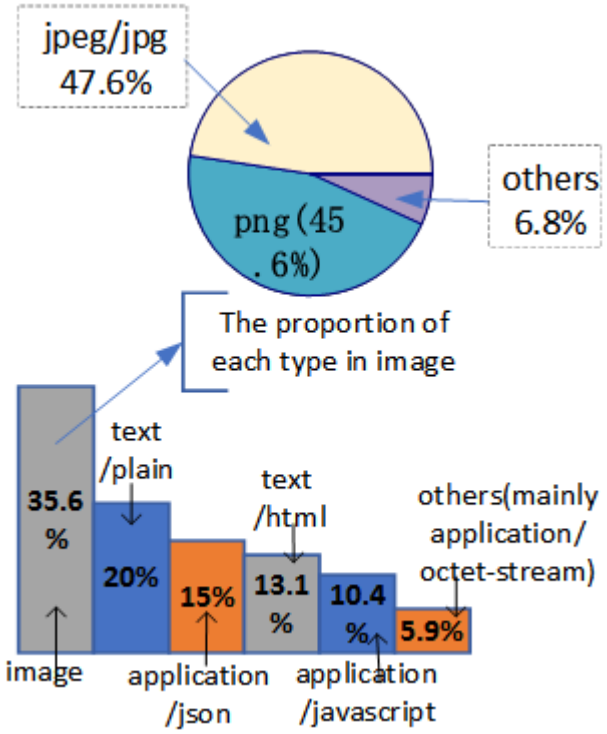


FIGURE 2. **The proportion of various types of content in network packets of the app (*com.ifeng.newvideo, com.culiukeji.huanletao*). *(We selected image, JSON, HTML, javascript, which has the highest proportion in the figure, as test objects. For plain and octet-stream, we only considered the plaintext JSON data.)***

2) When status code is 2XX/3XX, get the value of "Content-Type" field in response header, if it is in the range (text/html, image/gif/jpeg/png/icon/webp, text/javascript, application/json) and the value of "Content-length" field is not 0, according to test rules of each type, the content in response body is mutated and replaced, and then sent to the application.

3) REPEATABILITY OF FUZZING

An Android application only processes one response for one request. Repeatedly sending the same response packet (carrying different mutated data) to the application cannot repeatedly trigger its processing code for the data. This is not conducive to the repetitiveness of fuzzing. In the research process, we found that for response data used for content display:

1) The same application had the same parsing code for the same type of data (e.g., the application uses the same image parsing library to process various image data contained in response data.).
2) The application would receive numerous response data of the same type during the same test.

1) and 2) means that during the execution of an application, there are multiple response packets belonging to different request data (e.g., different response data with different image data are received by the application when operated different pages.) that could trigger the same processing code (e.g. image parsing code in the application) repeatedly.

Therefore, multiple response data of the same type (e.g. images) received during the execution of the same application can be regarded as effective carriers of the same type of mutated test cases. This partially achieves the repetitiveness of fuzzing. Specifically, for each response data whose transmission content is image/HTML, the original content is replaced by different samples generated before. For data of JSON format, since different contents could have different impacts on execution of the application, the action script of the application is recorded with the command line tool replaykit [29], which provides the ability to record touchscreen events from one device, to ensure the same execution flow each time. Multiple recurring executions of script implement repeated testing.

IV. EVALUATION

A. EXPERIMENT SETUP

Based on the proposed method, we implemented a fuzzing framework AAHF (standing for Android App HTTP/HTTPS Fuzzing) and performed experiments to verify its effect. The experiments were conducted on a Google Nexus5 mobile phone with operating system version 4.4.4 (It is worth noting that the method we proposed is for Android applications, not limited to the specific version of the Android operating system.) and a 64-bit win10 computer. The test samples are 10 widely used Android applications. The fuzzing algorithm of AAHF is shown in Fig.4.

As shown in Algorithm 1, AAHF first checkes whether the app has applied for network permissions, then replays the previously recorded script and initializes the app. AAHF starts the fuzzing and monitors whether the app page is replaced with the content of our test samples to determine if there is an HTML content replacement problem. After the test, through the log information recorded, we statistic the crash and ANR information. With the network data recorded during the test, we check whether there are response data from the replaced URL to check whether there is an URL hijacking problem.

