Intelligent Automation & Soft Computing

Tech Science Press

DOI: 10.32604/iasc.2024.047509

ARTICLE

BArcherFuzzer: An Android System Services Fuzzier via Transaction Dependencies of BpBinder

Jiawei Qin[1,2], Hua Zhang[1,*], Hanbing Yan[2], Tian Zhu[2], Song Hu[1] and Dingyu Yan[2]

[1]The State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, 100080, China

[2]The National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, 100012, China

*Corresponding Author: Hua Zhang. Email: zhanghua_288@bupt.edu.cn

Received: 07 November 2023 Accepted: 14 March 2024 Published: 11 July 2024

ABSTRACT

By the analysis of vulnerabilities of Android native system services, we find that some vulnerabilities are caused by inconsistent data transmission and inconsistent data processing logic between client and server. The existing research cannot find the above two types of vulnerabilities and the test cases of them face the problem of low coverage. In this paper, we propose an extraction method of test cases based on the native system services of the client and design a case construction method that supports multi-parameter mutation based on genetic algorithm and priority strategy. Based on the above method, we implement a detection tool-BArcherFuzzer to detect vulnerabilities of Android native system services. The experiment results show that BArcherFuzzer found four vulnerabilities of hundreds of exception messages, all of them were confirmed by Google and one was assigned a Common Vulnerabilities and Exposures (CVE) number (CVE-2020-0363).

KEYWORDS

Android OS; vulnerability detection; binder; fuzz testing; genetic algorithm

1 Introduction

The highest proportion of mobile operating systems is still Android [1], each version of the Android system is open source (AOSP). Many mobile phone manufacturers customize their own Android operating systems based on AOSP. The Android system provides many practical functions, such as taking photos, and playing multimedia. These functions are provided by Android system services. However, system services are also the target favored by attackers. Once the system service is attacked, it can directly cause crashes of the Android system or even more serious problems. Although Android systems are updated every year, a number of vulnerabilities have been reported [2]. Many of them are Android native system services vulnerabilities.

In 2016, Feng et al. [3] captured the input model of target services by recording the requests of 30 popular Android applications. This method cannot restore the precise data types of the parameters passed by service interfaces, such as variable names and types. In addition, since the acquisition of interfaces and data types completely depends on the triggered functions during the use of Android

applications, it is difficult to obtain all service interfaces. Next year, Iannillo et al. [4] proposed a tool, Chizpurfle, which uses Java reflection methods to obtain the parameter types of interfaces to do fuzzing for the custom Java services in the non-AOSP systems. In 2020, Liu et al. [5] proposed a server-based method to extract interfaces and detect vulnerabilities in Android native system services. This method directly analyzes source codes of the server of native system services. Based on the analysis data, the method extracts data types of interfaces for native system services to detect vulnerabilities. However, the test case-generating method is based on random generation, and the test coverage is low. The method of initializing the data types of Binder servers also misses some data types defined by the client interface, which reduces the number of discovered vulnerabilities.

In the vulnerability analysis of the native system services, we found that there are vulnerabilities caused by inconsistent data transmission and inconsistent data processing logic between the client and the server. The analysis of the two types of vulnerabilities needs the knowledge of the data type transmitted by the corresponding client interfaces of native system services, but these data cannot be directly obtained from the server code analysis of native system services. For the native system service vulnerability analysis methods, the test coverage of the methods of generating test cases for native system service vulnerability detection methods is low. To detect the above two vulnerabilities in native system services, the challenges we face are as follows: (1) How to extract more interfaces of native system services and construct the response between client and server? (2) How to accurately analyze the detailed transmission data type of interfaces and construct vulnerability test cases? (3) How to solve the mutation problem of multiple parameters in the test cases?

In this paper, we proposed an automatic fuzz testing method for the vulnerabilities caused by inconsistent data and inconsistent logic processing of Android native system services. We extract interfaces of the system service from the client source code of the system service and construct an abstract syntax tree (AST) for transmission data type corresponding to the interface. Based on the above data, we use a genetic algorithm to construct test cases for fuzz testing of native system services. The main contributions of this paper are as follows:

(1) We extract interfaces from the client source code of the Android native system service, and all data transmission types for each interface.

(2) We propose a multiple parameter mutation by genetic algorithm to generate test cases of interfaces. This method can improve the test coverage of test cases.

(3) To detect the vulnerabilities caused by inconsistent data and inconsistent logical processing of Android native system services, we propose and implement an automated fuzzer, BArcherFuzzer. Some vulnerabilities were successfully found from hundreds of abnormal crash information, these were confirmed by Google and one was assigned CVE-2020-0363 by CVE. BArcherFuzzer can work with most major Android versions.

2 Background

The design of Android Binder is the Client-Server communication mode. A process acts as a server, providing services such as media playback, audio, and video capture; Multiple App processes as clients send service requests to the server to obtain the required service functions. The binder is implemented on both the Client and Server sides. (1) For the Server, Binder can be regarded as the access point of a specific service provided by the Server to the Client, and the Client sends a request to the Server through the access point to use the service; (2) For Client, Binder can be regarded as a pipeline entrance

to Server. To communicate with a Server, the Client must first establish this pipeline and obtain the pipeline entrance.

Different from other Inter-Process Communication (IPC) methods, the Binder mechanism adopts object-oriented thinking to express the Binder as the access point and the pipeline entrance in the Client. Binder is an object in the Server, that can access the specific service method of the upper layer; The pipeline entries of different Clients can be regarded as the handle to the Binder entity object. Once the handle is obtained, the method of the object can be called to access the Server. From the perspective of communication, the Binder in the Client can also be regarded as the "agent" of the Binder object in the Server, which provides services to the Client on behalf of the remote Server locally. The Binder mechanism blurs the boundaries between processes and dilutes the communication process between processes, making the entire Android system run in the same object-oriented program.

2.1 Binder Communication

The Binder mechanism is a unique IPC method of the Android system, and it is also the basis for the establishment of Android system services.

Fig. 1 is the communication model of Binder [6], which includes four roles: Server, Client, ServiceManager and Binder driver. The relationship between these four roles is similar to the TCP/IP network structure: Server is the server; Client is the client; ServiceManager is the domain name server (DNS), and the driver is the router.

