DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag

Hui Ye[1], Shaoyin Cheng[1], Lanbo Zhang[2], Fan Jiang[1]
[1]University of Science and Technology of China, Hefei, 230027, P.R.China
[2]University of California, Santa Cruz, CA 95064, USA
sycheng@ustc.edu.cn

ABSTRACT
The Android system is getting more and more popular on the mobile devices. Thus, lots of apps have sprung up to facilitate people's daily life. However, many of the apps are released without sufficient testing work, so the users encounter a sudden app crash now and then. This will undoubtedly impact the user's experience and even lead to economic loss. Because current testing tools on Android apps mainly focus on the motion event on the screen, like click event, bugs concerned with data handling module in an app is neglected. In this paper, we propose an automated testing method to fuzz testing the Android apps. The test targets are the Activities which accept outside MIME data. These Activities are picked out by analyzing the Intent-filter tag in the AndroidManifest.xml file. An automated fuzzing tool, DroidFuzzer, is implemented based on the method. Finally, experiments are conducted to prove the effectiveness of it.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging – *monitors, testing tools*

General Terms
Design, Experimentation

Keywords
Android app, Activity, data input, fuzzing, automated testing tool

1. INTRODUCTION
Android has been one of the most popular operating systems since Google released its first version in 2008. According to the latest survey, the share of Android among the smart phone operating systems is more than 70 percent [1] in the five major Europe markets and 51 percent [2] in the United States. As the main Android Apps market, Google Play currently possesses around 700,000 apps and hits 25 billion downloads in just four years. These apps are diverse in kind, covering all aspects of social life, such as business handling, gaming, video playing, and so on. Apps have become an integral part of people's modern daily life.

Since many people spend a lot of time on apps, a number of developer groups start pouring into Android apps development, which leads to an explosion of various apps. This inevitably gives rise to a problem: the apps' homogeneity in function and declination in quality. Take the video-playing apps as an example, there are more than 50 different video-playing apps in Google Play and most of them can decode and play common video formats. However, the robustness of these apps differs a lot from each other. Users often encounter the situation that their video players crash without a sign when they are watching movies. This sometimes is caused by underlying hardware errors. However, since large-scale devices testing has already been applied to an app before release, the most important reason why crash happens is that the video-playing apps on their smart phones contain vulnerable codes which throw exceptions in runtime.

According to a survey [3] by Compuware Corporation, 56 percent of respondents encountered a problem (app crashes, freezes, errors, or either slow to load or won't load) while using their smart phones. Among those who have experienced a problem, 62 percent reported a crash, freeze or error; and 40 percent reported an app that even would not launch. Undoubtedly, frequent crashes of an app will lead to very poor user experiences and drive users to uninstall it. Another problem brought by app bugs is the possibility of economic loss since many smart phone apps deal with payment or important business data. Consequently, finding out the vulnerable code in an app and fixing it in time are very important.

It is well known that the necessary procedure to reduce the chance of bugs in code is adequate and rigorous testing during the application development. However, fierce competitions between companies have pushed developers to release new applications as soon as possible. Therefore, continuing to test the app after release and rolling out new versions to fix bugs is one of the most effective and practical methods. That is to say, an effective tool that can find hidden software bugs will be a good assist to developers.

On Android smart phones, most apps involve with some kinds of outside input, which can be divided into two categories: motion input and data input. Motion input includes user actions like screen touch, sliding and key inputs. Data input includes all the other data needed by the app in the form of files, character strings, and network data streams. For example, for video-playing apps, movie files are data input and screen touch to play the movie is motion input. Compared with motion inputs, data inputs are usually more complex in formats, thus are the main cause of app bugs.

In this paper, we present a method for testing Android apps with data input, and based on the method, we introduce a fuzz testing tool called DroidFuzzer.

MoMM2013, 2-4 December, 2013, Vienna, Austria.

DroidFuzzer is a tool that tests Android apps and detects crashes during runtime. We have used this tool to test several apps and found some bugs. Since DroidFuzzer works without the source code, it can be used for black-box testing of apps. Another usage of DroidFuzzer is to assist security researchers to discover potential vulnerabilities of Android apps.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 presents the design of DroidFuzzer. Section 4 includes the evaluation and analysis of the tool. Finally, Section 5 is the conclusion of our work.