B. EVALUATION RESULT

The purpose of the experiment is to verify the validity of AAHF, that is, to be able to discover the security problems caused by unexpected network communication data. In this paper, applications' security problems are classified as follows.

1) *Crash*, apps crash and exit.
2) *ANR (Application Not Responding)*, apps fail to respond to users' actions, and the Android system would display a popup window to prompt users to kill the process or wait for it.
3) *Redirected access*, apps visit the redirected URL in response data without checking.
4) *HTML content replacement*, apps display the HTML content contained in response data without verification.

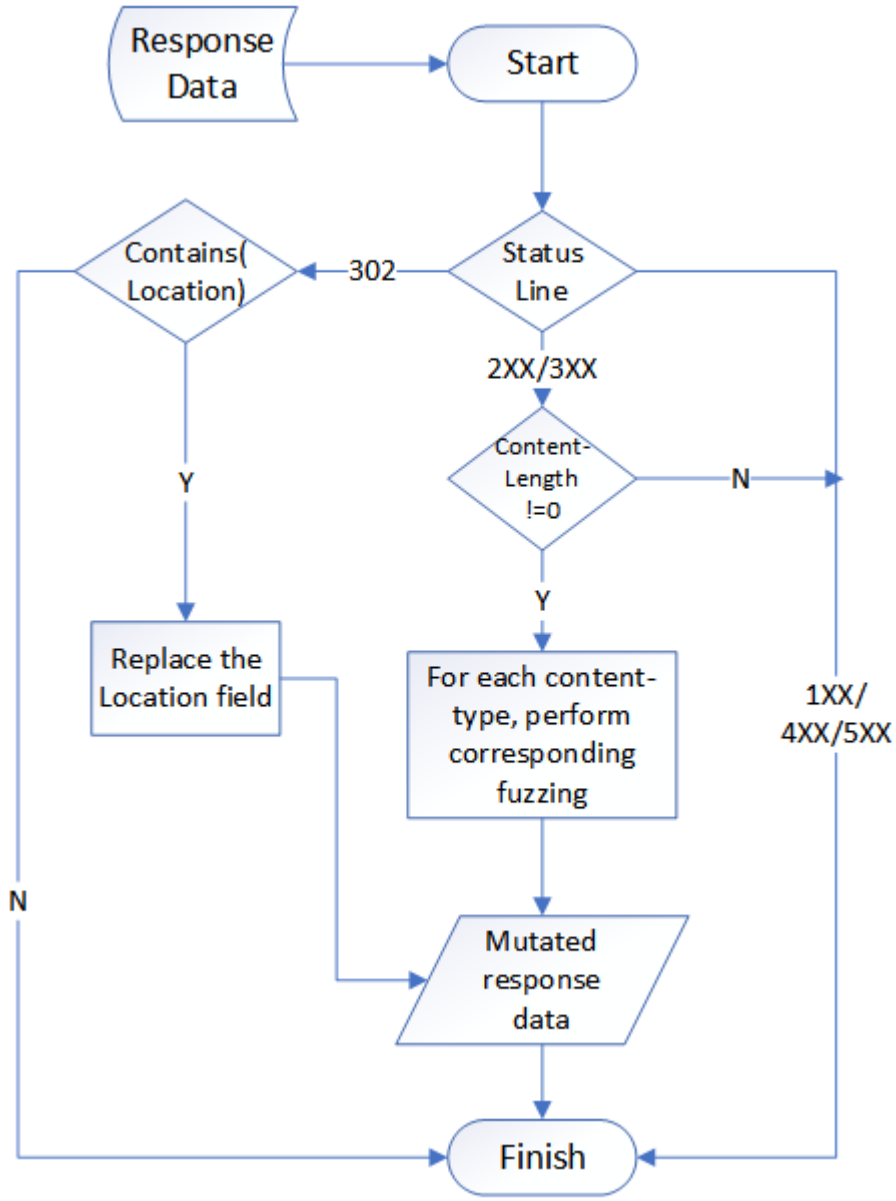


FIGURE 3. The general fuzzing strategy.

The experimental results indicated that AAHF can effectively find the above problems in each app during 1-3 hours' testing. The results are shown in Table 3.

1) ANR

As can be seen from Table 3, AAHF found ANR in 9 of the 10 apps. A total of 33 activities with ANR problems were discovered. The error log information and ratios are shown in Table 4.