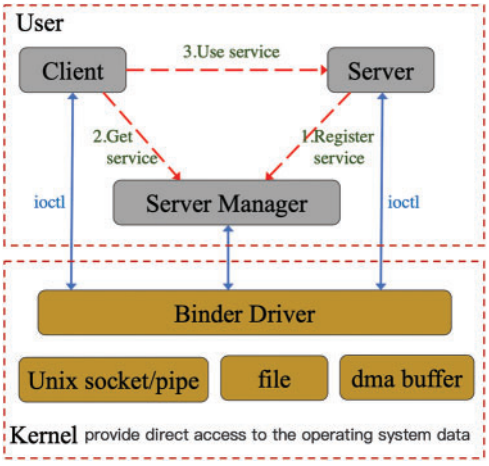

Figure 1: Binder communication model

After the Android system is started, the Server will start the internal service and create the Binder entity object corresponding to the service, then send a "registration service" request to the Service Manager. Service Manager will save a "name-reference" table in itself according to the request packet, just like the "domain name-IP" table stored in DNS. When the Client wants to use a specific service function, the Client will send a "Get Service" request to the Service Manager and query the "Name-Reference" table to obtain the Binder proxy (reference) object. After that, the Client can directly use the specific service corresponding to the Server side through the Binder proxy object. All these communication processes are built on the Binder driver. The Binder driver is like a router. It is responsible for the establishment of the Binder communication and the transfer of Binder objects between processes. It is the core of the Binder communication.

2.2 Binder Attack Model

In recent years, many CVEs [7] about Android are relevant to the Binder interface. Few researchers explored the security of the Binder mechanism itself, but only used Binder as an attack entry point. After analyzing a large number of Android system vulnerabilities, the BinderCracker author team found that [3], the fundamental reason for choosing Binder as the attack vector is: That the attacker can manipulate the underlying Binder interface to send the constructed malicious data to the server side, thereby bypassing all the integrity check of the upper client. As we know, Android is a mobile operating system jointly developed in Java and C++. The Android system generally sets a very comprehensive check in the public Application Programming Interface (API) provided by the upper Java framework layer and then uses the BpBinder interface encapsulated in Java step by step to the C++ Binder interface, and the bottom layer performs actual cross-process communication through the Binder driver. However, due to the existence of Java Native Interface (JNI) and Native Development Kit (NDK), an attacker can bypass the check in the Java public API by manipulating the underlying Native Binder interface directly.

3 Motivation

Inconsistent data transmission vulnerability. The inconsistent data transmission vulnerability between the client and the server can be shown in Fig. 2. The 3rd line data type of the client code of a system service transmits is long, but the data type of the corresponding server is int. Long occupies 8 bytes in memory, and int occupies 4 bytes in memory. From the 3rd line of the server, the starting position of the data read by the server will be wrong, so the data read in each following step will be wrong, which will directly cause the system service crash.

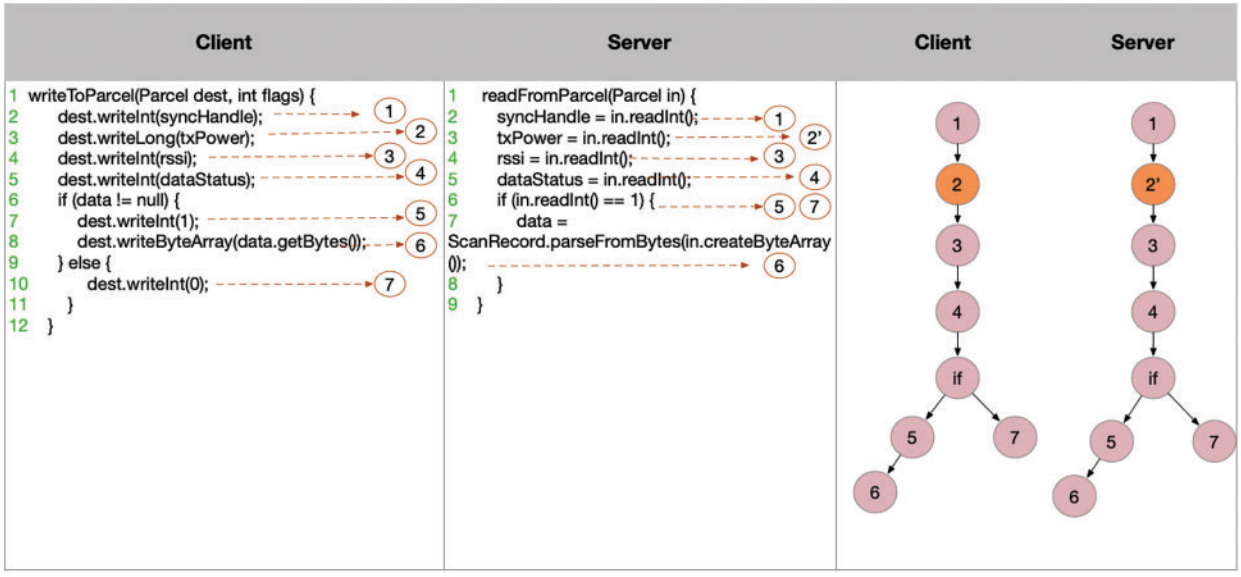

Figure 2: Inconsistent data transfer vulnerability between client and server

Inconsistent processing vulnerability. The inconsistent processing vulnerability between the client and the server can be shown in Fig. 3. There are a variety of processes of data interaction between client and server. For this sample code, there are four possible processes of data interaction of sending data from client to server: (1) 1 –> 2 –> 3 –> 4 –> 5; (2) 1 –> 2 –> 3 –> 4 –> 6; (3) 1 –> 2 –>

4 –> 5; (4) 1 –> 2 –> 4 –> 6. For the data receiver of server, there are two processes: (1) 1 –> 2 –> 3 –> 4 –> 6; (2) 1 –> 2 –> 4 –> 6. Based on the above analysis, if the client uses the data.writeint32 method in step 5 when constructing data, the server will use the data.readint64 method in step 6 for receiving data. This will cause the server to receive data out-of-bounds exceptions, resulting in the system service memory overflow.