2. RELATED WORK

Apps test is an essential step during their development cycles. There are several well-known tools can be used for testing Android apps. Monkey [4] is a stress-test tool that runs on Android emulators or devices. It can generate pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. Monkey can only test apps on motion input and the test cases are random. MonkeyRunner [5] provides an API for writing scripts to control devices and testing progress. It is often used for function testing and regression testing. However, test cases provided by MonkeyRunner are also motion input. DroidPilot [6] can capture screen objects via emulators or devices. Developers then use these objects to build up the testing scripts and apply to hundreds of devices at one time. Since DroidPilot provides the convenience for getting UI objects, it is good at UI element testing. iTestin [7] can record testing scripts on one device, and then run the scripts on different devices. It is used for large-scale device testing and regression testing. Robotium [8], which is based on the Android Instrumentation class, is a test framework what makes it easy to write powerful and robust automatic UI test cases for Android apps. With the support of Robotium, test case developers can write function, system and acceptance test scenarios, spanning multiple Android Activities. Robotium can be used for both white-box and black-box testing. Overall, the common feature of DroidPilot, iTestin and Robotium is that they mainly focus on UI element testing and performance testing. In addition to these tools, Dynodroid [9] is a system for generating relevant event inputs to Android apps. By instrumenting the framework once for all, the system monitors the reaction of an app upon each event in a lightweight manner, using it to guide the generation of the next event to the app. With Dynodroid, the author tested several open-source Android apps, and discovered totally 15 bugs.

Fuzz testing [10], or fuzzing, is an effective software testing technique, which is commonly used for testing security problems of software and operating systems. Generally, it makes use of invalid, unexpected, or random data as the input of the target. On operating systems beyond Android, researchers have successfully discovered lots of bugs and vulnerabilities with fuzz testing tools. In 1990, Miller [11] tested UNIX system with the method of simple fuzz testing and found that more than 25 percent of the random test cases could lead to system crashes. In 2002, Aitel [12] found multiple bugs and vulnerabilities on UNIX, using a fuzz testing tool called SPIKE. In 2008, Godefroid [13, 14] applied the fuzz testing method on large-scale windows apps, and found dozens of 0-day vulnerabilities.

As to fuzzing on Android, the research has just started. In 2009, Mulliner [15] tried to construct abnormal input data to test the SMS module on Android system, and detected several exceptions at runtime. In 2012, the same method was used to test the NFC (Near Field Communication) module [16] on Android. Another method to fuzzing the NFC module was proposed by Stirparo [17], who tried to vary the exchange message between two NFC devices according to the NFC data exchange format and then monitor the state of the NFC module. But until now, the development of the solution has not been completed. Maji [18] tried to fuzzing the IPC mechanism in the Android system. By constructing abnormal Intent objects and sending them to Android components, some significant results were found. However, the tool JarJarBinks, named by Maji, just focused on the inner attributes in the Intent object and ignored the outside data referenced by the URI section in the Intent object. So, the bugs concerned with outside data handling module in the components were neglected. Besides, three students from Singapore Polytechnic designed and developed the Android Kernel Fuzzer [19], aimed to discover bugs on the LINUX kernel layer of Android, but did not get interesting results. There is another open-source fuzz testing tool named Intent Fuzzer [20] for Android apps. Developed by iSEC partners, this tool can fuzz test the components of all installed apps. However, the only input test data is NULL, which weakens the ability to discover bugs in apps.

3. DESIGN

Android apps typically consist of four components: Activity, Service, Broadcast Receiver and Content Provider. Among these four components, Activity is the main interface on which users can interact with apps. User input, including motion input and data input, is firstly received and handled in Activity. On Android, an Intent object provides the communication bridge between Activities. Outside data is often packed or referred in an Intent object. So we can fuzz the data input, construct a specific Intent object to pack or refer the data, and then send the object to the target Activity for testing. In this way, we propose a fuzz testing method to find bugs in Android apps and a new tool is developed named DroidFuzzer, as shown in Figure 1. Different from many existing testing tools which are focused on motion input, our method in this paper is focused on data input.