As can be seen from Table 4, AAHF discovered two kinds of ANR.

1) There is no window for the application.
2) There is a window but the waiting queue of input events is not empty.

Next, we analyzed the two cases.

AAHF automatically operates applications using Monkey. The first case is caused by Monkey's fast operations. When switching between two pages, one page is being initialized, the other page has been ''stacked'' (i.e., loses focus). At this point, clicking the back button quickly will cause the application to become unresponsive. This is triggered by the ANR monitoring strategy of the Android system and will not appear when the application is executed normally. The second case has the following possible reasons.

1) AAHF performs some operations on the current page created by the application, and this operation will keep waiting until getting some specific response data from the server. If some other actions are performed during waiting, the application will become unresponsive.

Algorithm 1 AAHF's fuzzing algorithm

Input: $Testing\_App$ list, $able$
Output: $Test\_results$ list

```
function FUZZING(Testing_App, able) ▷ able-a boolean value controls whether to start the test or not.
    if able then
        n ← len(Testing_App) − 1
        for i = 1 to n do
            it ← Testing_App[i]
            CHECKAPP(it)
            Test_results ← MUTATION(it)
        end for
    end if
    return Test_results
end function

function MUTATION(App)
    A.Use Monkey to oprate App.
    B.Capture and Mutate App's network data according to the fuzzing strategies.
    C.Send the mutated data to App.
    D.Detect and capture App's log information.
    return App's log information and network data.
end function

function CHECKAPP(App)
    if App applies for network permissions then
        Install it on the phone.
        Playback pre-recorded Apps initialization script.
    end if
end function
```

FIGURE 4. AAHF's fuzzing algorithm.

2) Synchronous network request operations may cause thread blocking, and it will cause the application to be unresponsive if the blocking in the main thread lasts for more than 5 seconds. Although Android 4.0 and later system prohibit synchronous network request operation in the main thread, if the main thread needs to wait for the execution results of network operations in the child thread, and the child thread is blocked, the main thread will be blocked too, which also may trigger ANR.

TABLE 3. Test results of 10 apps. *(ANR indicates the number of activities that have ANR in the app. Downloads indicates the number of downloads of the app in a well-known app market such as Google, 360.)*

	Package name	Crash of java layer	Crash of native layer	ANRs	Redirection	Html content replace	Downloads	Test Time(/h)
1	com.amazon.mShop.android.shopping	2	5	6	✓	✓	100,000,000+	2.28
2	com.ss.android.ugc.trill	2	1	2	✓	✓	100,000,000+	2.62
3	com.handsgo.jiakao.android	4	4	6	✓	✓	135,000,000+	1.62
4	com.fenbi.android.solar	0	4	4	✓	✓	41,350,000+	2.60
5	ccom.ganji.android	4	1	2	×	✓	40,170,000+	2.35
6	com.androidesk.livewallpaper	4	1	7	×	✓	23,990,000+	2.78
7	com.gotokeep.keep	7	0	1	×	✓	13,780,000+	1.25
8	com.naver.linewebtoon	0	2	0	✓	✓	10,000,000+	1.50
9	com.nhn.android.music	0	1	2	✓	✓	10,000,000+	1.50
10	com.yahoo.mobile.client.android.weather	2	6	3	✓	✓	10,000,000+	2.53

TABLE 4. Two types of errors in the discovered ANRs and the proportion of each of the 33 activities.

Reason	Number	Propotion
Input dispatching timed out (Waiting because no window has focus but there is a focused application that may eventually add a window when it finishes starting up.)	22	67%
Input dispatching timed out (Waiting because the focused window has not finished processing the input events that were previously delivered to it.)	11	33%

For example, in the application *com.amazon.mShop.android.shopping*, *"ModesActivity"* calls the method *"setupViewPager"* in the subclass *"FailureLandingPagePresenter"*, which does not check whether the return value is empty when calling the method *ModesConfigProvider.getInstance (VSearchApp.getInstance().getContext()). getFLPContents()*, thereby causing a null pointer exception. The reversed analysis code is shown in Fig.5. *"ModesActivity"* has been waiting for the processing result of *"setupViewPager"*, therefore, the program is blocked, and the ANR error message appears after 5 seconds, as shown in Table 7.