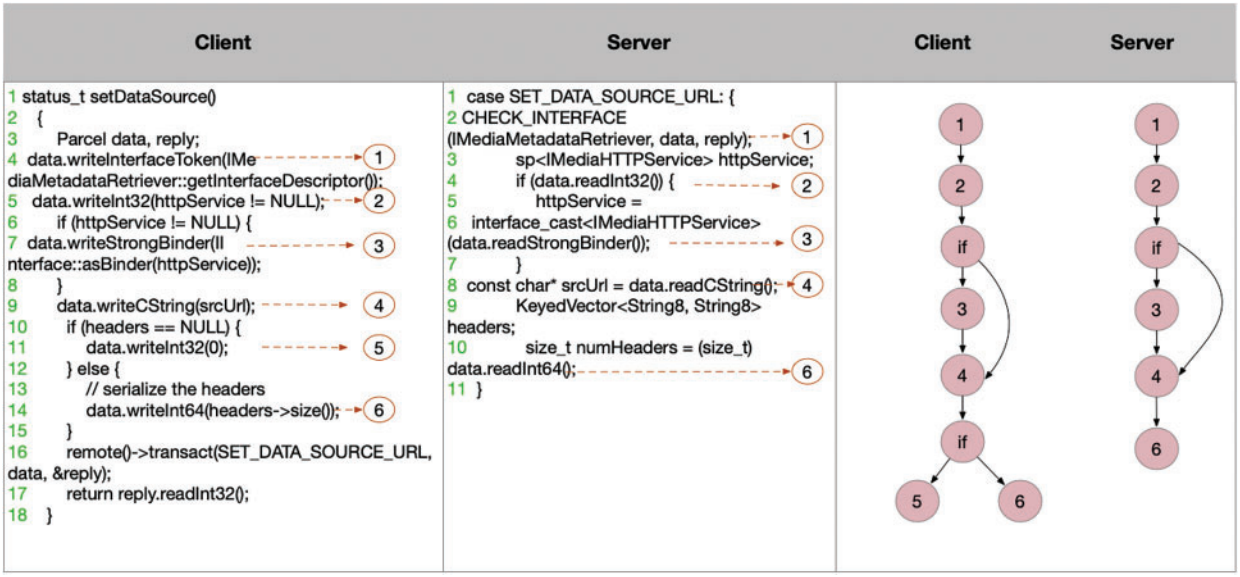

Figure 3: Inconsistent processing vulnerability

Based on the above analysis, the above two vulnerabilities cannot be directly analyzed based on the server code of the system service. The analysis of these two vulnerabilities needs to be based on the data transfer process between the client and the server.

4 Design and Implementation

This section elaborates on the proposed vulnerability mining method of the Android native system service. The overview design is shown in Fig. 4. First, We extract all interfaces of the system service through the client source code of the system service. These interfaces are the test entry points for fuzz testing. Second, based on interfaces, we need to know the number of parameters and data types required for each interface. We construct an abstract syntax tree (AST) for transmission data type corresponding to the interface. Third, based on the number of parameters and data types for each interface, we need to design algorithms to automate the generation of test cases. Fourthly, the fuzzing test of Android system services is conducted by using the constructed test cases, and triggering exceptions. BArcherFuzzer also monitors the code coverage in order to feed back to the test cases in Step 3 to dynamically adjust the generation strategy.

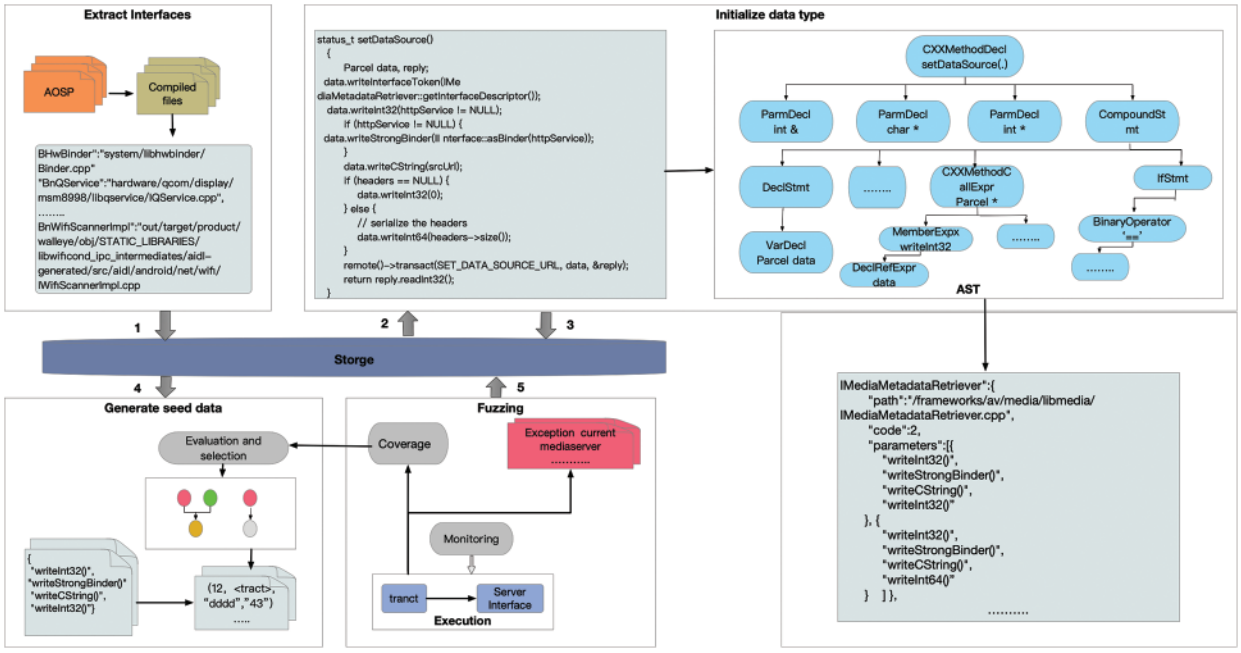

Figure 4: Overview of the framework of BArcherFuzzer

4.1 Extract Interfaces of Native System Service

In this paper, the Android system source code is used to obtain the native system service interface. Because BArcherFuzzer is based on the extraction of the client interface of the native system service, two problems need to be solved: (1) interface search; and (2) interface extraction.

(1) Interface search. The interfaces of the native system service in AOSP can be implemented in two forms: One is the code of C++; the other is encoding on the file of Android Interface Definition Language (AIDL). AIDL files are converted into C++ codes after AOSP compilation. Interface search aims to find all the interface definition classes and function codes of the client for the native system services in AOSP. Therefore, compiling AOSPI can unify the implementation methods of all interfaces of native system services into the C++ codes. In this paper, we manually analyze the code of the client interface of some native system services. The Java function calls the Binder server through the transact function of the BpBinder class. The sample code in Fig. 5 clearly shows that the class implemented by the client's interface inherits BpInterface. Through this feature, we can search all client interfaces in AOSP.

```
1    class BpMediaPlayerService: public BpInterface<IMediaPlayerService>{
2 public:
3    BpMediaPlayerService(const sp<IBinder>& impl)
4       : BpInterface<IMediaPlayerService>(impl){}
5    }
```

Figure 5: Sample code of a client interface of native system service

(2) Interface extraction. Based on the summarization of the interface characteristics, we design an interface extraction method based on a heuristic search algorithm. As shown in Algorithm 1, all the files of the compiled AOSP are AOSPFiles, and the content of them is fileContent. All

methods in the file are identified separately because a method belonging to a system service client interface has a unified inheritance interface BpInterface. So we match all methods based on rules that manual extraction to identify interfaces. Once a method matches a rule, the interface name interfaceName is obtained, and the interface name and the file path Interfaces where the interface is located.