In general, DroidFuzzer tries to fuzz testing the target apps on their outside data input. It extracts Activity information from the target APK (short for Application Package File) and gets the detail information about the data input that the app can accept. Then it chooses corresponding seed data and the seed data is varied to generate abnormal data. Finally, the abnormal data test case is sent to the app for testing and the running process is monitored to detect crashes.

DroidFuzzer is made up of three parts: pretreatment module, variation module and dynamic detection module. The input of this tool is the target APK. With the Activity information extracted from the AndroidManifest.xml file of the target APK in the pretreatment module, the variation module chooses the right seed data to generate abnormal data. Then the dynamic detection module uses the abnormal data to test the Activity and monitors the state of the process. Crash logs are captured if the target process encounters exceptions. Then the crash logs will be analyzed manually for discovering bugs and potential vulnerabilities in the APK, and detail information is reserved for future research.

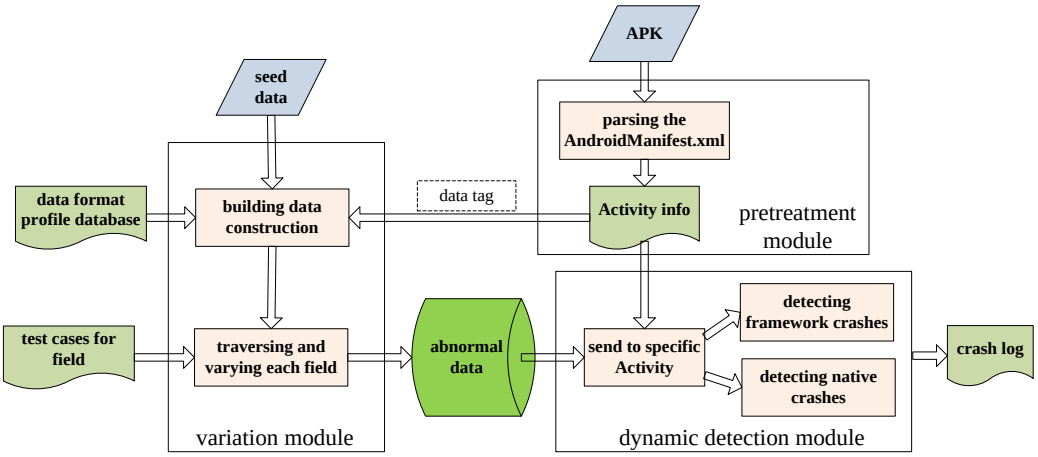

Figure 1. The architecture of DroidFuzzer

3.1 Pretreatment Module

DroidFuzzer aims at fuzz testing Activity components in the target app. However, not all the Activities accept outside data input. So the first step is to pick out the Activities that could be tested. This work is done by analyzing the AndroidManifest.xml file in the APK.

According to the development documentation of Android, each of the Activities that appear in an app should have a definition entry in the AndroidManifest.xml file. The format of the definition entry is shown in List 1.

List 1. Definition entry of an Activity

```
<activity>
    <intent-filter>
        <action/>
        <category/>
        <data/>
    </intent-filter>
    <meta-data/>
</activity>
```

In the definition entry, detailed information like Activity name, launch mode and other attributes are specified. Particularly, information about what kind of data can be accepted is defined in the tag 'intent-filter'.

There are three subtags in 'intent-filter': 'action', 'category', and 'data'. 'action' defines the action to be performed when the Activity is launched . 'category' contains additional information about the component that should handle the intent. The most important subtag is 'data', which is used to specify the URI and the type of the data input.

The subtag 'data' consists of two parts: URI and MIME data type. URI is in the form of "scheme://host:port/path". MIME specifies the media type of the data input. Based on the URI and MIME, we can determine the data type that the target Activity accepts.

In summary, by parsing the AndroidManifest.xml file, two types of information are extracted: the target Activity name and the data type the Activity can handle. The target Activity name identifies the Activity that will be tested and be monitored in the dynamic detection module. The data type determines the type of seed data in the variation module.

There are many types of MIME, and the common ones include: "video/*", "audio/*", "text/*", "image/*", "application/*", etc. All of these MIME types can be further categorized into more specific types. For example, "video/*.avi" is included in "video/*" and "text/html" is included in "text/*". These MIME data types have specifications that are publically available. Thus we can construct normal data input following the specifications and make some changes to the original data to generate abnormal data. This is the basic idea of the variation module, which will be introduced in the following section.