Attackers may achieve the purpose of Denial-of-service attack on an application with such security threats, by sending **a small number** of erroneous response packets when the application being **executed to a specific stage**. Developers are advised to avoid such ANR by the following means. After receiving the server-side error response data multiple times, the application can directly determine and prompt users for network failure or server problems, instead of waiting for correct response data.

TABLE 5. The *"com.amazon.vsearch.modes.config.ModesConfigProvider.getFLPContents"* method uses the Gson library to parse the received exception JSON string replaced by AAHF. Since the JSON data is different from the one under normal conditions, an error is reported: *It is time to parse the INT but actually parse the OBJECT.* The exception does not cause the application to crash, merely makes the return value of the *getFLPContents()* method to be null.

- 15:51:59.780 W/System.err: com.google.gson.JsonSyntaxException: Expected an int but was object at line 1 column 2815
- at com.google.gson.internal.bind.ReflectiveTypeAdapterFactory$Adapter.read(ReflectiveTypeAdapterFactory.java:176)
- at com.google.gson.Gson.fromJson**(Gson.java:803)**
- at com.amazon.vsearch.modes.config.ModesConfigProvider.getFLPContents(ModesConfigProvider.java:90)
- at com.amazon.vsearch.ModesActivity.initFailurePresenter **(ModesActivity.java:895)**
...

TABLE 6. Due to the error in Table 5, and method *"setupViewPager"* does not check whether its return value is empty when calling *"getFLPContents()"* (see Fig.4), resulting in a null pointer exception *(java.lang.NullPointerException)*.

- 15:51:59.820 E/com.amazon.device.crashmanager.AppFileArtifactSource:CRASH HAS OCCURRED
- E/com.amazon.device.crashmanager.AppFileArtifactSource: **java.lang.NullPointerException: storage == null**
- at java.util.Arrays$ArrayList.<init>(Arrays.java:38)
- at java.util.Arrays.asList(Arrays.java:155)
- at com.amazon.vsearch.failure.failurelanding.FailureLandingPagePresenter.**setupViewPager**(FailureLandingPagePresenter.java:348)
- at com.amazon.vsearch.ModesActivity.initFailurePresenter **(ModesActivity.java:895)**
...

2) JSON DATA INTERACTION

As can be seen from Table 3, a total of 25 Java layer's crashes and 25 Native layer's crashes of 10 apps were found. After preliminary analysis, it is determined that of the crashes in the Java layer are caused by wrong JSON data

```
private void setupViewPager() {
        this.mContent = Arrays.asList(ModesConfigProvider.getInstance(VSearchApp.
getInstance().getContext()).getFLPContents());
        this.setRecyclerItems();
        this.setRecyclerLabelsForFirstPage();
        this.setRecyclerViewProperties();
    }
```

FIGURE 5. The method *setupViewPager()* does not judge whether *"getFLPContents()"* is empty and directly refers to its content.

TABLE 7. The null pointer exception in Table 6 causes the main thread *"ModesActivity"* calling the *"setupViewPager"* method to be blocked. At this point, operating the application will cause ANR.

- 15:52:04.460 I/InputDispatcher: Application is not responding: AppWindowToken{434f7578 token=Token{429f52c8 ActivityRecord 42abe738 u0 com.amazon.mShop.android.shopping/com.amazon.vsearch.ModesActivity t18}}} - Window{42ba0b10 u0 com.amazon.mShop.android.shopping/com.amazon.vsearch.ModesActivity}. - It has been 5005.8ms since event, 5002.8ms since wait started. **- Reason: Waiting because the focused window has not finished processing the input events that were previously delivered to it.**

TABLE 8. JSON data error *(""behind "62" caused Invalid int)* in format *(java.lang.NumberFormatException)* causes application *com.ganji.android* to crash.

- E/AndroidRuntime(818): FATAL EXCEPTION: main) - Process: com.ganji.android, PID: 818 **- java.lang.NumberFormatException: Invalid int: "62 '"** - at java.lang.Integer.invalidInt(Integer.java:137) - at java.lang.Integer.parse(Integer.java:374) - at org.json.JSONTokener.readObject(JSONTokener.java:385) - at com.ganji.android.controller.YunyingActivityController$1-.onResponse(TbsSdkJava:132) ...