Algorithm 1: Heuristic search interface algorithm

Require: AOSPFiles ⇐ compiled AOSP files, Rules ⇐ rule set of service interfaces
Ensure: Interfaces {Store all interfaces}
1: **for** each fileItem ∈ AOSPFiles **do**
2: fileContent ⇐ open (fileItem)
3: **for** each method ∈ fileContent **do**
4: interfaceName ⇐ getInterface (method, Rules)
5: **if** interfaceName ≠ ∅ **then**
6: Interfaces.put (interfaceName, fileItem)
7: **end if**
8: **end for**
9: **end for**
10: **return** Interfaces

4.2 Extract the Types of Data Sent by the Client

The fuzzing method needs to simulate the client of the native system service to send real data to the server to call the service. Therefore, for the client interface of the obtained native system service, BArcherFuzzer needs to obtain the data types of each interface. Fig. 6 shows a sample code of the Android native system service, in which the client sends a service call to the server.

```
1   status_t setRetransmitEndpoint(const struct sockaddr_in* endpoint){
2      Parcel data, reply;
3      status_t err;
4      data.writeInterfaceToken(IMediaPlayer::getInterfaceDescriptor());
5      if (NULL != endpoint) {
6         data.writeInt32(sizeof(*endpoint)); /*write Int32 type data*/
7         data.write(endpoint, sizeof(*endpoint));/*write Int type data*/
8      } else {
9         data.writeInt32(0); /*write Int32 type data*/
10     }
11     err = remote()->transact(SET_RETRANSMIT_ENDPOINT, data, &reply);
12     if (OK != err) {
13        return err;
14     }
15     return reply.readInt32();}
```

Figure 6: Sample code for sending data from the client of the system service

The 11th line is that the client calls the transaction sending function transact() to communicate with the server of the system service. The data transmitted to the server through the network is data with Parcel type. SET_RETRANSMIT _ENDPOINT is the called destination service, and the reply is the response information sent to the client by the server. In lines 6 to 9, the data types contained in data are Int32 and Int, and different services will use different data types. Different data types occupy different numbers of bytes in memory. When the server receives the data and parses it, it will reverse

the corresponding data according to the number of bytes. Otherwise, the service will read the number of bytes out of range, and the system service will crash.

The target sending data extracted by BArcherFuzzer is data with Parcel type in each interface. To identify all possible data types in data, BArcherFuzzer uses the abstract syntax tree (AST) to parse the source code of the client. Fig. 7 is the AST of Fig. 6, which shows the complete data type of data object, and it can show the different data types in different processes. Algorithm 2 describes the process of extracting the type of data included in data based on AST. AST is the format in which the client interface code is converted into AST. At first, traversing each node from the root node of AST, if the node has the identity of 'data', getting the type attribute value saved by the type of the node. Then storing the attribute value in the type storage variable ExpRet. If the node belongs to the conditional branch IfStmt, it indicates that the filling data of data has multiple branches. To obtain all possible types of data, copy the data of ExpRet as ExpRetNew, and then for each subtree CallExpx belonging to the child node of the conditional branch node CallExpx childTree performs data type recognition by repeating the above process. Finally, DataTypeSet saves the type of data in all possible situations.

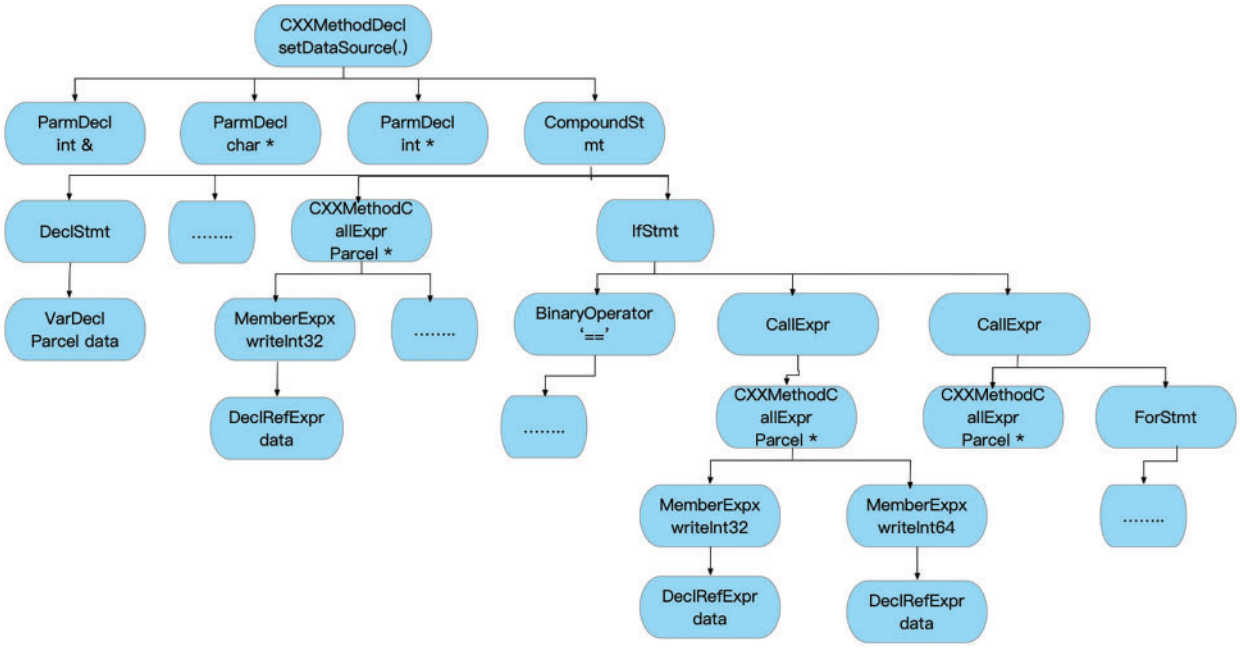

Figure 7: AST of the sample code

Algorithm 2: Construct the data types based on AST

Require: AST ⇐ AST, ExpRet ⇐ all data types for a conditional branch
Ensure: DataTypeSet {all data types set of an interface}
1: **for** each node ∈ AST **do**
2: **if** string 'data' ∈ node **then**
3: type ⇐ Get the data type of the node getType (node)
4: ExpRet ⇐ store data type in ExpRet add (ExpRet, type)
5: **end if**

(Continued)

Algorithm 2 (continued)

6: **if** string 'IfStmt' ∈ node then
7: **for** Get all nodes from the subtreeCallExpx callExpxNode **do**
8: ExpRetNew ⇐ copy the data deepcopy (ExpRet)
9: childTree ⇐ get all nodes from the subtreegetChildTree (callExpxNode)
10: TraveConstruct (childTree, ExpRetNew, DataTypeSet) {Recursively call the method of constructing the data type}
11: **end for**
12: **end if**
13: **end for**
14: DataTypeSet ⇐ Store all possible data types inDataTypeSet add (DataTypeSet, ExpRet)

4.3 Construct Test Cases

Based on the extracted data types in data, we introduce the method to generate test cases corresponding to the native system service interface. Test cases directly affect whether vulnerabilities can be triggered. As shown in Algorithm 3, the construction of a test case includes initialization and test case generation.