3.2 Variation Module

The variation module is the core of our design. It takes normal seed data as input and generates abnormal data as test cases. In this module, three problems need to be solved.

The first problem is how to provide different types of normal data as input. The second is how to construct the normal data input. The last problem is how to generate abnormal data.

To solve the first problem, seed data is prepared for each MIME data type. The seed data is constructed by strictly following the MIME data type specification. For example, for "video/*.avi", the seed data is a normal AVI file that can be accepted by all regular video-playing apps. Since there are many MIME types, the seed data collection should be as much complete as possible.

To solve the second problem, we use a data format profile database for seed data construction. The data format profile database contains the data format profiles of most existing MIME types including AVI, MP3, HTML, BMP, etc. These data format profiles are designed by extracting the construction of the specifications of each MIME data type. They are summarized manually.

A MIME data type specification defines the construction of the data type. It usually breaks an integral stream of data into many chunks, each of which consists of different basic elements called fields. A field, often in size of several bytes, stores different kinds of values. An example of data format profile is shown in List 2.

Based on the data format profile, the seed data can be constructed into many chunks, or in more detail, into many fields. This helps us to vary each field to generate abnormal data.

List 2. An example of data format profile

```
<Data_Format_Profile>
  <Chunk>
    <ChunkName> chunk name </ChunkName>
    <ChunkType> chunk type </ChunkType>
    <FieldCount> number of fields </FieldCount>
    <FieldSizeArray> array of field size </FieldSizeArray>
    <FieldTypeArray> array of field type </FieldTypeArray>
    ...
    <Tables>
      <Table>
        <ItemCount> number of items </ItemCount>
        ...
      </Table>
    </Tables>
  </Chunk>
  <Chunk>...</Chunk>
  ...
</Data_Format_Profile>
```

To solve the third problem of varying strategy, variation cases for different kinds of fields are designed. In this paper, we consider four main field categories: Integer, Flag, String and Byte Array.

Integer fields are used to store numeric values like the size of the String data, ID number, version number, timestamp and so on. The size of an Integer field is often a multiple of 8 bits. The boundary values of the Integer field, together with the values around the boundary values are used as variation cases. For example, the variation cases for a 16-bit Integer field include: 0, ±1, $\pm(2^{\wedge7}-1)$, $\pm2^{\wedge7}$, $\pm(2^{\wedge7}+1)$, $\pm(2^{\wedge15}-1)$, $\pm2^{\wedge15}$, $\pm(2^{\wedge15}+1)$.

Flag fields are a special kind of Integer fields whose values are either very small, ranging from 0 to 10, or varied in one bit, like "0001" and "0010". Flag fields are often used to store enumeration values. The variation cases of Flag fields also contain the boundary values. Besides, a bit variation method is used to traverse every bit of the Flag value.

String fields store character strings and their sizes are variable. Generally, String values are extremely vulnerable to software flaws. Many serious software vulnerabilities are involved with special string formats. So variation on strings should focus on special characters like quotation marks. In this paper, variation cases of string fields are designed to include special characters.

Sometimes, an array of bytes is needed to store special values. This kind of arrays is called Byte Arrays. Byte Array values may be resolved as a unique ID or a string. So the variation on Byte Arrays is the same as the variation on strings.

Following the above variation methods, basic variation cases for each field can be designed. Then, we use these variation cases to replace normal field values to generate abnormal data as test cases.

3.3 Dynamic Detection Module

With abnormal data test cases available, the next step is to use them as Activity input to detect bugs of the target app. An abnormal data input probably triggers a program exception. If the exception is not caught and handled properly, the program will crash. In the dynamic detection module, we discover bugs by detecting program crashes. On Android, app crashes may happen in two layers: the Dalvik layer and the native code layer. So we use two different methods to detect app crashes on both Dalvik layer and native code layer.

In the Dalvik layer, codes are written in Java and run in the form of Dalvik Byte Code. By overwriting the ActivityController class, it is convenient to detect Activity crashes and store the backtrace information.