TABLE 9. JSON data error *(The corresponding value of name "x-user-id" in the key-value pair is null and the third-party library "okhttp" does not check if this value is empty.)* causes the third-party library *"okhttp"* to crash, causing the *com.gotokeep.keep* application's crash. *(It may also cause other applications that use the okhttp library to crash.)*

FATAL EXCEPTION: OkHttp Dispatcher E/AndroidRuntime(14343): Process: com.gotokeep.keep, PID: 14343 **- java.lang.NullPointerException: value for name x-user-id == null** - at okhttp3.t$a.d(Headers.java:334) - at okhttp3.ab$a.a(Request.java:165) - at com.gotokeep.keep.data.b.f$a.intercept(RestDataSource.java:235) - at okhttp3.internal.c.g.a(RealInterceptorChain.java:147) ...

(java.lang.IndexOutOfBoundsException, java.lang.NullPointerException, java.lang.IllegalStateException, java.lang.NumberFormat Exception and java.lang.IllegalArgumentException).

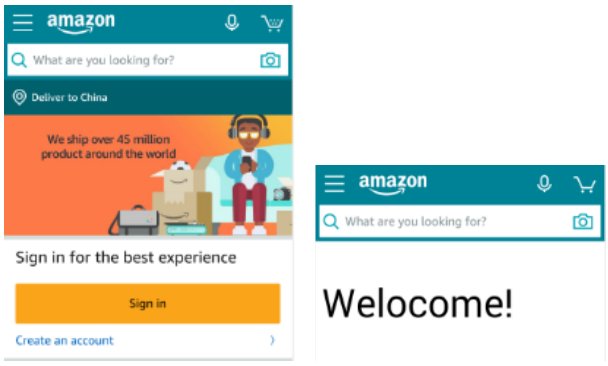
As the main test object of AAHF, JSON data occupy a large proportion in the whole test cases. If an app does not have a sophisticated checksum mechanism for JSON data, AAHF will trigger many crashes in a short period of time, such as *com.gotokeep.keep* and *com.handsgo.jiakao.android*. Fortunately, most of the current applications use popular third-party libraries (gson, etc.) to implement the processing of JSON data. These libraries are more sophisticated in handling common anomalous data (e.g., data with the wrong format), and errors caused by these data will be caught and not cause apps' crashes. In this case, if the mutated JSON data still cause a crash, it means that the mutated data have affected the execution flow of the application or triggered a security vulnerability in the third-party library, which is pretty harmful. The comparison of error log information for the two cases (an app's crash in Table 8 and a third-party library's crash in Table 9) is shown as follows.

3) HTML CONTENT REPLACEMENT

As shown in Table 3, the security risk of HTML content replacement appeared in all 10 apps, which means that the apps did not validate HTML content in HTTP/HTTPS communication data. What content they received, what content they displayed, as shown in Fig.6. In this case, if a user clicks on the replaced HTML content, it may cause security threats such as data leakage or malware download. This kind of attack has a great impact on the News applications. Attackers can change display content and spread wrong/harmful information based on this vulnerability. We reported a security vulnerability of this kind in a representative app to CNNVD. See the vulnerability CNNVD-201806-1043 [30] for details.