Algorithm 3: Test case generation based on genetic algorithm

Require: DataTypeSet ⇐ data all possible filling data types for data, w ⇐ threshold of the function, size ⇐ threshold of seed size
Ensure: seedDataSet {all data test cases}
1: **for** each dataType ∈ DataTypeSet **do**
2: seedDataSet ⇐ initialization seedData = random (dataType)
3: **end for**
{The initialized generation of test cases can be used to fuzzing, the following is the process of generating test cases after fuzzing}
4: **for** each s ∈ seedDataSet **do**
5: fitValue ⇐ fit(s)
6: **if** fitValue > w **then**
7: TmpSeedDataSet ⇐ Keep test cases as an alternativeseedItem
8: **end if**
9: **end for**
10: SeedDataSet⇐ ∅ {Empty the data set}
11: **for** each s ∈ random (TmpSeedDataSet) **do**
12: **if** fs (s,ni) == true **then**
13: continue
14: **end if**
15: paramList = sort (s)
16: **for** each param ∈ paramList **do**
17: seedDataSet ⇐ Do mutation operation mutate (s,param)
18: **end for**
19: **end for**

(1) Initialization

First, according to the extracted data type in data, initialize a test case seedData and add it to the set seedDataSet of test cases. The initialization of the test case is based on the data type of data. Data types include basic data types, such as integer (Int), floating-point type (Float), and string (String), as well as derived data types, such as Binder. To maximize the probability that test cases trigger the abnormality of the system service, the boundary value is used for each conventional data type in the initialization. For the system class type, the construction method belonging to the object class can be obtained by the analysis of the client interfaces, so the entity of the corresponding class can be obtained. In summary, the initialization of different data types in the above test cases is shown in Table 1. The seed pool seedDataSet generated at the beginning can be applied to subsequent service tests directly.

Table 1: Initialize the data type of the test case

Data type	Initialize values	Data type	Initialize values
Int	0, 1, −1, null, Int.Max, Int.Min, random	Int[]	Construct an array of random size and insert integer values
Float	0, 1, −1, null, Float.Max, Float.Min, random	Float[]	Construct an array of random size and insert float values
String	null, “”, string with special character, random length string	String[]	Construct an array of random size and insert string values
Boolean	true, false	Boolean[]	Construct an array of random size and insert bool values
Double	0, −1.0, 1.0, null,Double.Max, Double.Min, random	Double[]	Construct an array of random size and insert double values
Long	0, 1, −1, null, Long.Max, Long.Min, random	Long[]	Construct an array of random size and insert long values
Byte	0, 1, −1, null, Byte.Max, Byte.Min, random	Byte[]	Construct an array of random size and insert byte values
System class	null, create method of system class		

(2) Test case generation based on genetic algorithm

After the first round of fuzz testing, to increase the test coverage of the system service to trigger more vulnerabilities with the service. As described in Algorithm 3, we use a genetic algorithm to construct test cases. For each test case s, we use the fitness function fit() shown in the formula 1 to calculate its fitness value fitValue.

$$fit\,(s) = \begin{cases} cr, & No\ \ exception\ \ is\ \ triggered \\ 0, & Triggered\ \ exceptions \end{cases} \tag{1}$$

If one test case s has triggered the service error, it is the optimal solution, and subsequent mutations are no longer needed, so the fitness value is 0. If the test case has not triggered the service error, its fitness value is the coverage rate cr of the services driven by it. When the fitness value of a test case is greater than the threshold w, it will be a candidate test case data to set TmpSeedDataSet. If w is set

too large, a large number of test cases will be eliminated. If w is set too small, it will reduce the test efficiency for too many test cases are included. We set w to be 20%.

The filtered test cases need to perform mutation operations on the data. Preliminary analysis shows that the data contains multiple parameters, which leads to the need for multi-parameter mutation in the parameter mutation stage. The mutation method designed in this paper is based on the test efficiency, that is, to ensure the test coverage and reduce the test time. Therefore, the parameters in the test case are divided into two priority levels: (1) The high priority consists of non-array type parameters; (2) The low priority consists of array type parameters. The mutation selects parameters with high priority and the process parameters with low priority. The method of random screening is adopted for the parameters of the same priority, but it is guaranteed that the same parameter will not be subjected to repeated mutation processing. As described in Algorithm 3, randomly obtain one of the test cases s in the candidate seed pool TmpSeedDataSet, and obtain the parameter priority order list of sparamList. According to the parameter priority, a new test case is formed by mutation and added to the seed pool seedDataSet.

To ensure the test efficiency of the test case, formula (2) is the termination function of the test case fs(). When the test case s triggers an exception, the test case does not need to be mutated. When the number of mutation iterations of the test case s has reached the maximum value MC (set to 400), the test case s will not continue the mutation test.

$$fs(s, ni) = \begin{cases} true, & s \ \ triggered \ \ exceptions \\ true, & ni \geq MC \\ false, & otherwise \end{cases} \tag{2}$$

4.4 Drive Native System Service

As shown in the 11th line of code in Fig. 6, the client of the native system service realizes the communication with the server by calling the transact() function. Based on the native system service interface obtained in the above process and the generated test data of the corresponding interface, BArcherFuzzer customizes a script to call the native system service. The script realizes the communication between the client and the server by explicitly calling the transact() function. BArcherFuzzer supports dynamic fuzzing of all native system service interfaces. In the process of fuzzing, this article also monitors the abnormal log of the system and the code coverage of the tested system service. This article is based on Frida [8] to monitor the code coverage of system services. All abnormal data during the fuzzing is stored for finding vulnerabilities. Once the test case is used, the code coverage driven by it is recorded. These data will be used in conjunction with algorithm 3 to generate new test seed data.

5 Evaluation

In the experimental section, we focus on answering two research questions (RQs):

(1) **RQ1:** Can BArcherFuzzer detect vulnerabilities for native system services?
(2) **RQ2:** Can the genetic algorithm optimize the fuzz test coverage?