To improve the performance of complex calculation and raise the reuse rate of current code, many developers prefer to code and compile their core algorithm into ".so" lib file and then pack it into apps. The code in ".so" file is named native code and called through the mechanism of JNI. Because the native code runs on the layer together with the libraries of Android system, detecting native crashes in real time is difficult. In the dynamic detection module, the tombstone files are monitored to detect native crashes. A tombstone is a file containing important information about a crashed process. Every time a process crashes on Android, a tombstone is written for the process. By monitoring changes in tombstone files, native crashes can be detected.

3.4 Implement the Automated Tool

According to the design of the three above modules, we implement an automated fuzz testing tool called DroidFuzzer. When the tool starts, the pretreatment module unpacks the target app and the inner AndroidManifest.xml file is analyzed to get the detail information of Activities. Then the seed data that the target app can handle is picked out as the input of the variation module. In the variation module, each field of the seed data is varied to generate abnormal data iteratively. Every time a piece of abnormal data is generated, the variation module sends it to the target app and then turns into sleeping state for a while, waiting for the target app finishing handling the data. The dynamic detection module will monitor the process of the target app and store all the available information if a crash happens.

DroidFuzzer is an automated fuzz testing tool aimed at the outside data input of the target apps. It just takes the target app as input and generates crash logs as test results. By extending the data format profile database in the variation module, DroidFuzzer can support most of the outside input data types. Besides, it is implemented as an app on Android, which makes it convenient to use for program developers.

4. EVALUATION

Experiments are conducted to evaluate the effectiveness of the tool on the Android emulator of Version 4.0.3 and the results are verified on a real device. Several bugs have been found and causes of some of the bugs were analyzed.

4.1 Experiment Sample Set Generation

There are many MIME data types and each type has a corresponding seed data. In this paper, we choose two popular MIME data types (video and audio) to generate abnormal data as test cases. The goal of the experiment is to evaluate the capabilities of our tool on discovering bugs in apps. In our experiment, a bug in an app is defined as an error that either:

- causes the app to crash and exit
- causes the system to display an ANR
- causes the process to unlimitedly consume the resources of system

Table 1. Test results of video data

ID	Target APP	Seed Data	Modified Field (Chunk Name – Field Name)	Field Type	Bug
1	MX Palyer	MP4	stts (video configuration) – number of entries	Integer	ANR
2	Mobo Palyer	MP4	stts (video configuration) – number of entries	Integer	Consumption of Resources
3	Mobo Palyer	MP4	stts (audio configuration) – number of entries	Integer	ANR
4	QQ Player	AVI	strh (video configuration) – dwRate	Integer	Crash and Exit
5	QQ Player	AVI	strh (audio configuration) – fccType	String	Crash and Exit
6	QQ Player	MP4	esds – esds tag	Integer	Crash and Exit
7	QQ Player	MP4	esds – esds tag length	Integer	Crash and Exit
8	QQ Player	MP4	esds – stream priority	Integer	Crash and Exit
9	QQ Player	MP4	esds – DecoderConfigurationDescription tag	Integer	Crash and Exit
10	QQ Player	MP4	esds – decoder specific description tag	Integer	Crash and Exit
11	QQ Player	MP4	stsz – Sample size	Integer	Crash and Exit

In the second definition item, ANR means "Application Not Responding", which will cause the system display a popup window to prompt the user to kill the process or wait for the process.

4.2 Video Data Test

Three well-known video-playing apps (MX Player, Mobo Player, and QQ Player) are tested using our tool on three video data formats (AVI, MP4, and RM). Among the three apps, the market share of MX Player (version 1.7.14) is the largest in the globe and Mobo Player (version 1.3.240) holds the second place. QQ Player (version 2.1.377) is the most well-known video-playing app in China. Since these three apps are developed by relatively large companies with well-trained apps developers, they are expected to be more robust than most of the other video-playing apps. The test results of the three apps are shown in the Table 1.

In Table 1, we find that MX Player and Mobo Player are more robust than QQ Player, with only one and two bugs found respectively. As for data format, no bug is found when decoding RM, and more bugs are found when decoding MP4 than decoding AVI. This is probably because MP4 is the most complex data format among the three formats, followed by AVI and then RM. So the complexity of MP4 format brings a bigger challenge for developers to design a logical and fully functional decoder.

We analyze the causes of some of the bugs in Table 1 using the method of instrumentation. Take the bug with ID number 5 in Table 1 as an example, if the "*fccType"* field of the "*strh*" chunk in AVI seed data is filled with an abnormal String value, the QQ Player app will crash and exit suddenly. The crash information can be captured from the LogCat window of Eclipse, as shown in List 3.