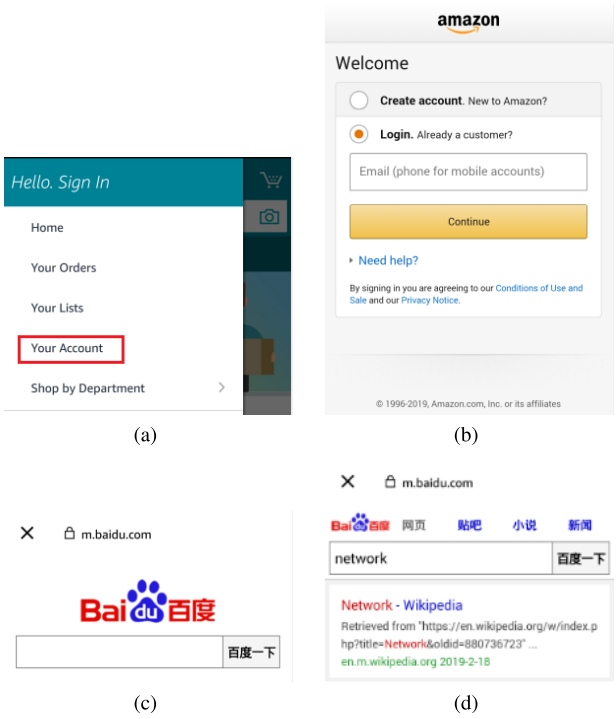
4) URL REDIRECTION

As can be seen from Table 3, 70% of apps have this problem. After replacing the "Location" field in response header with *"https://m.baidu.com"*, none of the seven apps verified this field. They directly accessed the modified URL, as shown in Fig.7. A more harmful situation is that attackers replace the redirected URL of the login interface, with a highly similar fake page. The User enters username and password on this redirected page, which results in information leakage. Additionally, access to malicious links may also lead the mobile phone to be implanted in Trojans.



(a) (b)

FIGURE 6. When the user opens the application *com.amazon.mShop.android.shopping*, the homepage interface is normally displayed as shown in (a). If the HTML content in the response data is replaced, it will display what shown in (b).



(a) (b) (c) (d)

FIGURE 7. When clicking the *"Your Account"* button in the application *com.amazon.mShop.android.shopping*, the login/registration interface shown in (b) will appear under normal circumstances. If we replace the URL with *"https://m.baidu.com"* when clicking the button in (a), the page shown in (c) will appear, and we can do something on the fake page (such as searching keyword *"network"* in (d).)

The above experimental results prove the effectiveness of the fuzzing method proposed in this paper. Developers should set up various validation rules (data format verification, redirect URL filtering and verification, etc.) when developing Android applications, to ensure the correctness of external network data. The developer can refer to the following suggestions to avoid corresponding security risks in the app.

1) *ANR:* If the server continuously returns unintended response data, the app should prompt users for network failure or server problems instead of waiting for correct response data.
2) *JSON:* If the application uses a custom JSON data processing mechanism, improve the filtering of data for various possible error formats.
3) *HTML content replacement:* This problem is mostly found in third parties, such as ad pages in apps. When jumping to a third-party page, the application should filter the content of the HTML data returned by the server before displaying it.
4) *URL redirection:* The app preferably avoids to use the response data of 302 in its key function. For the received 302 response data, the app should verify its legitimacy, filter the redirect URL, not jump except received the target URL.

V. DISCUSSION AND SUMMARY

A. LIMITATION AND FUTURE WORK

1) RESULT JUDGMENTS AND ACCURACY

AAHF cannot generate the root causes of these bugs directly. Similar to fuzzers such as IOTFUZZER [17], the design target of AAHF is to automatically produce vulnerability alerts (i.e., error message) to assist security analysts locating and confirming the root causes in applications easily. For example, analysts could search the keywords mentioned in error messages as the clue to locate the vulnerable binary code block. In fact, in general black-box testing, the final vulnerability confirmation always requires some kinds of manual efforts. AAHF might present false positive and false negative. For false positive, some ANR message reports are due to the Monkey operation (see Section IV). For false negative, since there is no reverse analysis of each test app, the test case is not targeted enough, AAHF will miss some risks.

2) ADJUSTMENTS AND IMPROVEMENTS

The JSON fuzzing framework PyJFuzz [31] used by this paper has a limitation on the data length. For the variation of long JSON data with arrays, it simply swaps the key-value pairs in sequence without changing its specific value. Therefore, the framework needs to be modified, or a new framework needs to be implemented to improve the ability to process JSON data. For data in HTML/javascript/image format, we consider increasing the diversity of samples. In the future, test samples generated by different fuzzers (AFL, PEACH, etc.) will be added to improve the testing effect. We can also perform a reverse analysis of the app to improve the relevance of test samples. How to handle HTTPS response data with custom encryption is also a problem worth studying.

3) EXPANSION OF THE APPLICABLE RANGE OF THE FUZZING METHOD

The generation of network communication data is not limited to Android applications. In the future, the applicable platform for this scheme can be extended (IOS, Windows, etc.), the types of protocols targeted can be increased, not just limiting to the HTTP/HTTPS protocol. For other protocols, we need to adjust the fuzzing strategy according to the

characteristics of the protocol data, adopt different data trigger mechanisms and exception handling mechanisms according to different platforms.

4) IMPROVE THE AVAILABILITY OF RESULTS

To exploit the vulnerabilities discovered in practice, the hacker needs to launch a successful man-in-the-the-middle attack first. This greatly increases the difficulty of utilization. In the future, it could be studied how to make more convenient use of these problems.