5.1 Experimental Data and Environment

Environment. The computer used in the experimental environment is an i7 CPU and 8 GB memory and is equipped with a Ubuntu 18.04 system. The smartphones are Nexus 5x and Pixel 2 with versions android-6.0, android-7.1.2,android-8.0, android-9.0, and android-11.0.

5.2 Results and Analysis

(1) Answering RQ1: BArcherFuzzer can detect native system service vulnerabilities

We set the Android 6.0 system in the Nexus 5x phone, and the Android 11.0 system in Pixel 2 phone. BArcherFuzzer continues to work for 30 days on the two mobile phones separately. Hundreds of exception messages were recorded in the experimental results. We manually confirmed the exception messages and reported them to Google. There were 4 confirmed vulnerabilities. The above vulnerabilities were caused by inconsistent data and inconsistent logic processing of Android native system services. After the official assessment of the difficulty of exploiting the vulnerabilities we submitted, Google assigned a CVE number to one of them.The details are shown in Table 2. Among them, CVE-2020-0363 is a vulnerability caused by inconsistent processing between the client and the server. In Table 2, it can be seen that BArcherFuzzer can effectively detect Android native system service vulnerabilities. Based on the vulnerabilities found in the test environment, BArcherfuzzer work on other versions of Android system with the same test cases and detect same vulnerabilities. BArcherFuzzer can work with most major Android versions.

Table 2: Details of detected vulnerabilities

Android ID	Server	Android OS	Status
132200822	Media server	6.0, 7.1.2, 8.0, 9.0	CVE-2020-0363
129579042	Media server	6.0, 7.1.2, 8.0, 9.0	Confirmed
172841078	Media server	11.0	Confirmed
172851300	Media server	11.0	Confirmed

1) Case study-CVE-2020-0363

There is a native system service BpMediaMetadataRetriever in the IMediaMetadataRetriever.cpp file in the Android 6.0 system. As shown in Fig. 8, the code for requesting the function of setDataSource in the client of the service. There are two conditional branches in the function process. If the condition of the 14th line is met, then the 15th line is executed, and the data encapsulated in the data is Int32; otherwise, the 18th line is executed, and the data encapsulated in the data is Int64. As shown in Fig. 9, in the code of the server response, the request sent by the client is obtained from the data in the deserialized data, and the processing logic has only one conditional branch. The 10th line indicates that data directly deserializes data according to Int64. In this way, once the encapsulated data of the client is Int32, the client's data encapsulates the data of Int32, resulting in the confused data by reading between the 10th and 15th lines of the server. It causes the existence of deserialization vulnerabilities, which can lead to risks such as Dos attacks and information leakage.

```
status_t setDataSource(const sp<IMediaHTTPService> &httpService,const char *srcUrl,
const KeyedVector<String8, String8> *headers){
   Parcel data, reply;
   data.writeInterfaceToken(IMediaMetadataRetriever::getInterfaceDescriptor());
   data.writeInt32(httpService != NULL); /* write Int32 type data */
   If (httpService != NULL) {
       /* write Binder type data */
      data.writeStrongBinder(IInterface::asBinder(httpService));
   }
   data.writeCString(srcUrl); /* write String type data */
   If (headers == NULL) {data.writeInt32(0); /* write Int32 type data */
   } else {
      // serialize the headers
      data.writeInt64(headers->size()); /* write Int64 type data */
      For (size_t i = 0; i < headers->size(); ++i) {
         data.writeString8(headers->keyAt(i)); /* write String8 type data */
         data.writeString8(headers->valueAt(i)); /* write String8 type data */}
         }
         /*Invoke server get system service*/
   remote()->transact(SET_DATA_SOURCE_URL, data, &reply);
   Return reply.readInt32(); }
```

Figure 8: Sample code for the client (BpBinder) to call the system service

```
case SET_DATA_SOURCE_URL: {
   CHECK_INTERFACE(IMediaMetadataRetriever, data, reply);
   sp<IMediaHTTPService> httpService;
   If (data.readInt32()) { /*read Int32 type data*/
     httpService =
     interface_cast<IMediaHTTPService>(data.readStrongBinder());
     /*read Binder type data*/
   }
   const char* srcUrl = data.readCString(); /*read String type data*/
   KeyedVector<String8, String8> headers;
   size_t numHeaders = (size_t) data.readInt64(); /*read Int64 type data*/
   For (size_t i = 0; i < numHeaders; ++i) {
     String8 key = data.readString8(); /*read String8 type data*/
     String8 value = data.readString8();/*read String8 type data*/
     headers.add(key, value);
   }
   reply->writeInt32(setDataSource(httpService, srcUrl,
                  numHeaders > 0 ? &headers : NULL));
   Return NO_ERROR; }
```

Figure 9: Sample code for the server (BBinder) to respond to system service request

Insight. Experimental results show that BArcherFuzzer can effectively detect vulnerabilities in Android native system services, and can detect vulnerabilities caused by inconsistent processing between the client and the server.

(2) Answering RQ2: The test case generation algorithm can improve code coverage

For the fuzzing test of Android system services, test cases are one of the key factors that affect the effect. A good test case construction algorithm can enable fuzzing tool to trigger more code coverage in a short time and find more exceptions. To increase the test coverage of the fuzzing test, the genetic algorithm (GA) is used to generate the test cases. The seed generation algorithm of random (RD) is commonly used in fuzzy testing. To verify the effectiveness of this method, we use seed data generated by a genetic algorithm to do a fuzz test (Fuzzer+GA) and seed data generated by a random algorithm to do a fuzz test (Fuzzer+RD) for comparison experiments. We used dynamic instrumentation technology (Frida) to count the number of executed code blocks. The experiment selects the same service interfaces and the same number of iterations for testing and then calculates the average coverage rate. The result is shown in Fig. 10, it can be seen the generation of seed based on a genetic algorithm can improve the coverage rate, and reach a coverage rate of 32% in a few iterations,

which can improve the testing efficiency of the fuzzing. Additionally, it also takes less time to achieve the same code coverage.