According to List 3, we can see it is an out-of-memory error and the error occurs in the native code of Android when QQ Player tries to create a bitmap object. Since Android is an open source system, we can view the source code of the function "*createBitmap"* in class "*android.graphics.Bitmap"*. This function needs the width and height of the bitmap as parameters, and these two parameters are passed from function "SetVideoSize()" in the source code of QQ Player. By instrumenting log print sentences into the "SetVideoSize()" function, we can get the width and height values in both normal case and abnormal case.

As shown in List 4, when decoding normal seed data, the width and height of the bitmap are 352 and 288 respectively. However, the two parameters become 44100 and 16000 when decoding abnormal data. Since the two parameters determine the buffer size in the heap memory, the abnormally large values will lead to the out-of-memory exception. Now, we calculate the buffer size that the player needs to decode abnormal seed data. Firstly, the buffer size of the bitmap object is calculated as:

$$Size = Width * Height * PixelSize$$

Because each pixel is stored in 2 bytes and the heap head is 16 bytes, the size of the heap memory that QQ Player requests is 1411200016 bytes, i.e. nearly 1.4 Gigabytes. Obviously, this will cause an out-of-memory exception. Unfortunately, QQ Player failed to capture this exception, so the process is killed by the system.

List 3. Crash information of the bug with ID number 5 in Table 1

```
Out of memory on a 1411200016-byte allocation.
"main" prio=5 tid=1 RUNNABLE
| group="main" sCount=0 dsCount=0 obj=0x409c1460 self=0x12810
| sysTid=670 nice=0 sched=0/0 cgrp=default handle=1074082952
| schedstat=( 4113927421 3360925598 1514 ) utm=360 stm=51 core=0
at android.graphics.Bitmap.nativeCreate(Native Method)
at android.graphics.Bitmap.createBitmap(Bitmap.java:605)
at android.graphics.Bitmap.createBitmap(Bitmap.java:585)
at com.tencent.research.drop.PortAndroid.AndroidNativePainterHelper.SetVideoSize((null):-1)
at com.tencent.research.drop.PlayingControllerWarp.TmpOpenId(Native Method)
```

List 4. The width/height values of normal and abnormal seed data

```
//QQPlayer
//Package: com.tencent.research.drop
//parameters of normal seed data
D/param-width(822): 352
D/param-height(822): 288
…
//parameters of abnormal seed data
D/param-width(822): 44100
D/param-height(822): 16000
```

Another example we will analyze is the bug with ID number 2 in Table 1, which is a resource-consuming bug. This bug is triggered when the Mobo Player broadcasts an abnormal MP4 seed data. In that abnormal seed data, the Integer field 'number of entries' of chunk 'stts' is modified to be the value of 0x80000000. Because the 'stts' chunk stores the time-to-sample map table, which is used to get the right sample data to broadcast, the error in parsing the table may lead to a broadcast exception. In our test, if this bug is triggered, the main process of Mobo player will lose control of the broadcast thread, and the thread will be stuck in an infinite loop.

From the timeline log shown in List 5, we can see the app continuously prints the error information infinitely. Unless users kill the Mobo Player service by hand, the uncontrolled erroneous broadcast thread will never stop running. Moreover, if Mobo Player tries to broadcast the abnormal seed data again and again, more and more uncontrolled threads will be started, leading to the exhaustion of system resources. By decompiling the source code of the Mobo Player APK, we locate this bug in the "SystemPlayer" class.

List 5. Error information of the resource-consuming bug

```
10-15 07:18:06.939: E/SystemPlayer(661): 0 >=0
10-15 07:18:07.967: E/SystemPlayer(661): 0 >=0
10-15 07:18:08.988: E/SystemPlayer(661): 0 >=0
10-15 07:18:10.053: E/SystemPlayer(661): 0 >=0
10-15 07:18:11.080: E/SystemPlayer(661): 0 >=0
10-15 07:18:12.122: E/SystemPlayer(661): 0 >=0
10-15 07:18:13.150: E/SystemPlayer(661): 0 >=0
10-15 07:18:14.193: E/SystemPlayer(661): 0 >=0
10-15 07:18:15.217: E/SystemPlayer(661): 0 >=0
```

4.3 Audio Data Test

Audio Data is another common MIME data type that many apps take as input, such as music players, audio edit apps, music alarms and so on. To conduct the experiment on audio data, we choose MP3 and WMA, two most popular Audio data formats on PC platform, as the test case sources. On the selection of target apps, we pick TTPod (version 5.6) and Baidu Music Player (version 3.0). Those two players are among the most famous music player apps in China. The test results are shown in Table 2.