B. CONCLUSION

This paper proposed a black-box fuzzing method for Android applications, which uses network interaction data as test input to explore potential security risks from a new perspective. We tested 10 popular apps to verify the effectiveness of the method, discovered 4 kinds of security problems: application's ANR, application's crash caused by JSON data, HTML content replacement and URL redirection. The research indicated that network data, as one of the important attack surfaces of Android applications, would cause security impact and harm to the applications, in addition to causing the leakage of user's private data. Exploiting these discovered vulnerabilities requires a successful man-in-the-middle attack, which is difficult. Even so, when developing applications, mobile software developers should pay attention to perfecting the legality and correctness of the verification mechanism for network data, to avoid the above security risks as much as possible.

REFERENCES

[1] J. Ma, S. Liu, Y. Jiang, X. Tao, C. Xu, and J. Lu, "LESdroid: A tool for detecting exported service leaks of Android applications," in *Proc. 26th Conf. Program Comprehension*, New York, NY, USA, May 2018, pp. 244–254.

[2] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li, "Resilient decentralized Android application repackaging detection using logic bombs," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, Feb. 2018, pp. 50–61.

[3] A. Hamidreza and N. Mohammed, "Permission-based analysis of Android applications using categorization and deep learning scheme," in *Proc. Eng. Appl. Artif. Intell. Conf.*, Jan. 2019, Art. no. 05005.

[4] L. Zhang *et al.*, "Inveter: Locating insecure input validations in Android services," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Toronto, ON, Canada, Oct. 2018, pp. 1165–1178.

[5] L. Casati and A. Visconti, "The dangers of rooting: Data leakage detection in Android applications," *Mobile Inf. Syst.*, vol. 2018, Feb. 2018, Art. no. 6020461. doi: 10.1155/2018/6020461.

[6] S. Kelkar, T. Kraus, D. Morgan, J. Zhang, and R. Dai, "Analyzing HTTP-based information exfiltration of malicious Android applications," in *Proc. 17th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun./12th IEEE Int. Conf. Big Data Sci. Eng.*, New York, NY, USA, Aug. 2018, pp. 1642–1645.

[7] D. Kumar and S. Saxena, "Android application memory leakage detection approach," *Eng. J. Appl. Scopes*, vol. 2, no. 1, pp. 59–69, Feb. 2017.

[8] J. Corina *et al.*, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Dallas, TX, USA, Nov. 2017, pp. 2123–2138.

[9] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru, "Chizpurfle: A gray-box Android fuzzer for vendor service customizations," in *Proc. IEEE 28th Int. Symp. Softw. Rel. Eng.*, Oct. 2017, pp. 1–11.

[10] M. Zalewski. *American Fuzzy Lop (2.52b)*. Accessed: Dec. 23, 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[11] M. Eddington, *Peach Fuzzer*. Accessed: Dec. 24, 2018. [Online]. Available: https://www.peach.tech/products/peach-fuzzer/

[12] *UI/Application Exerciser Monkey*. Accessed: Dec. 27, 2018. [Online]. Available: https://developer.android.com/studio/test/monkey

[13] *Monkeyrunner*. Accessed: Dec. 27, 2018. [Online]. Available: https://developer.android.com/studio/test/monkeyrunner

[14] C. Zuo, Q. Zhao, and Z. Lin, "AuthScope: Towards automatic discovery of vulnerable authorizations in Online services," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Dallas, TX, USA, Nov. 2017, pp. 799–813.

[15] C. Zuo, W. Wubing, L. Zhiqiang, and W. Rui, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services," in *Proc. NDSS*, San Diego, CA, USA, Feb. 2016, pp. 1–17.

[16] Z. Zhou C. Sun, J. Lu, and F. Lv, "Research and implementation of mobile application security detection combining static and dynamic," in *Proc. 10th Int. Conf. Measuring Technol. Mechatron. Automat.*, Feb. 2018, pp. 243–247.

[17] J. Chen *et al.*, "IOTFUZZER: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. NDSS* San Diego, CA, USA, Feb. 2018, pp. 1–15.

[18] W. Xiong, S. Chen, Y. Zhang, M. Xia, and Z. Qi, "Reproducible interference-aware mobile testing," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2018, pp. 36–47.

[19] M. Xu, X. Yang, and J. Zhang, "A review of network traffic analysis targeting mobile devices," *Telecommun. Sci.*, vol. 34, no. 4, pp. 98–108, 2018.

[20] E. Vanrykel, G. Acar, M. Herrmann, and C. Diaz, "Leaky birds: Exploiting mobile application traffic for surveillance," in *Proc. 20th Int. Conf. Financial Cryptogr. Data Secur.*, Heidelberg, Germany, Feb. 2016, pp. 367–384.

[21] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Analyzing Android encrypted network traffic to identify user actions," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 1, pp. 114–125, Jan. 2016.

[22] N. Malik, J. Chandramouli, P. Suresh, K. Fairbanks, L. Watkins, and W. H. Robinson, "Using network traffic to verify mobile device forensic artifacts," in *Proc. 14th IEEE Annu. Consum. Commun. Netw. Conf.*, Las Vegas, NV, USA, Jan. 2017, pp. 114–119.

[23] H. Yao, G. Ranjan, A. Tongaonkar, Y. Liao, and Z. M. Mao, "SAMPLES: Self adaptive mining of persistent LExical snippets for classifying mobile application traffic," in *Proc. 21th Annu. Int. Conf. Mobile Comput. Netw.*, Paris, France, Sep. 2015, pp. 439–451.

[24] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "AppScanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *Proc. IEEE Eur. Symp. Secur. Privacy*, Saarbrucken, Germany, Mar. 2016, pp. 439–454.

[25] A. Continella *et al.*, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *Proc. NDSS* San Diego, CA, USA, Feb. 2017, pp. 1–15.

[26] *Telerik Fiddler*. Accessed: Dec. 29, 2018. [Online]. Available: https://www.telerik.com/fiddler

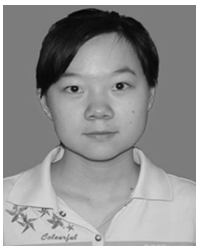
[27] S. Hocevar. *Zzuf–Multiple Purpose Fuzzer*. Accessed: Dec. 22, 2018. [Online]. Available: http://caca.zoy.org/wiki/zzuf

[28] *Dom Fuzzer Domato*. Accessed: Dec. 30, 2018. [Online]. Available: https://github.com/googleprojectzero/domato

[29] M. R. Manohar and A. Prakash, "Dealing with synchronization and timing variability in the playback of interactive session recordings," in *Proc. ACM Multimedia*, pp. 45–56, Nov. 1995.

[30] *CNNVD-201806-1043*. Accessed: Dec. 5, 2018. [Online]. Available: http://www.cnnvd.org.cn/web/xxk/ldxqById.tag?CNNVD=CNNVD-201806-1043

[31] D. Linguaglossa. (2017). *Pyjfuzz*. [Online]. Available: https://github.com/mseclab/PyJFuzz



XINYUE HUANG received the B.Eng. degree from the College of Electronics and Information Engineering, Sichuan University, Chengdu, China, in 2017, where she is currently pursuing the master's degree with the College of Cybersecurity. Her current research interests include vulnerability mining, malicious code analysis, network security, and mobile security.

ANMIN ZHOU received the B.Eng. degree from the Northwest Institute of Telecommunication Engineering, Xi'an, China, in 1984. He is currently a Research Fellow with the College of Cybersecurity, Sichuan University, China. His current research interests include malicious detection, network security, system security, and artificial intelligence.

PENG JIA received the B.Eng. degree from Sichuan University, Chengdu, China, in 2012, where he is currently pursuing the Ph.D. degree with the College of Electronics and Information Engineering. From 2015 to 2016, he was a Visiting Student with the School of Computer Science, University of Ottawa, Ottawa, ON, Canada. His current research interests include binary security, network science, and malware propagation.

LUPING LIU received the M.S. degree from Sichuan University, Chengdu, China, in 2015, where he is currently pursuing the Ph.D. degree with the College of Electronics and Information Engineering. His current research interests include binary security, information extraction, and artificial intelligence.

LIANG LIU received the M.S. degree from Sichuan University, Chengdu, China, in 2010. He is currently an Assistant Professor with the College of Cybersecurity, Sichuan University. His current research interests include malicious detection, network security, system security, and artificial intelligence.