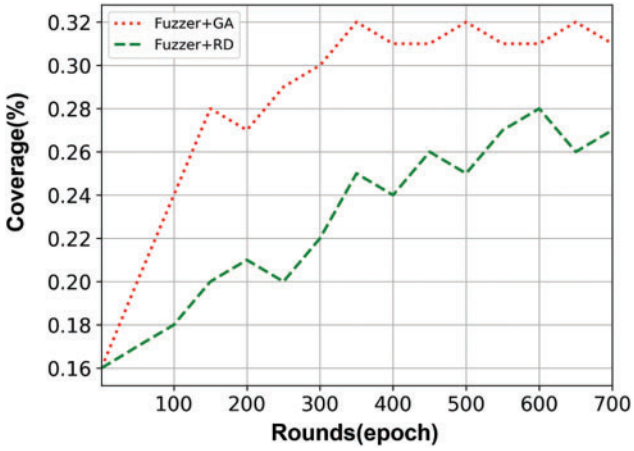

Figure 10: Coverage rate of fuzzing using genetic algorithm and non-genetic algorithm

Insight. The genetic algorithm to generate test cases can not only improve the coverage of the test code but also increase the coverage to 32% in a short time. The efficiency of the fuzzing is sped up.

6 Related Work

The research on Android vulnerability detection is mainly divided into two aspects: One is vulnerability detection for Android applications and the other is vulnerability detection for Android systems.

First, review the vulnerability research of the Android system. Gong et al. [9] paid attention to the security of the Binder IPC interfaces of the Android system service. They pointed out that Binder is the actual security boundary of the Android system service. Through manual testing, they discovered the critical vulnerabilities related to Binder and proved its insecurity, they also reflected the importance of Binder's security to the Android system. Wang et al. [10] proposed a fuzzing test method for the Android Java service interface compiled from AIDL files. Chizpurfle [4] used the Java reflection mechanism instead of relying on the Android system source code to obtain the Java service interface and its parameter types. Based on the above information, Chizpurfle was used to test the Java service vulnerabilities of the Android system of the third-party manufacturer's customized system. However, this detection method cannot be used to detect vulnerabilities in Android local system services. In addition, some studies [11,12] are focusing on input verification vulnerabilities of Android services. Several other studies [13–15] focused on vulnerabilities caused by inconsistency in access control associated with Android services in the Android framework. KOOBE [16] provided a comprehensive survey on kernel vulnerability mining technology, summarizing the current methods and technologies and pointing out future research directions. Wang et al. [17] proposed an Android kernel vulnerability mining technology based on static analysis. The authors used static analysis methods to scan and analyze the source code of the Android kernel to identify potential security vulnerabilities. Lu et al. [18] proposed an Android kernel vulnerability mining technology based on dynamic analysis. The authors used dynamic analysis methods to monitor and test the running Android kernel to identify potential security vulnerabilities.

For the vulnerability research of Android applications, Wei et al. [19] analyzed the vulnerabilities caused by the use of JavaScript in the APP. However, they did not give a verification method for the vulnerability. Mutchler et al. [20] analyzed large-scale applications and found vulnerabilities in every corner of the Android ecosystem. Nevertheless, they did not give an analysis method for specific vulnerabilities. Chin et al. [21] focused on analyzing Android's WebView component-related vulnerabilities and summarized the vulnerability analysis methods. Li et al. [22] conducted a detailed analysis of vulnerabilities in the background communication process of music Android applications and proposed a detection tool by analyzing the causes of the vulnerabilities. Yang et al. [23] are concerned about the security issues caused by hybrid Origin Stripping Vulnerability (OSV) in Android applications. Based on the analysis, a detection tool for the above-mentioned vulnerabilities was proposed. Qian et al. [24] used an APP attribute graph (APG) structure to represent APP vulnerabilities. Then, detection tools are proposed for 5 common Android application layer vulnerabilities. Wu et al. [25] detected the confused deputy vulnerability of the Android application based on the features of the Android Manifest file and the control flow graph (CFG).

Analysis and research on Android inter-process communication vulnerabilities [26–31] usually use data flow analysis methods. AmanDroid [26] identified privacy leak vulnerabilities by tracking component interactions. Iccta [29] solved the problem of privacy leakage through taint flow analysis. Epicc [30], IC3 [31] and Lee et al. [32] are both used to statically extract information from Android applications to detect communication vulnerabilities between APP processes. Ma et al. [33] proposed a method that converts Java code into abstract syntax tree (AST) features and then used machine learning models to detect Java programs' vulnerabilities. At the same time, they also proposed unique repair fragments for different vulnerabilities. To achieve the purpose of both detection and repair, Dam et al. [34] also proposed a vulnerability detection model for the Java language. They used LSTM to obtain the grammatical features of the Java program, obtained the semantic features through the clustering algorithm, and used the random forest algorithm for vulnerability detection.

7 Limitation

This study has potential limitations. Although the research results of this paper have shown that BArcherFuzzer can find related vulnerabilities in multiple versions of Android systems, the number of CVEs is still small. In addition, there are a large number of bugs found at present, and it takes manpower to re-verify and screen effective and confirmed vulnerabilities.

Future research should be undertaken to explore the method applicable to the vulnerability discovery of other IoT devices of Android systems and to explore the test case generation algorithm with better performance.

8 Conclusion

In this paper, we aimed at the data inconsistency vulnerability and processing between the client and the server of the Android native system service. We propose a fuzzing method based on interfaces of the native system service of the client. To optimize the fuzzing code coverage and test efficiency, we propose a test case construction method that supports multi-parameter mutation based on the genetic algorithm and the priority strategy. Based on the above methods, we design and implement a fuzzer-BArcherFuzzer, It can achieve a coverage rate of 32% in 200 test case iterations. In the system test for Android 6.0 and 11.0, 4 vulnerabilities were successfully discovered from hundreds of crash messages, 3 of which were confirmed by Google and 1 was assigned a CVE number (CVE-2020-0363).

Acknowledgement: We are hugely grateful to the possible anonymous reviewers for their constructive comments with respect to the original manuscript.

Funding Statement: This work was supported by the National Key R&D Program of China (2023YFB3106800) and the National Natural Science Foundation of China (Grant No. 62072051). We are overwhelmed in all humbleness and gratefulness to acknowledge my depth to all those who have helped me to put these ideas.

Author Contributions: Study conception and design: Jiawei Qin, Hua Zhang; data collection: Song Hu, Dingyu Yan; analysis and interpretation of results: Jiawei Qin, Hua Zhang; draft manuscript preparation: Hanbing Yan, Tian Zhu. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: All data generated or analyzed during this study are included in this published article.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] iiMedia.cn, "Operating system industry data analysis: In 2019, the market share of android operating system on mobile phones reached 68.63%," 2020. Accessed: May 01, 2020. [Online]. Available: https://www.iimedia.cn/c1061/70980.html

[2] Google, "Android security bulletins," 2019. Accessed: Feb. 10, 2019. [Online]. Available: https://source.android.com/security/bulletin/

[3] H. Feng and K. G. Shin, "Understanding and defending the binder attack surface in android," in *32nd Annual Conf. Comput. Secur. Appl.*, Los Angeles, California, USA, 2016, pp. 398–409.