As shown in Table 2, three bugs are detected. Two bugs are concerned with TTPod, the other is concerned with Baidu Music Player. From the view of audio data types, no bug related with MP3 data is detected. We infer that the possible reason may lie in the complexity of the data format. Since MP3 is a simple data format with only three main chunks, it is easier for developers to develop a full functional decoder. At the same time, WMA has a big and multilevel nested header, which poses a challenge to the developers when they design the decoder.

The first bug in TTPod is an ANR error. When the app receives abnormal data from outside, the phone screen turns black and no touch event will be responded to. Even the app is restarted, the screen remains black. Unless the app is reinstalled, users can not use the app any more.

The second bug in TTPod will lead to system overload, mainly CPU overload. When handling abnormal data, the system response slows down sharply. To make things even worse, the main process of the app does nothing to stop the endless resource consumption, but stay in the situation even after the app is restarted. We recorded CPU utilizations every 20 seconds for both normal and abnormal, and shown them in Figure 2 and Figure 3.

In the two figures, we can see the TTPod has a main process named "*com.sds.android.ttpod*" and a service thread named "*com.sds.android.process.ttpod.support*". The work of decoding and broadcasting input data is done by the service thread. Under general situation, the CPU utilization is nearly 50 percent when normal audio data is broadcasted. However, when abnormal data is send to TTPod, the CPU utilization rises sharply to nearly 96 percent.

The third bug is a divide-by-zero exception. If this bug is triggered, the Baidu Palyer app will crash and exit immediately.

4.4 Other Data Test

Some extra tests are done with other MIME data types to show the utility of the tool. On testing browsers, we validated the browser cross-application scripting vulnerability of CVE-2011-2357.

The vulnerability is caused by the flaw in WebView class, which is the common underlying engine of most browser apps. If the WebView instance has already loaded a URL, and the same instance is used to load an abormal URL, like "javascript:code", then the javascript code is executed in the domain of the loaded URL. This helps third party apps to bypass the sandbox in which the browser runs, since it allows apps to inject scripts into loaded pages, and control the WebView. In the validation test, we construct the URL data to be "javascript:alert(document.cookie)", and send it to the QQ Browser and the UC Browser (two most popular browser apps in China). Both of the two browsers execute the code and show the cookie of the current webpage.

Table 2. Test results of audio data

ID	Target APP	Seed Data	Modified Field (Chunk Name – Field Name)	Field Type	Bug
1	TTPod	WMA	File Properties Object - Object Size	Integer	ANR
2	TTPod	WMA	File Properties Object – Minimum Data Packet Size	Integer	Consumption of Resources
3	Baidu	WMA	File Properties Object - ID	Byte Array	Crash and Exit

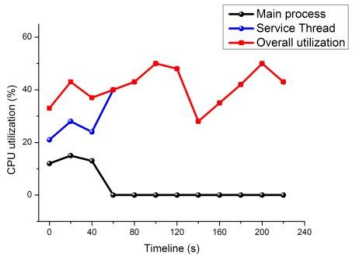

Figure 2. CPU utilization with normal data

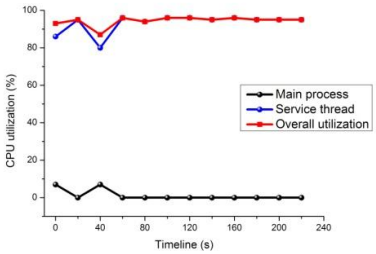

Figure 3. CPU utilization with abnormal data

5. CONCLUSION

In this paper, we propose a fuzz testing method to do the black box test of Android apps and implement an automated tool called DroidFuzzer. The core of the tool is the variation module which takes the normal data as the input and generates abnormal data as test cases. Then the test cases are sent to the target app and the process is monitored for bug information. Typical MIME data types are used to generate test cases, and several well-known apps are chosen to validate the effectiveness of the tool. As a result, a total number of 14 bugs are found and a vulnerability is verified. Because the tool is implemented as an app and the overhead is low, it will assist a lot in discovering app bugs and latent vulnerabilities.