[4] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nitarotaru, "Chizpurfle: A gray-box android fuzzer for vendor service customizations," in *IEEE 28th Int. Symp. Softw. Reliab. Eng. (ISSRE)*, Kyoto, Japan, 2017, pp. 1–11.

[5] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan and J. Zhuge, "FANS: Fuzzing android native system services via automated interface analysis," in *29th USENIX Secur. Symp.*, Berkeley, CA, USA, 2020, pp. 307–323.

[6] H. Yuan, "The Android operating system architecture," 2020. Accessed: Feb. 11, 2020. [Online]. Available: http://gityuan.com/android/

[7] The MITRE Corporation, "CVE," 2020. Accessed: June 20, 2020. [Online]. Available: https://cve.mitre.org/

[8] Ravnas, "FRIDA," 2020. Accessed: Jun. 28, 2020. [Online]. Available: https://frida.re/

[9] Guang Gong, "Fuzzing android system services by binder call to escalate privilege," 2020. Accessed: Jun. 02, 2020. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Gong-Fuzzing-Android-System-Services-By-Binder-Call-To-Escalate-Privilege.pdf

[10] K. Wang, Y. Q. Zhang, Q. X. Liu, and F. Dan, "A fuzzing test for dynamic vulnerability detection on Android Binder mechanism," in *2015 IEEE Conf. Commun. Netw. Secur. (CNS)*, Florence, Italy, 2015, pp. 709–710.

[11] C. Cao, N. Gao, L. Peng, and X. Ji, "Towards analyzing the input validation vulnerabilities associated with android system services," in *31st Annual Comput. Secur. Appl. Conf.*, Orlando, Florida, USA, 2015, pp. 361–370.

[12] L. Zhang *et al.*, "Invetter: Locating insecure input validations in android services," in *2018 ACM SIGSAC Conf. Comput. Commun. Secur.*, Toronto, Ontario, Canada, 2018, pp. 1165–1178.

[13] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. Mao, "Kratos: Discovering inconsistent security policy enforcement in the android framework," in *Netw. Distrib. Syst. Secur. Symp. (NDSS'16)*, San Diego, California, USA, 2016.
[14] Y. Aafer *et al.*, "Normalizing diverse android access control checks for inconsistency detection," in *Netw. Distrib. Syst. Secur. Symp.(NDSS'18)*, San Diego, California, USA, 2018.
[15] S. A. Gorski *et al.*, "ACMiner: Extraction and analysis of authorization checks in android's middleware," in *Ninth ACM Conf. Data Appl. Secur. Privacy*, New Orleans, Louisiana, 2019, pp. 25–26.
[16] C. Weiteng, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *29th USENIX Secur. Symp. (USENIX Security 20)*, Boston, Massachusetts, USA, 2020, pp. 1093–1110.
[17] Y. Wang, X. Li, and W. Zhang, "Android kernel vulnerability mining based on static analysis," in *Proc. 10th Int. Conf. Inf. Secur. Cryptol.*, Beijing, China, 2021, pp. 147–162.
[18] S. Lu, X. Kuang, Y. Nie, and Z. Lin, "A hybrid interface recovery method for android kernels fuzzing," in *2020 IEEE 20th Int. Conf. Softw. Quality, Reliab. Secur. (QRS)*, Macau, China, 2020, pp. 335–346.
[19] W. Song, Q. Huang, and J. Huang, "Understanding javascript vulnerabilities in large real-world android applications," *IEEE Trans. Dependable Secure. Comput.*, vol. 17, no. 5, pp. 1063–1078, Jun. 2018. doi: 10.1109/TDSC.2018.2845851.
[20] P. Mutchler *et al.*, "A large-scale study of mobile web app security," in *Mobile Secur. Technol. Workshop (MoST)*, 2015, pp. 50.
[21] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in android applications, information security applications," *Inform. Secur. Appl.*, vol. 8267, no. 1, pp. 138–159, Mar. 2014. doi: 10.1007/978-3-319-05149-9_9.
[22] H. Li, L. Qian, S. Zhang, H. Zhang, and J. Liu, "Data leakage between C/S communication: A case study on Android music app," in *2017 9th Int. Conf. Wirel. Commun. Signal Process. (WCSP)*, Nanjing, China, 2017, pp. 1–6.
[23] G. Yang, J. Huang, G. Gu, and A. Mendoza, "Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications," in *2018 IEEE Symp. Secur. Privacy (SP)*, San Francisco, California, USA, 2018, pp. 742–755.
[24] C. Qian, X. Luo, Y. Le, and G. Gu, "VulHunter: Toward discovering vulnerabilities in android applications," *IEEE Micro*, vol. 35, no. 1, pp. 44–53, Jan. 2015. doi: 10.1109/MM.2015.25.
[25] J. Wu, T. Cui, T. Ban, S. Guo, and L. Cui, "PaddyFrog: Systematically detecting confused deputy vulnerability in android applications," *Secur. Commun. Netw.*, vol. 8, no. 13, pp. 2338–2349, Jul. 2015. doi: 10.1002/sec.1179.
[26] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Trans. Priv. Secur.*, vol. 21, no. 3, pp. 5–37, Aug. 2018. doi: 10.1145/3183575.
[27] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *3rd ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, Edinburgh, UK, 2014, pp. 1–6.
[28] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE Trans. Soft. Eng.*, vol. 41, no. 9, pp. 866–886, Sep. 2015. doi: 10.1109/TSE.2015.2419611.
[29] Li Li *et al.*, "IccTA: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE Int. Conf. Soft. Eng.*, Florence, Italy, 2015, pp. 280–291.
[30] D. Octeau *et al.*, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *22nd USENIX Conf. Secur.*, Berkeley, California, USA, 2013, pp. 543–558.
[31] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *IEEE/ACM 37th IEEE Int. Conf. Soft. Eng.*, Florence, Italy, 2015, pp. 77–88.

[32] Y. K. Lee, J. Y. Bang, G. Safi, A. Shahbazian, Y. Zhao and N. Medvidovic, "A sealant for inter-app security holes in android," in *2017 IEEE/ACM 39th Int. Conf. Soft. Eng. (ICSE)*, Buenos Aires, Argentina, 2017, pp. 312–323.

[33] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "VuRLE: Automatic vulnerability detection and repair by learning from examples," in *Comput. Secur.–ESORICS 2017*, Oslo, Norway, 2017, pp. 229–246.

[34] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Trans. Soft. Eng.*, vol. 47, no. 1, pp. 67–85, Jan. 2021. doi: 10.1109/TSE.2018.2881961.