6. ACKNOWLEDGMENTS

This research was supported in part by the Fundamental Research Funds for the Central Universities of China (WK2101020004, WK0110000007), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20113402120026), the Natural Science Foundation of Anhui Province, China (1208085QF112) and the Foundation for Young Talents in College of Anhui Province, China (2012SQRL001ZD).

7. REFERENCES

[1] Survery : Smartphone sales data – almost half of all smartphones sold in Europe are a Sumsung. DOI= http://www.kantar.com/media/mobile/010713-smartphone-sales-data-uk-and-europe/.

[2] Survery : US smartPhone data - windows growth fueled by share gains among 25 to 34 years old. DOI= http://www.kantar.com/media/mobile/030613-smartphone-sales-data-us/.

[3] Survery : Mobile Apps : What consumers really need and want. DOI= http://offers2.compuware.com/rs/compuware/images/Mobile_App_Survey_Report.pdf.

[4] Monkey. DOI= http://developer.android.com/tools/help/monkey.html.

[5] MonkeyRunner. DOI= http://developer.android.com/tools/help/monkey.html.

[6] DroidPilot. DOI= http://www.droidpilot.com/zh_tw.aspx.

[7] iTestin. DOI= http://www.testin.cn/portal.action?op=Portal.iTestin.

[8] Robotium. DOI= https://code.google.com/p/robotium/.

[9] MacHiry, A., Tahiliani, R., Naik, M. 2013. Dynodroid: An Input Generation System for Android Apps. *In Proceedings of 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Saint Petersburg, Russia, August 18-26, 2013). ESEC/FSE 2013. ACM, New York, NY, 224-234. DOI= http://dl.acm.org/citation.cfm?id=2491450.

[10] Miller, B. P., Koski, D., Lee, C.P., Maganty, V., Murthy, R., Natarajan, A., Steidl, J. 1995. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. *J. Computer Sciences Technical Report.* Madison, WI 53706-168.

[11] Miller, B. P., Fredrikson, L., So, B. 1990. An empirical study of the reliablity of UNIX utilities. *J. Communications of the ACM*. CACM, New York, NY, (Dec. 1990), 32 - 44. DOI= http://dl.acm.org/citation.cfm?doid=96267.96279.

[12] Aitel, D. 2002. The advantages of block-based protocol analysis for security testing. New York: Immunity Inc.

[13] Godefroid, P., Levin, M. Y., Molnar, D. A. 2008. Active property checking. *In Proceeding of the 8th ACM International Conference on Embedeling oftware* (Atlanta, GA, October 19 - 24, 2008). EMSOFT '08, ACM, New York, NY, 207-216. DOI= http://dl.acm.org/citation.cfm?doid=1450058.1450087.

[14] Godefroid, P., Levin, M. Y., Molnar, D. A.2008. Automated whitebox fuzz testing. *In Proceeding of Network Distributed Security Symposium* (San Diego, CA, February 8 – 11, 2008).

[15] Mulliner, C. 2009. Fuzzing the Phone in your Phone. 26th Chaos Communication Congress (26c3) Berlin, Germany (December, 2009).

[16] Mulliner, C. 2012. Dynamic Binary Instrumentation on Android. RuxCon 2012, Melbourne, Australia.

[17] Stirparo, P. 2013. A Fuzzing Framework for the Security Evaluation of NDEF Message Format. *In Proceedings of 5th International Conference on Computational Intelligence, Communication Systems, and Networks* (Madrid, Spain, June 5-7, 2013).

[18] Maji, A. K., Arshad, F. A., Bagchi, S., Rellermeyer, J. S. 2012. An Empirical Study of the Robustness of Inter-component Communication in Android. *In Proceedings of the International Conference on Dependable Systems and Networks* (Boston, MA, June 25 - June 28, 2012). DSN, page 1-12. IEEE Computer Society, (2012)

[19] Android Kernel Fuzzer. DOI= http://androidfuzzing.com/.

[20] Intent Fuzzer. DOI= https://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx.