IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 4, 2020, accepted May 23, 2020, date of publication May 27, 2020, date of current version June 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2998043

Vulnerability Detection on Android Apps–Inspired by Case Study on Vulnerability Related With Web Functions

JIAWEI QIN[1], HUA ZHANG[1], (Member, IEEE), JING GUO[2], SENMIAO WANG[1], QIAOYAN WEN[1], AND YIJIE SHI[1]

[1]State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China
[2]National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing 100029, China

Corresponding author: Hua Zhang (zhanghua_288@bupt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB0804703.

ABSTRACT Nowadays, people's lifestyle is more and more dependent on mobile applications (Apps), such as shopping, financial management and surfing the internet. However, developers mainly focus on the implementation of Apps and the improvement of user experience while ignoring security issues. In this paper, we perform the comprehensive study on vulnerabilities caused by misuse of APIs and form a methodology for this type of vulnerability analysis. We investigate the security of three types of Android Apps including finance, shopping and browser which are closely related to human life. And we analyze four vulnerabilities including *Improper certificate validation(CWE-295:ICV)*, *WebView bypass certificate validation vulnerability(CVE-2014-5531:WBCVV)*, *WebView remote code execution vulnerability(CVE-2014-1939:WRCEV)* and *Alibaba Cloud OSS credential disclosure vulnerability(CNVD-2017-09774:ACOCDV)*. In order to verify the effectiveness of our analysis method in large-scale Apps on the Internet, we propose a novel scalable tool - VulArcher, which is based on heuristic method and used to discover if the above vulnerabilities exist in Apps. We download a total of 6114 of the above three types of samples in App stores, and we use VulArcher to perform the above vulnerability detection for each App. We perform manual verification by randomly selecting 100 samples of each vulnerability. We find that the accuracy rate for *ACOCDV* can reach 100%, the accuracy rate for *WBCVV* can reach 95%, and the accuracy rate for the other two vulnerabilities can reach 87%. And one of vulnerabilities detected by VulArcher has been included in China National Vulnerability Database (CNVD) ID(CNVD-2017-23282). Experiments show that our tool is feasible and effective. For the convenience of researchers in related communities, We make our data and tool available at https://buptnsrclab.github.io/blog/2020/01/03/vularcher-site-launched.

INDEX TERMS Static analysis, vulnerability, mobile agents, security, application software, detection algorithms.

I. INTRODUCTION

With the rapid growth of functional requirement of mobile Apps, the development iteration of Apps is more and more rapid. In this circumstance, developers focus on how to quickly implement the logical functions of Apps and make it more user-friendly. Nevertheless, they spend little time on security issues of Apps.

The security issues of Apps are complex and challenging. Related research on Android security has been proposed, including static [1]–[8] and dynamic [9], [10] methods. Wei *et al.* [11] just analyzed Android vulnerabilities which are assosiated with JavaScript. Mutchler *et al.* [12] and Chin and Wagner [13] analyzed a large-scale of mobile web Apps, and found that vulnerabilities are presented in all corners of the Android ecosystem. They did not propose methods for vulnerability verification. Li *et al.* [14] analysed a small set of Android music Apps and focused on the issue of data leakage during client and server communication. Li *et al.* [15], [16] proposed a solution to quickly identify Android source code vulnerability ignoring to analyse Apps' vulnerabilities. Mu *et al.* [17] contributed to how to quickly

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

reproduce the vulnerabilities by combining the vulnerability reports with widely crowdsourced information. In 2019, Diao *et al.* [8] analyzed one vulnerability caused by the usage of the accessibility API. Amin *et al.* [9] focused on proposing a detection tool for Android vulnerabilities, they did not give out a methodology for analyzing Android vulnerabilities. Luo *et al.* [18] was concerned about the vulnerabilities of the Android Framework and adopt a symbolic execution method to detect vulnerabilities. Wu *et al.* [19] conducted a systematic study for Android system vulnerabilities.

Aforementioned vulnerabilities researches are related to the web vulnerabilities [20]. Although these vulnerabilities have been reported for a long time, they have received little attention. Moreover, researchers did not consider how to analyze the vulnerabilities of these Apps if it is packed [21]–[23]. To the best of our knowledge, there is little effective analysis on aforementioned vulnerabilities. There is a very serious problem that many vulnerabilities reported early still exist in latest Apps, even in the same App. The main reason is that there is no feasible and complete vulnerability verification workflows for developers. In additional, more and more web functions are used in Apps [24], these functions may also increase new vulnerabilities of Apps. Users send their information through the network in the process of Apps,which makes it is particular important for Apps to ensure(assure) their security.

In this paper, our target of vulnerabilities are related to web functions. By manual analysis of 400 Apps, we find the vulnerabilities caused by the misuse of APIs accounted for the majority, and the most of vulnerabilities caused by API misuse are *WRCEV*, *WBCVV*, *ICV* and *ACOCDV*. So we chose these four vulnerabilities as our study goals. At the same time, we divide the above four vulnerabilities into three categories based on their behaviors: *Overridden method(Cat1:OM)*, *Use unsafe settings(Cat2:USS)* and *Data leakage of sensitive information(Cat3:DLSI)*.

Contributions. Based on the above analysis, we explore the severity of these vulnerabilities and perform the comprehensive study on the four vulnerabilities caused by the misuse of APIs and form an analysis methodology for each category of vulnerabilities. In order to mitigate them, we propose corresponding improvement recommendations. Based on the methodology, we propose a vulnerability verification workflow for each vulnerability. In order to verify the effectiveness of our analysis methodology in large-scale Apps on the Internet. Based on our findings, we propose a vulnerability detection tool, named VulArcher. It can detect the above four vulnerabilities in packed or unpacked Android Apps,and the average accuracy rate is 91%.The main contributions are as follows:

(1) For the four vulnerabilities that are caused by the misuse of APIs, we divide them into three categories based on their behaviors. we perform the comprehensive study on them and form an analysis methodology for each vulnerability, which can help researchers to discover other vulnerabilities in the three categories. Based on the analysis, we provide improvement recommendations to mitigate the vulnerability threat.

(2) Based on the methodology, we propose a vulnerability verification for workflow for each vulnerability, relevant researchers can verify corresponding vulnerabilities following them.

(3) In order to verify our methodologies in large-scale Apps, we propose a static and scalable vulnerabilities detection tool based on our findings for vulnerability analysis,VulArcher. We download 6,177 Android Apps containing transactions from the Internet, which are the latest version in 2018. After our random manual verification of the results of VulArcher, it shows that the average accuracy rate is 91%. In addition, VulArcher can also detect the vulnerabilities on packed Apps. By adding rules in a configuration file, VulArcher can detect other categories of vulnerabilities. We will share VulArcher at https://buptnsrclab.github.io/blog/2020/01/03/vularcher-site-launched.

II. BACKGROUD

A. ANDROID APP OVERVIEW

An Android App file is a compressed file that contains *AndroidManifest.xml* file, certificate files, *dex* file, static resource files and so on.

1) *Androidmanifest.xml*, which defined and included in every App, describes the name, version, permissions, component name and other information of the App.
2) Certificate files hold the signature information that ensures the integrity of the App.
3) *Dex* file is an executable file, which contains all the operating instructions and runtime data of an the App. Its structure is as follows:
   (1) **Header:** containing metadata, and a body which contains the majority of the data.
   (2) **String_ids:** recording the offsets of each string in the data area.
   (3) **Type_ids:** recording the string indexs for each type.
   (4) **Proto_ids:** recording the declaration strings of methods, return type strings, parameter lists.
   (5) **Field_ids:** recording the classes, type, and method name which belongs to.
   (6) **Method_ids:** recording the classes' name, declaration and the name of the methods and other information.
   (7) **Class_defs:** containing the type, inheritance hierarchy, access metadata, and other class metadata.
   (8) **Data:** containing data of each classes.
4) Static resource files that exist in the *res* folder mainly contain pictures, audios and layout files of the App.

B. ANDROID API AND ANDROID SDK

An Android API is an interface provided to developers to invoke system services, making App development more efficient and convenient. Here are a few Android APIs related to this paper.

TABLE 1. WebView APIs' name and their functions.

Class Name	Method Name	Function
WebView	addJavascriptInterface	javaScript excute java code
	setWebViewClient	setWebViewClient
	getSettings	Get a settings manager
	loadUrl	Display WebView Content
	loadData	

TABLE 2. Some commonly used APIs for certificate validation.

Class Name	Method Name	Function
X509TrustManager	checkClientTrusted	Verify the client certificate
	checkServerTrusted	Verify the server certificate
WebViewClient	onReceivedSslError	Handling certificate errors

TABLE 3. Some APIs provided by Alibaba Cloud OSS service.

Class Name	Method Name	Function
OSSTokenCredentialProvider	init	Initializes the key of the connection
OSSClient	init	Create a connection client
	asyncGetObejct	The asynchronous interface
ClientConfiguration	setHttpDnsEnable	Sets whether DNS mode is turned on
	setUserAgentMark	Set up a custom user-agent

WebView is an Android component that displays webpages and supports the execution of JavaScript. And its APIs are designed to customize WebView for developers. TABLE 1 shows the WebView APIs and their functional description.

Certificate verification APIs are used for HTTPs communication process of Apps, through which developers can customize the verfification rules. TABLE 2 summarizes related APIs.

SDK is a set of tools provided by an organization, and are used to develop Android applications. For example, *SDK* provided by Alibaba help developers use the company's services. TABLE 3 shows APIs in Alibaba OSS Cloud service *SDK*.

III. CASE STUDY

A. SELECTING VULNERABILITIES

1) IMPROPER CERTIFICATE VALIDATION *(ICV)*

HTTPs is an extension of the Hypertext Transfer Protocol (HTTP). It is used for secure communication over a computer network, and is widely used on Apps. In the use of such a network transmission protocol, the App verifies whether the certificate on the server side is a compliant certificate, so that both sides of the communication can be trusted. Apps use the *X509TrustManager* class to trust certificates. The weakness that the App improper certificate validation is described in CWE-295, when a certificate is invalid, it might allow an attacker to spoof a trusted entity by using a man-in-the-middle (MITM) attack. The weakness code is shown in Fig 1.

TABLE 4. Statistics of Vulnerabilities in the 100 Apps ('ICV' refers to 'Improper certificate validation, 'WBCVV' refers to 'WebView bypass certificate validation vulnerability', 'WRCEV' refers to 'WebView remote code execution vulnerability' and 'ACOCDV' refers to 'Alibaba Cloud OSS credential disclosure vulnerability').

CVE/CWE Id	Name	Total
CWE-295	*ICV*	81
CVE-2014-5531	*WBCVV*	86
CVE-2014-1939	*WRCEV*	28
CNVD-2017-09774	*ACOCDV*	20

```
class MyTrustManager implements
    X509TrustManager{
  /* The client certificate is not
      validated */
  public void checkClientTrusted(
      X509Certificate[]
      paramArrayOfX509Certificate, String
      paramString){}
  /* The server certificate is not
      validated */
  public void checkServerTrusted(
      X509Certificate[]
      paramArrayOfX509Certificate, String
      paramString){}
}
```

FIGURE 1. The sample code of Improper certificate validation(md5 of this App is 621fd4c77d310312a1916c12d6dc2c32).

We extract 100 Apps in the dataset, After manual analyzing, we find that 81 of these Apps with the weakness, as shown in Table 4. Hence we analysis this weakness in the paper.

2) WebView BYPASS CERTIFICATE VALIDATION VULNERABILITY *(WBCVV)*

The WebView component supports certificate verification when loading webpages with HTTPs protocol. As described in CVE-2014-5531, when an App is displaying data through WebView, at the same time, the function of verifying the validity of the certificate in this component does not work. In this case, the App will be vulnerable to man-in-the-middle attacks, And attacks can obtain sensitive information via a crafted certificate. As shown in Fig 2, the WebView is used to load the webpage, when the certificate verification fails, it still allows the data exchange between the App and the server side of the crafted certificate. Table 4 shows that we find 86 Apps in 100 have this vulnerability, so it is worthwhile to analyze the vulnerability in detail.

3) WebView REMOTE CODE EXECUTION VULNERABILITY *(WRCEV)*

According to Google [25], there are 2 billion active devices on Android monthly, and close to 10% are running in Android 4.3 or lower versions and more than 4% are running in Android 4.1 or lower versions. That indicates that JS-to-Java interface vulnerabilities may still affect hundreds of millions of users. Table 1 describes some of the APIs for the WebView component. The *addJavascriptInterface(Object object, String name)* function provides a way

```
WebView.setWebViewClient(new WebViewClient
    (){
@Override
public void onReceivedSslError(WebView view
    , SslErrorHandler handler, SslError
    error) {
   super.onReceivedSslError(view, handler,
       error); /* When the certificate
       verification fails, no exception
       handling is performed. Normal data
       interaction between App and server.
       */
  }

}
```

FIGURE 2. The sample code of *WBCVV*(md5 of this App is 28ac441942de453dd3c5756b2b223f9d).

```
class JsObject{
@JavascriptInterface
public String toString(){
   return "injectedObject";
   }
 }
 /* enable JavaScript execution */
WebView.getSettings().setJavaScriptEnabled(
    true);
/* JavaScript code can invoke JsObject via
    "injectedObject" */
WebView.addJavascriptInterface(new JsObject
    (), "injectedObject");
WebView.loadData("", "text/html", null);
WebView.loadUrl("javascript:alert(
    injectedObject.toString())");
```

FIGURE 3. The sample code of *WRCEV*(md5 of this App is ef400edb1ece62a44b97edabfdc5e076).

to bridge between JavaScript and Java. It is equivalent to inject a Java object into the environment of JavaScript. So the JavaScript code running in the WebView can call all the public methods of the Java object by the object name. However, in this scenario, proper security measures are not used, which can lead to *WRCEV* as described in CVE-2014-1939, an attacker can fake a malicious JavaScript script to attack an App. The code shown in Fig 3 that uses this Java-to-JavaScript, but it does not have security restrictions, so it causes the vulnerability. Table 4 shows that the vulnerability exists in current App Stores, and we should analyze why this vulnerability still occupies a high proportion.

4) ALIBABA CLOUD OSS CREDENTIAL DISCLOSURE VULNERABILITY *(ACOCDV)*

ACOCDV is a new vulnerability with wide influence in 2017. Alibaba cloud Object Storage Service (OSS) is a massive, safe and highly reliable cloud storage service provided by Alibaba. As shown in Fig 4, when developers use Alibaba cloud OSS service framework by the *SDK*, sensitive information, such as *AccessKey* and *AccessKeySecret*, is directly written into the code (for data protection, we perform sensitive information erasure processing). An attacker can easily

```
private void c()
{
this.b = "ywapp";
/* AccessKey and AccessKeySecret are
    hardcoded in the App, it causes
    sensitive information disclosure */
com.alibaba.sdk.android.oss.common.auth.
    OSSPlainTextAKSKCredentialProvider v3_1
     =
new com.alibaba.sdk.android.oss.common.auth
    .OSSPlainTextAKSKCredentialProvider("
    LTAImqvHSk9*****2", "
    uSjHPix26qWaQWnxoZdQsbS*****sn");
this.a = new com.alibaba.sdk.android.oss.
    OSSClient(com.ewuapp.view.base.BaseApp.
    a(), "http://oss-cn-shenzhen.aliyuncs.
    com", v3_1, v2_1);
return;
}
```

FIGURE 4. The sample code of *ACOCDV*(md5 of this App is 0aa8d5d2e46902509d4343077c14d15e).

get sensitive information through reversing the App, it causes data leakage from developers on cloud services. The data in Table 4 shows that the vulnerability the existence rate of the vulnerability is 20%, so the vulnerability of data leakage should be taken seriously.

B. CASE STUDY 1-WebView REMOTE CODE EXECUTION vulnerability(WRCEV)

1) VULNERABILITY TO EXPLAIN

java/android/webkit/BrowserFrame.java in Android 4.4 and earlier uses the *addJavascriptInterface* in conjunction with creating an object of *SearchBoxImpl* class, which allows attackers to execute arbitrary Java code by leveraging access to the *SearchBoxJavaBridge_* interface at certain Android version.

An App in the above environment, when the WebView component uses the *addJavascriptInterface(Object object, String name)* API to establish the connection between JavaScript and Java without security restrictions. JavaScript in the webpage not only has permission to call Java object's methods in its runtime environment, but to execute Android system commands.

The above description of the vulnerability is summarized that when the App runtime environment belongs to Android version 4.4 or below, if the WebView *addJavascriptInterface(Object object, String name)* API is used in the App and the developer has not added security restrictions, an attacker can build malicious JavaScript code perform instruction execution on the App.

2) VULNERABILITY POC

(1) File MD5: cdc18db0cad812225d8412be4e27182b

(2) version: 4.2

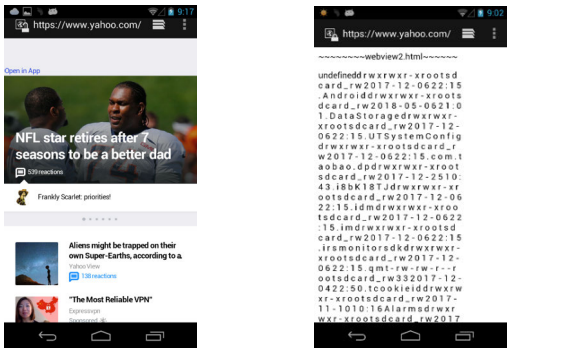
(3) Exploit: The App supports running under Android version 4.4. Fig 5 shows the code that causes the vulnerability in the App, it uses the WebView component to display a

TABLE 5. The functions to remove *WRCEV*.

Function	Description
@searchBoxJavaBridge	remove Javascript Interface ("searchBoxJavaBridge_");
@accessibility	remove Javascript Interface ("accessibility");
@accessibility Traversal	remove Javascript Interface ("accessibility Traversal");

```
public void loadUrl(String arg5, Map arg6,
      a b) {
    Bitmap v3 = null;
    if(this.mMainView != null) {
       this.mPageLoadProgress = 5;
       this.mInPageLoad = true;
       /* make the code in JavaScript
            call the b object in Java by
            the string a */
       this.mMainView.
            addJavascriptInterface(b, "a")
            ;
       this.mMainView.loadUrl(arg5, arg6)
            ;
    }
}
```

FIGURE 5. The vulnerability code of WebView Remote Code Execution.



(a) A web page shown by WebView loading a url (b) Get information for sdcard by embedding malicious code in JavaScript

FIGURE 6. Get the sdcard information by using *WRCEV*.

```
String endpoint = "http://oss-cn-hangzhou.
    aliyuncs.com";
/* init OSSCredentialProvider by accesskey
    and secretkey */
OSSCredentialProvider credentialProvider =
    new
        OSSStsTokenCredentialProvider("<
            AccessKeyId>",
                "<SecretKeyId>");
OSS oss = new OSSClient(getAppContext(),
    endpoint, credentialProvider);
```

FIGURE 7. The demo code provided by documentation of Alibaba cloud OSS.

webpage. Also, it uses *addJavascriptInterface(Object object, String name)* API to make the JavaScript code can call the *b* object by string *a* method in the called *HTML* page. However, this App does not limit the vulnerability through some protection methods, so the App has the vulnerability.

We install the App on an Android emulator, we open the browser to tap an url, such as https://www.yahoo.com. As is shown in Fig 6a, the App correctly displays the page passed by the url. Viaing a MITM attack, we forged malicious JavaScript code into the webpage. Malicious JavaScript code executes the command *"ls - al /MNT/sdcard/"* to read file information in *sdcard*. As shown in Fig 6b, malicious JavaScript normally runs on the vulnerable App, which reads the file information from the *sdcard* directory of the Android device. It indicates that this vulnerability exists and can be exploited in the App.

3) INSIGHT

When developers use the *addJavascriptInterface (Object object, String name)* API to call a web page. As summarized in Table 5, the functions of removing the risk interface should be added to limit the permission of JavaScript code. They can eliminate the *WRCEV* in Apps.

C. CASE STUDY 2-ALIBABA CLOUD OSS CREDENTIAL DISCLOSURE VULNERABILITY (ACOCDV)

1) VULNERABILITY TO EXPLAIN

Alibaba Cloud Object Storage Service (OSS) is a secure and reliable cloud Storage Service provided by Alibaba. When the service is used in an Android App, the APIs provided by the SDK of the service need to be called in the App. However, developers usually simply copy the official sample code and don't consider the security weekness of programming. Fig 7 is the demo code provided by the Alibaba cloud. Attackers can obtain this set of credentials by using the OSS management tool to get the data in the service easily. It will result in information leakage and constitute a serious security weekness.

2) VULNERABILITY POC

(1) File MD5: fea5226fdf7a1a1ed95d6c24a6e30f06

(2) version: 4.37

(3) Exploit: In order to facilitate the manual analysis and reading of the source code of the App, we use the Jeb [26] tool to reverse the App when manually verifying the vulnerability. Because the App is packed, we use our unpacking tool [27] to get the source code of the App. As shown in Fig 8, it is the source code which is obtained after unpacking the App.

Fig 9 is the code which contains sensitive information in the App. In *onCreate* method, we find construct the object Alibaba cloud storage service initializes the object. When constructing an object, this method directly invoke the *accessKeyId* and *accessKeySecret*, and both of these sensitive values are hard coded by the developer in the program, thus it causes the weakness of sensitive information.

```
import com.mbaobao.tools.WebViewBridgeUtil;

public class MBBActivityAct extends
    BaseActivity {
   class com.mbaobao.activity.
       MBBActivityAct$1 extends Handler {
      com.mbaobao.activity.MBBActivityAct$1
          (MBBActivityAct arg1) {
         MBBActivityAct.this = arg1;
         super();
   }
   public void handleMessage(Message arg3)
       {
    switch(arg3.what) {
      case 0: {
         MBBActivityAct.this.
             pullRefreshWebview.setMode(
             Mode.PULL_FROM_START);
         break;
   }
  final class InJavaScriptLocalObj {
      InJavaScriptLocalObj(MBBActivityAct
          arg1) {
         MBBActivityAct.this = arg1;
         super();
   }
  @JavascriptInterface public void
      updateShareContent(String arg6) {
    MBBLog.d("MBBActivityAct", "[
        updateShareContent] html = " + arg6
        );
      if(arg6 != null) {
         try {
         Uri v1 = Uri.parse(arg6);
         if(!StringUtil.isEmpty(v1.
             getQueryParameter("title"))) {
         MBBActivityAct.this.shareTitle =
             v1.getQueryParameter("title");
   }
```

FIGURE 8. Get its source code through decompiling the unpacked App.

3) INSIGHT

Inorder to prevent data leakage of sensitive information of *accessKeyId* and *accessKeySecret*. Developers should encrypt sensitive information when using the Alibaba OSS SDK.

D. VULNERABILITY CHARACTERISTICS

As shown in TABLE 6, the characteristics of vulnerabilities are summarized after a comprehensive case study of each vulnerability. In addition, after a lot of manual analysis of Apps of above vulnerabilities. We divided the four vulnerabilities into three categories, and generalized the analysis methods and characteristics of each type of vulnerability based on the four specific vulnerabilities. The details are described as follows.

1) IMPROPER CERTIFICATE VALIDATION *(ICV)*

An App uses the method of *checkServerTrusted*. No certificate verification operation is performed in it, or in this method, only the expiration time of the certificate is verified, and no other information is verified.

```
static {
      Config.endpoint = "http://oss-cn-
          hangzhou.aliyuncs.com";
      /* Sensitive information is hardcoded
           in the initialization of object
          */
      Config.accessKeyId = "fEQ1ry61***
          C6HEm"; /* accessKeyId */
      Config.accessKeySecret = "Z402SQ***5
          FvAL9h6QzIWsRTUjP3"; /*
          accessKeySecret */
      Config.Bucket = "maibaobao";
   }
protected void onCreate(Bundle arg5) {
      this.oss = new OSSClient(AppContext.
          getInstance(),
      Config.endpoint,
      new OSSPlainTextAKSKCredentialPro
          vider(Config.accessKeyId,Config.
          accessKeySecret));
      this.updateView();
   }
```

FIGURE 9. The vulnerability code in the App.

2) WebView BYPASS CERTIFICATE VALIDATION VULNERABILITY *(WBCVV)*

An App uses the WebView API for certificate verification, but the method *onReceivedSslError(WebView view, SslErrorHandler handler, SslError error)* which is for handling the exception certificate does not perform any processing. This causes the abnormal certificate to pass the verification.

Vulnerability in overridden method (OM). The functions that lead to the above two vulnerabilities have common characteristics in analytical methods. These are subclass methods provided by the Android system to developers to customize the strong validation logic, and the parent class methods they inherit have the vulnerable logic in default. Developers don't strictly follow the security specifications to customize the security policy code. The analysis method for this type of vulnerability is described in TABLE 6. An App has a overridden method that is to do the security validation, and its inherited parent method is a weak logic in default. If the method of the App is not written with strong verification, then the App is vulnerable.

3) WebView REMOTE CODE EXECUTION VULNERABILITY *(WRCEV)*

There are three elements that can cause an App to have this vulnerability. First, the App needs to support to run on Android version 4.4 or below. Second, the App uses WebView's API *addJavascriptInterface(Object Object, String name)* to allow JavaScript to run Java. Finally, the App does not add an API as shown in TABLE 5 to remove the weekness interface in the context of using the above API.

Vulnerability of using unsafe APIs (USS). For an App, it uses a potentially vulnerable API, and the developer does not perform codes to protect against the vulnerability in the

TABLE 6. Vulnerability characteristics with case study('OM' refers to 'Overridden method', 'USS' refers to 'Use unsafe settings', 'DLSI' refers to 'Data leakage of sensitive information'; 'ICV' refers to 'Improper certificate validation, 'WBCVV' refers to 'WebView bypass certificate validation vulnerability', 'WRCEV' refers to 'WebView remote code execution vulnerability' and 'ACOCDV' refers to 'Alibaba Cloud OSS credential disclosure vulnerability').

Category	Vul	Description	Analytical method
OM	ICV	An App inherits the *X509TrustManager* class and overrides *checkServerTrusted*. If the method does not perform security verification, this vulnerability exists in it.	Checking the override method body, whether the developer has correctly written the code according to the security specifications.
	WBCVV	An App overrides the *onReceivedSslError* method of the WebView. If the overridden method body does not make a security judgment and directly executes the default all-trusted method of *handler.proceed()*, this vulnerability exists in it.	
USS	WRCEV	The minimum supported version of the Android SDK that the App supports is less than 17. It uses the *addJavascriptInterface* method of WebView. If there is no way to add a secure interface as shown in TABLE 5, then it exists the vulnerability.	For an App, confirm the suspected API and its context, and determine the version of Android that the App supports. Synthesize the existence logic of the vulnerability to detect the existence of the vulnerability.
DLSI	ACOCDV	Check whether the App uses the ALibaba OSS SDK method of *OSSPlainTextAKSKCredentialProvider*, and trace objects of *AccessKeyId* and *SecretKeyId*. If these are hard-coded in the App, the vulnerability exists.	Find the method which an App uses sensitive information, trace the variables of sensitive information, and confirm whether the vulnerability exists by hard-coding them.

calling link and the context of it. Because some vulnerabilities are related to the Android environment, API level required for Apps is also one of the elements to identify the vulnerability. We can combine the above elements to determine whether the vulnerability exists.

4) ALIBABA CLOUD OSS CREDENTIAL DISCLOSURE VULNERABILITY *(ACOCDV)*

For an App that uses the API *OSSPlainTextAKSKCredentialProvider* provided by the Alibaba OSS *SDK* to create credential information with server, its *secretKey* needed in the method is hard coded in the program. This causes the existence of information disclosure vulnerability.

Data leakage of sensitive information (DLSI). The analysis and characteristics of this category vulnerabiliy is shown in TABLE 6. An App has an invoke method that uses sensitive information. The object of the sensitive information is traced to determine whether it is hard-coded in the App. If it is hard-coded in it then the vulnerability exists.

IV. VulArcher-A TOOL TO DETECT VULNERABILITIES

A. SYSTEM OVERVIEW

As shown in Fig 10, the working process of VulArcher can be divided into the following steps:

1) DECOMPILATION

For an App, VulArcher decompresses it to obtain the *classes.dex* file which contains the source code, the *AndroidManifest.xml* file which contains component information, permissions and the resource files. VulArcher reverses the *classes.dex* based on Androgurd [28]. Through this process, It obtains the classes and methods in the source code of the App. Also it decompiles the *AndroidManifest.xml* file to obtain registered component information and configuration information.

2) PACKER IDENTITION

VulArcher recognizes whether an App is packed according to formula (1) and (2). *AC* indicates the number of all components registered by the App, and *CC* indicates the number of classes in *classes.dex* which can obtain from the *AC*. Because the packed App almost hides all components and other logic codes, *classes.dex* does not contain these information. Some packing methods have signature files, ie. *libexe*.so* and *libexecma**.so*. In order to further clarify packing methods, VulArcher uses the fingerprint to identify the detailed packing type and version of the App.

$$F = \frac{CC}{AC} \tag{1}$$

$$App \begin{cases} unpacked, & if\ F \geq 0.8 \\ packed, & otherwise \end{cases} \tag{2}$$

3) UNPACKING

At present, more and more Apps are packed. Static analysis can not directly get the real code of the packed Apps. This causes static detection to fail to analyze vulnerabilities of such Apps. Therefore, the analysis of packed Apps should be carried out by unpacking method, for extracting the original code of Apps. The unpacking method used in this system *DexX* [27] is a result of our previous research. It can handle the six packers, such as Ali [29], Baidu [30], Bangcle [31], Tencent [32], Qihoo 360 Mobile [33], and ijiami [34]. This paper does not elaborate too much for the method.

4) BUILDING TAINT PATH

VulArcher creates a taint control flow graph(TCFG) of a *classes.dex*, the algorithm is shown in Algorithm 1. VulArcher creates a control flow graph(CFG) of the *classes.dex*. the set V of p nodes in CFG is shown in formula (3). Each v_k contains the package name(pkg_k), class name(c_k) and method(m_k). As shown in formula (4), the set IA of m interested APIs is a subset of V. For each iA_j, we follow the heuristic method to find its taint path tp_j, it's shown in formula (5). V' is a subset of V, the points in V' are related to the control flow of iA_j. If v_k has a control flow to v_w,

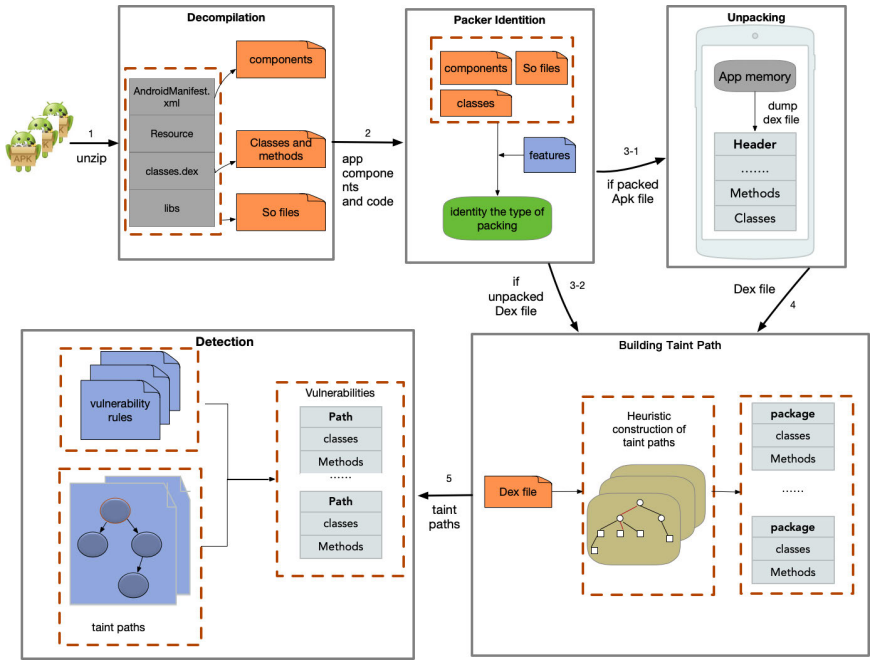


FIGURE 10. The overview of VulArcher, it describes the modules of the overall tool, as well as the workflow.("Y" indicates that the App is packed and "N" indicates that it is unpacked).

Algorithm 1 The Algorithm of Building TCFG

1: **Input:** IA, CFG {IA the set of interested APIs.}
2: **Output:** TCFG
3: INITIALIZE TCFG $= \emptyset$
4: **for** each $iA_j \in IA$ **do**
5: $tp_j \Leftarrow HeuPath(CFG, iA_j)$ {A heuristic method to find taint path tp_j of iA_j.}
6: $BuildG(TCFG, tp_j)$ {Building the TCFG with tp_j.}
7: **end for**
8: **return** TCFG

we think there is a taint control flow relationship from v_k to v_w <v_k, v_w>, we denote the set of all the above relationships as E'. As shown in formula (6), TCFG is composed of all $tp_j(j = 1, \ldots, m)$.

$$V = \{v_k = (pkg_k, c_k, m_k) | k = 1, \ldots, p\} \tag{3}$$

$$IA = \{iA_j = (pkg_j, c_j, m_j) | j = 1, \ldots, m, and\ m \leq p\} \tag{4}$$

$$tp_j = \{(V'_j, E'_j) | < v_j k, v_j w > \in E'_j, \quad V'_j = \{v_j k | k = 1, \ldots, q\}\} \tag{5}$$

$$TCFG = \{tp_j | j = 1, \ldots, m\} \tag{6}$$

5) DETECTION

VulArcher uses the rules of above vulnerabilities and the TCFG of an App generated in the ***Building Taint Path*** for vulnerability judgment. In order to detect vulnerabilities faster, we propose a heuristic vulnerability search algorithm as described in Algorithm 2, so that we can avoid matching vulnerability rules of all App code. For tp_j, VulArcher extracts its context slices(cs_j). R is a set of vulnerability rules. The input to function $f(.)$ is R and CS. If the m matches the vr, the App is commented with the vulnerability which vr represents.

$$CS = \{cs_j | j = 1, \ldots, r\} \tag{7}$$

$$R = \{vr_1, vr_2, \ldots, vr_n\} \tag{8}$$

$$f(R, CS) \begin{cases} Vul, & if\ vr_i\ is\ matched\ in\ CS \\ notVul, & otherwise \end{cases} \tag{9}$$

We just need to find interesting points directly in the paths of the vulnerabilities. In this way, the entire vulnerability search process is fast and accurate.

B. EXTRACT SUSPICIOUS CODE SEGMENT

An App contains many classes and methods, if we iterate through each method to search vulnerabilities and weaknesses, it will certainly affect the efficiency of detection. So we first built the TCFG of the App, recording classes and methods that contain suspicious interesting paths. When we are looking for vulnerabilities, we can use the heuristic-based approach described in Algorithm 2 to find out if a vulnerability exists. The *interestSet* in the algorithm represents all sensitive APIs and methods that may cause vulnerabilities in the App. *FR* represents a collection of rules for vulnerability fixes and *TR* represents the set of rules that trigger the vulnerability. The algorithm first obtains a suspected vulnerable point in *interestSet*, then in the already constructed TCFG, it constructs the relevant slice content of all objects and

Algorithm 2 A Heuristic Vulnerability Search Algorithm

Input: *interestSet*, *FR*, *TR* {*interestSet* in the algorithm represents all sensitive APIs and methods that may cause vulnerabilities in the App. *FR* represents a collection of rules for vulnerability fixes and *TR* represents the set of rules that the vulnerability triggers.}
Output: *output(path,contextSlice)* {This stores the detailed code snippet of the vulnerability and the path where the vulnerability is located.}
INITIALIZE *output(path,contextSlice)* = ∅
{*FindPath* can find the caller paths in the APK through a function that is suspected of a vulnerability.}
for each *ti* ∈ *InterestSet* **do**
  path ⇐ *FindPath(TCFG,ti)*
  contextSlice ⇐ *GetContextSlice(path)* {Geting detailed function information by the path.}
  if (*FR* ≠ ∅) *and* (*TR* ≠ ∅) **then**
    if *HeuSearch(contextSlice,TR)* == True *and* *HeuSearch(contextSlice,FR)* == False **then**
      output(path,contextSlice)
    end if
  end if
  if *FR* ≠ ∅ **then**
    if *HeuSearch(contextSlice,FR)* == True **then**
      continue
    else
      output(path,contextSlice)
    end if
  end if
  if *TR* ≠ ∅ **then**
    if *HeuSearch(contextSlice,TR)* == True **then**
      output(path,contextSlice)
    else
      continue
    end if
  end if
end for
return *output(path,contextSlice)*

variables in the point, that is, *contextSlice*. It uses a heuristic search algorithm to search in *contextSlice*. If there is a rule for repairing the vulnerability in the slice, then the App does not have the vulnerability. Otherwise, if it finds the rule for triggering the vulnerability in slice through the search algorithm, then it records the complete path information and slice of the vulnerability of the App.

C. VERIFY METHOD AND WORKFLOW

Nowadays there are more and more vulnerability detection tools. To some certain extent, they can detect some vulnerabilities and weaknesses. But they do not provide developers with some way to reproduce vulnerabilities. This is the reason why more and more vulnerabilities are ignored by developers, as a result, many Apps still have vulnerabilities that have been reported for a long time. In this paper, we provide semi-automated verification methods and workflow for the vulnerabilities and weaknesses we analyzed. This work can verify the detection results of VulArcher and provides readers with clear methods of vulnerabilities' verifications.

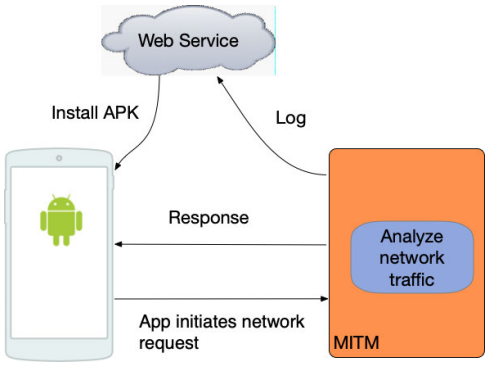


FIGURE 11. Attack architecture for Certificate validation vulnerability.

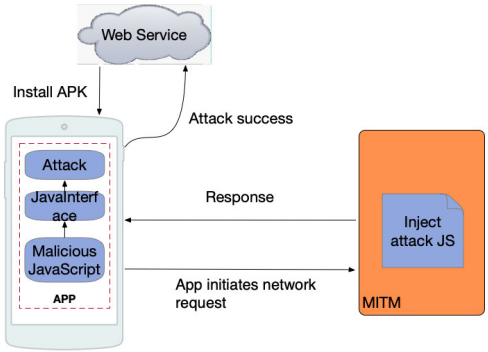


FIGURE 12. Attack architecture for *WRCEV*.

1) A VERIFICATION METHOD FOR MAN-IN-MIDDLE ATTACK VULNERABILITY CAUSED BY WEAK CERTIFICATE VERIFICATION

Fig 11 depicts the attack architecture for this weakness. First, we set up the environment to enable *Burpsuite* [35] to block all network traffic from mobile. We write a script based on *Burpsuite*, which is mainly responsible for analyzing the traffic information of mobile, extracting the Host and body and sending it to the server. The *web service* module is mainly responsible for auto installing Apps and recycling the results of analyzing network traffic, also, determining whether the App is attacked by MIMT automately.

2) A VERIFICATION METHOD FOR *WRCEV*

As shown in Fig 12, our verification method is mainly to inject malicious JavaScript code in to the App's network traffic by MIMT. We use *Burpsuite* to block all traffic sent by the mobile and injecting the written malicious JavaScript code in the response. If the JavaScript can successfully execute malicious code, the code will send a record to the *web service* to record whether the App exists the vulnerability.

TABLE 7. The distribution of three types of Apps from January to August 2018('S' refer to 'Shop', 'F' refers to 'Financial', 'B' refers to 'Browser' and 'T' refers to 'Total').

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug
S	358	213	391	526	435	455	504	72
F	291	224	376	557	467	475	706	64
B	8	4	3	4	19	9	14	2
T	657	441	770	1087	921	939	1224	138

TABLE 8. Three types of Apps' size distribution(Mb)('S' refers to 'Shop', 'F' refers to 'Financial', 'B' refers to 'Browser' and 'T' refers to 'Total').

	0-5	5-10	10-15	15-20	20-25	25-30	30-35	35-40	40-45	45-50	≥50
S	418	677	510	492	235	194	130	92	55	35	116
F	511	736	685	408	275	177	101	65	58	36	108
B	21	15	7	10	2	2	1	2	1	2	0
T	950	1428	1202	910	512	373	232	159	114	73	224

TABLE 9. All types of vulnerability of the Apps test results manually('N' refers to 'Numbers of Samples' and 'NFP' refers to 'Numbers of False Positives';'ICV' refers to 'Improper certificate validation, 'WBCVV' refers to 'WebView bypass certificate validation vulnerability', 'WRCEV' refers to 'WebView remote code execution vulnerability' and 'ACOCDV' refers to 'Alibaba Cloud OSS credential disclosure vulnerability').

Vulnerability	N	NFP
ACOCDV	100	0
WRCEV	100	13
WBCVV	100	5
ICV	100	13

3) A VERIFICATION METHOD FOR *ACOCDV*

We use the *Jeb* to reverse the App, and we look for the code module initialized by OSSClient in the reverse smali code. We find out whether initialize method by using plaintext *accessKey* and *secretKey*. We get these keys, and use the OSSUtils to connect the service.

V. EXPERIMENT

Our experiments are performed in the following environment: The CPU is e5-2640v4, the memory is 32G, system is Linux 64 and python 2.7.

A. COLLECT ORIGINAL SAMPLES

In this paper, the main types of Apps we focus on are financial, shopping, and browser. We downloaded a total of 6177 Apps from the Baidu [36], Wandoujia [37], Qihoo [38] and Huawei [39] App Stores, which were released from January to August 2018. Of these, 3,160 Apps are in the financial category, 2,954 Apps are in the shopping category, and the remaining 63 Apps are in the browser category. Table 7 shows the distribution of three types of Apps from January to August 2018. It includes classification by sample category of Apps, classification by time. Table 8 shows the distribution of three types of Apps' size.

B. ANALYZE THE RESULTS

TABLE 10 shows the details of the test results of all the Apps. We divide the samples vulnerability type into two categories according to whether or not is packed. It can be seen that even if the App is packed, vulnerabilities cannot be fixed. Through the results, *WBCVV* and *ICV* have a high ratio, because most of the Apps' network communication protocols now use HTTPs, developers use APIs to develop these functions, and ignore the security of certificate verification for the convenience of the development process. *WRCEV* is the least, there may be a direct relationship with the current Apps that rarely supports version of Android 4.4 and below.

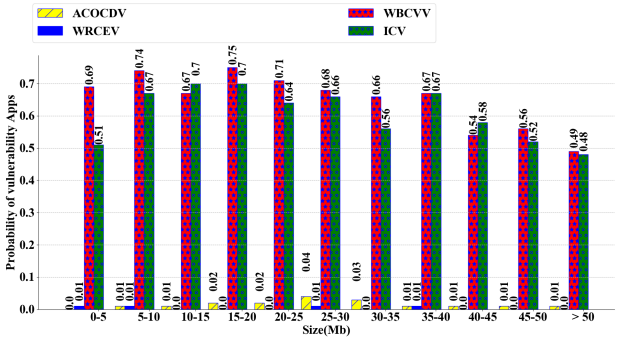


FIGURE 13. The relationship between sizes of Apps and vulnerabilities.('ACOCDV' refers to 'Alibaba Cloud OSS credential disclosure vulnerability', 'WRCEV' refers to 'WebView remote code execution vulnerability', 'WBCVV' refers to 'WebView bypass certificate validation vulnerability' and 'ICV' refers to 'Improper certificate validation').

Finding 1: Although the *WRCEV* was first reported in 2014, there is still a 4.7% Apps in the currently released browser Apps.

Finding 2: When the JS-to-Java reflection function in WebView is used in Apps, it is able to strictly limit the Android version to be greater than 17, and use the interface function shown in TABLE 5.

Manually Verfify. We randomly select samples with and without vulnerabilities in the detection results. We manually analyze those samples. For those which need a user account to use, and these accounts are not free to register externally, these are not the scope we consider. The analysis results are shown in TABLE 9, it is showed that VulArcher detects *ACOCDV* with high accuracy.

Finding 3: When we do the MITM, we find that many Apps not only do not carry out certificate verification, but also transmit sensitive data such as user name and password in plaintext. It is very easy to cause the leakage of sensitive information. So it is necessary to encrypt sensitive information.

Finding 4: The detection accuracy for *ACOCDV* can reach 100%, and for *WBCVV* can reach 95%. Also the accuracy is lightly lower and can reach 87% for the other two vulnerabilities. VulArcher can help developers detect vulnerabilities before Apps are released. It can avoid releasing Apps with the above vulnerabilities.

The irrelevance of vulnerabilities and sizes of Apps. Fig 13 shows the number of vulnerabilities in Apps with different sizes. It can be seen that the same category of

TABLE 10. Summary table of all sample detection results.('P' refers to 'Probability of vulnerability(%)'; 'ICV' refers to 'Improper certificate validation, 'WBCVV' refers to 'WebView bypass certificate validation vulnerability', 'WRCEV' refers to 'WebView remote code execution vulnerability' and 'ACOCDV' refers to 'Alibaba Cloud OSS credential disclosure vulnerability').

Packed	Vulnerability	Vulnerability Apps	Summary	P
Yes	*ACOCDV*	70	79	1.278
No		9		
Yes	*WRCEV*	19	28	0.453
No		9		
Yes	*WBCVV*	2851	4285	69.370
No		1434		
Yes	*ICV*	2594	3936	63.720
No		1342		

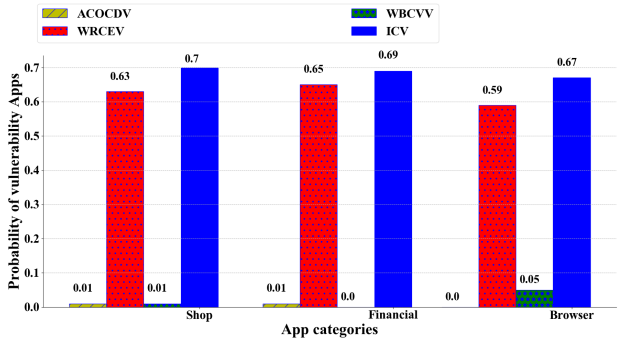


FIGURE 14. Distribution of vulnerabilities in different category of Apps.('ACOCDV' refers to 'Alibaba Cloud OSS credential disclosure vulnerability', 'WRCEV' refers to 'WebView remote code execution vulnerability', 'WBCVV' refers to 'WebView bypass certificate validation vulnerability' and 'ICV' refers to 'Improper certificate validation').

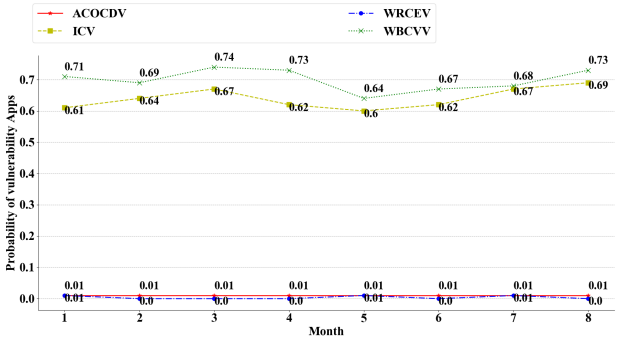


FIGURE 15. Distribution of vulnerabilities per month in 2018. ('ACOCDV' refers to 'Alibaba Cloud OSS credential disclosure vulnerability', 'WRCEV' refers to 'WebView remote code execution vulnerability', 'WBCVV' refers to 'WebView bypass certificate validation vulnerability' and 'ICV' refers to 'Improper certificate validation').

vulnerabilities have almost the same probability distribution in different sized apps. Both *WBCVV* and *ICV* have a high percentage in Apps of different sizes.

Finding 5: The number and the type of vulnerabilities in Apps are independent of the size of them.

The relationship between vulnerabilities and categories. Fig 14 shows the probability of vulnerabilities for each of the categories of Apps. *ACOCDV* does not exist in the browser Apps, because they do not need to use this service to store a large number of images. The reason that the percentage of *WRCEV* in these Apps is slightly higher than the other two types of Apps is most of these Apps use the WebView component and many of them support for the Android 4.4 version.

Finding 6: More and more Apps face the vulnerability of MITM, because developers do not validate certificate for convenience. So certificate validation must be done strictly in the development.

Finding 7: Although Alibaba OSS SDK has released new instructions to circumvent this credential information leak [40], results in Fig 14 shows that many Apps still have this vulnerability. This means that Apps should be timely updated the SDKversion to fix the vulnerability.

The trend of the vulnerability. Fig 15 shows the trend of probability of vulnerabilities for Apps in each month from January to August in 2018. It can be intuitively reflected that there are vulnerabilities and weekneses in the monthly release of Apps, and the number of vulnerabilities does not decline over time.

Finding 8: As shown in Fig 15, the trend of vulnerabilities tends to be in a stable state. This shows that various vulnerabilities still exist in Apps and developers do not mitigate these vulnerabilities.

C. VulArcher PERFORMANCE

Efficiency. VulArcher takes an average of one minute to analyze an App. However, if the analysis App is packed, then the time to unpack it is an average of 50 seconds.

Computing cost. For an App, the memory consumption is 400M, CPU utilization is 40%.

Scalability. VulArcher supports the rules of vulnerabilities to detect them. Hence, for a new vulnerability, VulArcher only need to konw the vulnerability rule to complete the detection of the new vulnerability.

VI. DISCUSSIONS AND LIMITATIONS

We download the Apps to analyze the four types of vulnerabilities, but the Apps was two years ago. We should use the latest data for analysis. However, the latest Apps compared with the previous ones, the Android system versions they support has not been updated much, the APIs used in them has not been updated. Our findings are not biased.

TABLE 11. Details of 98 Apps with Webview bypass certificate validation risk.

MD5	Attack	MD5	Attack	MD5	Attack
03667ef03069bb720167cff22d55ee0d	√	05bca1351275470b6809cd51ed9f4067	√	05eee66c453e786e0f9c80330111a7cf	√
0eb49eeb5db706d7dae878e53f85d054	√	0ebc730d9efc09f2224a0f68513402b9	√	1239e063719a90bd319c837748eecdb7	√
15ae7d3a2508ec6abfcfaf8301c42ad6	√	1aeacb57f4057e43b29efc870bcbca29	√	288ae8374be5d2156ed9789fdf177f16	√
2a66b2d3b3d40feb2160a4a9e0bb28dd	√	2c844da2a9d8253a37a57b6bc3dc590b	√	2e4077b9e2bc76ae8604f37b31d46bb7	√
34944702616984d5fb53c132b66d364c	√	3547c1bac8596e7c6168f737bd20a7d1	√	359f8a0a1040e0640184f01619dc94c4	√
36350a840b455d3c40774a0211210f06	√	3a2896bdb086ec80f6303e383c6d5ce4	√	3ab9d03b8b890009f2b67f18ea74bef5	√
3c408f7ea173432189b40edd384338a9	√	3db1f22f88cf7e810c787011c79e275a	×	3e2e999d418251d165752242a54f552f	√
3ed5685744454814266 90acdb8924f04	√	40732db6efdb2bf40586015b2a6b70ce	√	412d8e0848bc87e9248e1ad1a5b9784f	√
42465f7177bfd75b16f092a467e2f094	√	4347fccc027da22f8518dbbf078deba8	√	439ad521e7fe12bc8bf35e3d4b1adc06	×
43f5989217d4ec0fe50334dab0a298fa	√	47549d8973211066f47970fcc3915096	√	49ea2ed60bf422ce273adb157e9972fd	√
4bcc07197595550b8eb533fb62f2e8c8	√	4db9965ff4360ec5c49e3d836692dbde	√	4f58c6acf93b7ae1ecae8592ebaa5fda	√
4fb419b60d8fec7f2ca70c536bdf09f3	√	5286b83ca1d3a6dc14b3345822f8ba93	√	560dc1cc8b0b07857371f042d1c270b6	√
59c5c36685bb097d84571a0bc30a1f4c	√	5e88e86c97d39996ee8b4da0a7d8e047	×	5f2e2b84579599e4d03eac2998483af1	√
62e69b26d58ad25cabcf5fbbea429caa	√	64bc08f2a69ae2ed602cd603c7468fc4	√	66a5268bdb7678387a6753c4a09f19f2	√
6939fd27cc1f30ea91c65893395f598d	√	6a6ea16f24d73756b3330e0939ca39fd	√	6a873ff1ea2a59015c756b8b8b01cb57	√
6d74726f2a1081e49d27b03f06457439	√	6f063b799a1f0383dcd3a600ddf3d51d	√	7048a54195151a0f5c93563b8ab31468	×
73ab531aac1db8fccf53335be2fa5b43	×	7921391e747b4318c556cc4270a58a05	√	79d3f503b953f3ee463354ee4cf03193	√
7b8557c0697505fcca30f85423b9a341	√	7df75b0b9fc7b418aa248dda9c126daa	√	81f578ca3aa5daba83b4b648ea5bfbb4	√
84f9cf283a87381a087a6ff7e9d1c4c2	√	8780076e0fab05a6e3ccebd0b1d41506	√	8e9c5ce360aa24bc5a74b0e73de86280	×
9633f567840c68cc9b3aeed750961403	×	97940b5146552d572ab70f6a932453d0	√	9818ac8c3ce55dbf91cf200ff38c088a	√
99258f0c7c1767b6f8cfd6a73d2a7ed3	√	9c00c56d3a21dc8c937ffe9b9b10b545	√	9c168db8ebdcfa911b6f64b012f037cf	√
9e027a939e2b46560feefe4ca9b5c429	√	9e5862cf055b4ddaacd3d34708f02e05	√	a7abb6e0e9a4adf0c2d0a8591ca16720	√
a8454441452925877e8e536cf9f66ed85	√	aabae1c5b8b7e538270b39cd36e3be8b	√	ae2f7b6b139014fb38a9728bb24de9e9	√
ae674a9ab49f91dbd90f809c81afc48a	×	ae7ca835ab6f45905fb1e52bf9193bd4	√	b10ae69bf51bd6b8f553ff99bdc8a848	√
b298b9957ecd3a3bed476ca8ccd49142	√	b2ba16f9f79cf92e8cf73383fc94558f	√	b49cef2f16efe1fc4e8a6d6857b98f7d	√
b63e2e66aea4b08c0c6f354b3d2920d9	√	b95e7a960253db796020a1b3ab70ea4a	√	bed151614dd6fc48d8dfc72a6a364af4	×
c07db7e3743909285aa9edab18ec7c62	√	c2a11ec2985b6f162fd0996906dce0bf	√	c3773ad91efe83026ab713553a1f2d80	√
c7ef7e3939c26f01f4602464583cc73f	√	c957897642db39ee8ebf6775bf730aa3	√	cae49083d8360defd5e8c1fd8dd96cab	√
cbe4518e6f182e7758ff941b1b541cd3	√	d256317312f6038f5888bd7bb2b94e4b	√	d68d609ebcd133167811ae9cb8d9bd7a	×
d78ad3097a3b7fefe703036d4c8dd07f	√	d95aebb3378dc12373e2a26c853d1df2	√	d98939e981a1e206219284341df8eacf	√
dae97bc44d9c5c98076f24dd7b5933d1	√	e54498341f43eaed06cc77689e4f18bb	×	e8c84dd656c5fc4a90abf6717f02b436	√
ebac536db14abe1f32cfd4cc307f76f1	×	ed142036b504ba5c89b8903eee832324	√	edadabb981e983941d1f95c76059f8ec	√
ef93cbbad294019e8b1a86f023a798b3	√	f82050d7626a0d391d85f95a48d6c1a6	×	-	-

The vulnerability results detected by VulArcher contain both third-party components and the App itself. At present, we regard these two aspects as all the vulnerabilities of the App. If we want to distinguish the vulnerability results. We can organize a third-party component library, which contains information such as the package name of the component. The vulnerability results detected by VulArcher contains the package name and class name of a vulnerability, so we use the package name and the class name to go back to the component library to check whether the vulnerability belongs to the App itself or third-party components.

VII. RELATED WORK

The study of Android vulnerabilities can be divided into two aspects, one is about the Apps and the other is about the Android OS.

Li *et al.* [14] just conducted a detail analysis of the vulnerabilities that will occur in the music Apps. The music Apps that are analyzed mainly produce man-in-the-middle hijacking attacks when communicating with servers. Qian *et al.* [23] proposed a structure called app property graph (APG) for Android vulnerabilities. Based on this feature representation method, a detection tool is proposed for five common vulnerabilities. But they did not give workflows for the verification of the vulnerabilities. Chen *et al.* [41] manually analysed several OAuth providers for Apps and determined how differences between the Apps and browser environments lead to the OAuth vulnerability. Wu *et al.* [1] proposed a detection method for the confused deputy vulnerability in Android applications based on features of *AndroidManifest.xml* file and Control Flow Graph (CFG). Fang *et al.* [5] studyed an input validation vulnerability in Android inter-component communication. This kind of vulnerability is caused by the incomplete security verification mechanism of Apps developers. Yang *et al.* [10] used a combination of static and dynamic methods to analyze the App's dynamic loading vulnerability. This kind of vulnerability is caused by the insecure verification on the loaded executable file. Yang *et al.* [20] studyed about security issues caused by mixed postMessage *Origin Stripping Vulnerability*(OSV) in Apps. Ranganath and Mitra [42] proposed a detection tool for supporting large-scale Android vulnerabilities, they didn't conduct analysis method for vulnerabilities. Farhang *et al.* [43] focused on how different when vendors deal with Android vulnerabilities.

Other studies explored different vulnerabilities from we studied. Feng and Shin [44] analyzed some vulnerabilities related to *Binder*in Android, and proposed an automated tool for detecting this type of vulnerabilities from the analysis of

TABLE 12. Details of 99 Apps with Improper certificate validation.

MD5	Attack	MD5	Attack	MD5	Attack
0199d0463b6f6b570daa43712b665013	√	05806c8ab3e7650ed6da605bb0eea86e	√	097bc9a7e1eed2378c65a63dba97136a	√
0bcc69f5221828701dec39edfde520ba	√	0fc46e610646ae40757e08354ca02c52	×	124b0c3400d8ebe6dfdd9bba9feab2a7	×
13d420a1582f74067280872ea9db8ea9	×	159b059e4fb2c5229e7685abcddd9de6	√	159bf43fc40c936432a81fe474c8e5f5	√
1739fb415e137281c0f59eca0d126f28	√	18701801bbae946f6e21f869634ae9cd	√	1991000b3785943f109ce6d6bb4c1b91	×
1b6b13ae21abc461cc1a486fce85f4ff	×	1bee172a44fc9acf0b9754b7a76d1a88	√	1c807e4d48cef29f9b33beca3894891e	√
1e55831ce60fff73ef0d8a6908b0316a	√	219ab3a97a99ef58ec23ad525ab8c5c5	×	22e4fc053371d5bd4901fd084b0c4e91	√
237027436d778f9425ecd958709623b3	√	2523424567dd41e1e01f054b353f1d55	√	2716be5c817aa6f5fffc0611e85db07d	√
29671b09686e8ea5444c26076a1c8fe4	√	2c08ce8f303137aaaf2803107cdbd8b8	√	37ed08f61d9dabd6b3db4453eb176ffc	√
38c4cb85804dfae58b82b211b152a891	×	38d22def85f3c2801e51813ea10d092b	×	3feae5ad8fbcd69786165128b530e76e	√
49325776c4b11c36fb0e80dbb5aada3d	√	49f6038c5c3f75a166b3c8bf4167f4ed	√	4bf7790c67bc558c54e1fbff5cf498a3	√
4c24329cfda7a88407078019de1d380e	√	4d6aa2d8e76145a9388a5ca5583d3875	√	4fd2c6b948eb93fb34ae0442c5a7837d	√
4ff6a0cc1b56922d998cb4143d2792db	×	5273d770b7754948b1394818db62e6fb	√	5b0f112662aa274884097b0e83e7ee85	√
5dcc16e75f372a83879157bc9762bc20	√	5de8e8a4d8a4317c1aa2b71e65f2cfcd	×	621fd4c77d310312a1916c12d6dc2c32	√
656a340a8d682e0012e199f9bfc1b5aa	×	665261de4e402b68ab0fed8534f63818	×	6734e923b997919e75a1056c57ac846b	√
67718ebb4c91848ee20cccc1548bfc79	√	68a81b2eb3b32d32dca4f92970f15a2b	√	6fbc4be998363b6ad807bbdb1552dddb	√
707f764313cb4d3e15edb100468aa59c	√	72c1a89547c505777d08416535d9eb88	√	730d24f5dc9d62c9f68eb4f3cafea579	√
7491faf8e11065dda236358174bd304c	√	7733f8fb7a3cb23331cf804db4ca960f	√	781946faf82fe71ae9b760331d59fac9	√
7ac77a7e5601c17cf471d23617c56fff	×	7c467af6cbd6eeac512f935c7ff2da61	√	7ff91ef039f0c03170630681befb6df8	√
81ae433c84458bc9083de82458cee7bb	√	8d6995187eb061563c9aadb613206636	√	90eaaa4aac771f854824328cbcff0e2c	√
91a0e946294c4606f9b2f8bb7b8dfa5f	√	9208eb59a53cc8c64c9fed9f32dde49d	√	93ff612146b4e2fc84c239b94d0984c3	×
946d418e9d1ba0ffff51702152a87a14	√	96a36c30407979fb7275feeb1f02fe67	×	98e2bc0ec70537ea2ddf6e63a3d9e186	√
9afee25d44f79fe3b276d328b4ecb2e2	×	9c00c56d3a21dc8c937ffe9b9b10b545	√	9cc9b9857e30174c1551218f34595758	√
9e19a9e0c043c0d7a45d311eaa8a837a	√	9f682227dca10806709c28ae30407f54	√	9fa47bf2438371b553f5af2e2c9ec9bf	×
a89bc9e5e391cc864cdd120346fed347	×	aaee0844f807927e21176f40b5f97cfa	√	af82b3f84451564d67e98d4a0743b102	√
b0bdf3b1e2d5c2b893384880cf403008	√	be04f7890576dfb2745ff212e0be43f3	×	c65e548c07e6dfc7881b28ec181d4ede	√
c6b6633fc7c562e6cc59fdd95bc62ae9	√	c7092ea9a5a7869c87a5272bf6e8040c	√	c7c79320f63021a33d7e695507b3d9ea	√
c8bdc884a16cf57e472d2fe75737b9ed	√	cd0d862fd57be7c220a024b7204b2608	√	ce8feae7cb54c53e195f69a8bca035c4	√
cef64e907b9f914837f612b988da1000	×	cf19269bbac0d99aaec5355add805529	√	d35ecc7fcb382f66f077052782159a31	√
d94cddb43ece4b281223110487c5e557	√	db10d0c593e3254beb9eafadf7a39dd0	√	dc5329d2f31197490b3274b4130a1e3a	√
e224afe8017cb00f46139676486e1c0a	√	e5b260e33b3bc0fb2778659785906d7a	√	e95654750c43cb5585599282b55c5197	√
eb2e22e532922db34c77c4f91b6d68e0	√	eb3a0a1480f88a9f074a3963f3998747	√	ec859a40fec458028a8d7487f56ef16b	√
f24b8b3794026aaca8042c0a63296581	√	f38d08671b05e9e3ec00a9932712aa1b	√	f71b9669aa8fee68dd5d8fded32becf0	√
fb8285fa9659ba163e1a4ccbd580ae11	√	fbdd49db42b8725bbb807f5fb6dde686	√	fe01622e9e8ab80e1a01db50425530db	√

TABLE 13. Details of 20 Apps with Alibaba cloud OSS certificate information leakage vulnerability.

MD5	Attack	MD5	Attack	MD5	Attack
05d24cd3291708abbc8265a5ce4c86c6	√	0aa8d5d2e46902509d4343077c14d15e	√	0aaf494d7cd239de49f3153fbf0b5d7c	√
4a9a568a86c64e59af818e1170be6a23	√	536b1e4bbefc2c561215789a8674a9cc	√	62b4da683eb371924fa5a979f2d15e6a	√
649af7584a13d6c19fce15d337f83d3d	√	72bc4e43c176de1b4ff46fe154b5ef7d	√	9abc182e84613484566f9b9ad6458c5b	√
9bcd13ddc51d8386f311278005296227	√	a166f47beeb255a68b9d547a574afb32	√	a436be298f5c30240fd82848361dd14a	√
a79d23fea70dff90db7ff58c5862acd5	√	a87c643134205ef3cc765bd66254e5e5	√	b5433b6c227f0c4112cc02f34ee4b0cf	√
bfbc9aa735d98302cc437b64c0a5f81d	√	c10b1d00fff31b3e922a0b45caa83b21	√	c30fb26ed7501ad01496cd328d4e12e4	√
c6a771a65a0202b291348fcffa2cb26f	√	c7c43fb6f31fa62b9f1906161085ac69	√	-	-

TABLE 14. Details of 28 Apps with Webview remote code execution vulnerability.

MD5	Attack	MD5	Attack	MD5	Attack
2bba0cc4beff8153e3e6505771ac3a8c	√	3e0b00f671748ec43c9f5e18e82d7d40	×	4499c04f441b1818dc6f6d30b0f8f53f	√
aeeecd1e24bbd979c03d01eecd6e9cb8	√	b63e2e66aea4b08c0c6f354b3d2920d9	√	d984aebfd9bb23fea1b0f89d18365aeb	√
df56369ea683d490108940bc2886fd3b	×	117edf6cfeee62ee7825c82c531d555d	×	27d0f5205a24820d29dfa0525f0b46c2	×
3cc8191183b1af5bf95763fc454cc52e	×	4b744390c167d4a08fcd9d4b638cc644	√	5f43c0b2f54ffe2f9f6d3361821fe4f1	×
5fbd372e6ad31a2150b63b7cbe1afa86	×	67718ebb4c91848ee20cccc1548bfc79	√	74f678e882f04d8ac11ec3e09d8d054b	√
7776c72af0cdfa5e0dde807bba92840e	√	85ff34863f95ee89545ecc4ef53e1056	√	87c23cdda69b4cfb2c828c38dfd925bc	√
90e701f2d4258a9224c016aa99a6fca6	×	92198367cb9e532fc31a8312e00e38f8	×	9ed211b2f67a2d2b8dd37f403e213f96	√
b8e37efdeec2ef7586e740f3f0f66963	×	c01023fbd9425c2ff4ac1d744a4d5b6f	√	e5dadd2a3a9b0b908e04a91c691145f0	×
edb2dde442b1609d2fbb8ad423ea9b61	×	9b29761786908a03ec68a7fc2477d605	√	c9f8a338e30a46728f7d3173ff62d89e	√
fd9f38478dba9f9973b3b32a00170dc1	×	-	-	-	-

the causes of the vulnerabilities. Linares-Vásquez *et al.* [45] analyzed Android system vulnerabilities by using the Bulletin resource. Zhang *et al.* [21] mainly studyed the Android system-level vulnerabilities. Through analysis rules and results, they proposed a detection tool for this type of vulnerabilities.

VIII. CONCLUSIONS

We deeply analyzed the four kinds of vulnerabilities on a data set with more than 6000 Android Apps. The aforementioned vulnerabilities are *ACOCDV*, *WRCEV*, *WBCVV* and *ICV*. We found that webview and HTTPs are existed in most of finance and shopping apps. According to this, attackers can easily perform MITM. As for *WRCEV* which has been disclosed for a long time, it still exist in browser Apps, because developers lack the methods of vulnerability identification and security development awareness. Apps that use Alibaba Cloud OSS services, information leakage vulnerability of Alibaba cloud OSS credentials is caused by the weak secure awareness of developers. We developed a vulnerability detection tool, VulArcher. The experiments show that it can automatically detect the above vulnerabilities, and has good scalability. One of vulnerabilities which VulArcher detected had been included in China National Vulnerability Database (CNVD) ID(CNVD-2017-23282). It detected more than 6000 Apps and found nearly 3000 Apps with the above vulnerabilities. and we have manually verified the accuracy of results among nearly 300 Apps. The source codes of VulArcher and sample data presented in this paper can be utilized by further researchers. We incentivize this by making our data publicly available [46].

APPENDIX

Table 13 gives the the attack results of ''Alibaba Cloud OSS credential disclosure vulnerability ''. Table 14 gives the attack results of ''WebView remote code execution vulnerability''. Since there are no more than 100 Apps for the above two types of vulnerabilities, we have all analyzed. Table 11 gives a detailed description of 98 Apps randomly selected which were detected ''WebView bypass certificate validation vulnerability''. As Table 12 shown, we selected 99 Apps that were taged as ''Improper certificate validation'', and we recorded whether or not it can be attacked for each App.

REFERENCES

[1] 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, 2015.
[2] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, ''Software vulnerability prediction using text analysis techniques,'' in *Proc. 4th Int. workshop Secur. Meas. metrics*, 2012, pp. 7–10.
[3] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, ''Predicting vulnerable software components via text mining,'' *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.
[4] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, ''VuRLE: Automatic vulnerability detection and repair by learning from examples,'' in *ESORICS*. Springer, 2017, pp. 229–246.
[5] Z. Fang, Q. Liu, Y. Zhang, K. Wang, and Z. Wang, ''IVDroid: Static detection for input validation vulnerability in Android inter-component communication,'' in *Information Security Practice and Experience*. Springer, 2015, pp. 378–392.
[6] F. Wu, J. Wang, J. Liu, and W. Wang, ''Vulnerability detection with deep learning,'' in *Proc. 3rd IEEE Int. Conf. Comput. Commun. (ICCC)*, Feb. 2017, pp. 1298–1302.
[7] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, ''Cross-project transfer representation learning for vulnerable function discovery,'' *IEEE Trans. Ind. Informat.*, vol. 14, no. 7, pp. 3289–3297, Jul. 2018.
[8] W. Diao, Y. Zhang, L. Zhang, Z. Li, F. Xu, X. Pan, X. Liu, J. Weng, K. Zhang, and X. Wang, ''Kindness is a risky business: On the usage of the accessibility apis in android,'' in *Proc. 22nd Int. Symp. Res. Attacks, Intrusions Defenses*, 2019, pp. 261–275.
[9] A. Amin, A. Eldessouki, M. T. Magdy, N. Abdeen, H. Hindy, and I. Hegazy, ''Androshield: Automated Android applications vulnerability detection, a hybrid static and dynamic analysis approach,'' *Information*, vol. 10, no. 10, p. 326, 2019.
[10] T. Yang, H. Cui, and S. Niu, ''Dynamic loading vulnerability detection for Android applications through ensemble learning,'' *Chin. J. Electron.*, vol. 26, no. 5, pp. 960–965, 2017.
[11] S. Wei, Q. Huang, and J. Huang, ''Understanding javascript vulnerabilities in large real-world Android applications,'' *IEEE Trans. Dependable Secure Comput.* early access, Jun. 11, 2018, doi: 10.1109/TDSC.2018.2845851.
[12] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, ''A large-scale study of mobile Web app security,'' in *Proc. Mobile Secur. Technol. Workshop (MoST)*, 2015, pp. 1–8.
[13] E. Chin and D. Wagner, ''Bifocals: Analyzing Web view vulnerabilities in Android applications,'' in *Information Security Applications*, Y. Kim, H. Lee, and A. Perrig, Eds. Cham, Switzerland: Springer, 2014, pp. 138–159.
[14] H. Li, L. Qian, S. Zhang, H. Zhang, and J. Liu, ''Data leakage between C/A communication: A case study on Android music app,'' in *Proc. Int. Conf. Wireless Commun. Signal Process.*, 2017, pp. 1–7.
[15] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, ''Vulpecker: An automated vulnerability detection system based on code similarity analysis,'' in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, 2016, pp. 201–213.
[16] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, ''Vuldeepecker: A deep learning-based system for vulnerability detection,'' in *Proc. NDSS*, 2018, pp. 1–8.
[17] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, ''Understanding the reproducibility of crowd-reported security vulnerabilities,'' in *Proc. 27th USENIX Secur. Symp.*, Baltimore, MD, USA, 2018, pp. 919–936.
[18] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu, ''Tainting-assisted and context-migrated symbolic execution of Android framework for vulnerability discovery and exploit generation,'' *IEEE Trans. Mobile Comput.*, early access, Aug. 20, 2019, doi: 10.1109/TMC.2019.2936561.
[19] D. Wu, D. Gao, K. T. Eric Cheng, Y. Cao, J. Jiang, and H. R. Deng, ''Towards understanding Android system vulnerabilities: Techniques and insights,'' in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, New York, NY, USA, 2019, pp. 295–306.
[20] G. Yang, J. Huang, G. Gu, and A. Mendoza, ''Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications,'' in *Proc. IEEE Symp. Secur. Privacy (SP)*, Oct. 2018, pp. 742–755.
[21] J. Zhang, Y. Yao, X. Li, J. Xie, and G. Wu, ''An Android vulnerability detection system,'' in *Network and System Security*, Z. Yan, R. Molva, W. Mazurczyk, R. Kantola, Eds. Cham, Switzerland: Springer, 2017, pp. 169–183.
[22] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, ''Modeling and discovering vulnerabilities with code property graphs,'' in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 590–604.
[23] 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.
[24] S. M. Kerner. *Mobile Internet Traffic Growing Fast*. Accessed: May 10, 2019. [Online]. Available: http://www.enterprisenetworkingplanet.com/netsp/mobile-internet-traffic-growing-fast.html
[25] B. Popper. *Google Announces*. Accessed: Jun. 4, 2018. [Online]. Available: https://www.theverge.com/2017/5/17/15654454/ android-reaches2-billion-monthly-active-users
[26] *PNF*. Accessed: Jul. 8, 2018. [Online]. Available: https://www.pnfsoftware.com/
[27] C. Sun, H. Zhang, S. Qin, N. He, J. Qin, and H. Pan, ''Dexx: A double layer unpacking framework for android,'' *IEEE Access*, vol. 6, pp. 61267–61276, 2018.
[28] *Androgurd*. Accessed: May 6, 2018. [Online]. Available: https://github.com/androguard/androguard
[29] *Alibaba*. Accessed: Jun. 8, 2018. [Online]. Available: http://jaq.alibaba.com/
[30] *Baidu*. Accessed: Jun. 8, 2018. [Online]. Available: http://apkprotect.baidu.com/

[31] *Bangcle*. Accessed: Jun. 8, 2018. [Online]. Available: http://www.bangcle.com
[32] *Tencent*. Accessed: Jun. 8, 2018. [Online]. Available: http://www.qcloud.com/product/product.php?item=appup
[33] *Qihoo360*. Accessed: Jun. 8, 2018. [Online]. Available: http://dev.360.cn/protect/welcome
[34] *Ijiami*. Accessed: Jun. 8, 2018. [Online]. Available: http://www.ijiami.cn
[35] *Burpsuite*. Accessed: Jul. 18, 2018. [Online]. Available: https://portswigger.net/burp/
[36] *Baidu*. Accessed: Apr. 10, 2018. [Online]. Available: http://pcappstore.baidu.com/en/index.php
[37] *Wandoujia*. Accessed: Apr. 10, 2018. [Online]. Available: https://www.wandoujia.com/
[38] *Qihoo360*. Accessed: Apr. 10, 2018. [Online]. Available: http://zhushou.360.cn/
[39] *Huawei*. Accessed: Apr. 10, 2018. [Online]. Available: http://app.hicloud.com/
[40] *Alibaba*. Accessed: Feb. 2, 2020. [Online]. Available: https://help.aliyun.com/knowledge_detail/66168.html
[41] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, Nov. 2014, pp. 892–903.
[42] V.-P. Ranganath and J. Mitra, "Are free Android app security analysis tools effective in detecting known vulnerabilities?" *Empirical Softw. Eng.*, vol. 25, no. 1, pp. 178–219, 2020.
[43] S. Farhang, M. Bahadir Kirdan, A. Laszka, and J. Grossklags, "An empirical study of Android security bulletins in different vendors," 2020, *arXiv:2002.09629*. [Online]. Available: http://arxiv.org/abs/2002.09629
[44] H. Feng and G. Kang Shin, "Understanding and defending the binder attack surface in android," in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, New York, NY, USA, 2016, pp. 398–409.
[45] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez, "An empirical study on android-related vulnerabilities," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, Oct. 2017, pp. 2–13.
[46] H. Zhang and J. Qin. (2019). *Experimental Material*. [Online]. Available: https://buptnsrclab.github.io/blog/2020/01/03/vularcher-site-launched

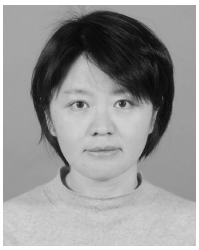
JIAWEI QIN received the B.S. degree in computer science from Shenyang Aerospace University, China, in 2015. He is currently pursuing the Ph.D. degree in computer science with the Beijing University of Posts and Telecommunications, China. His research interests include malware analysis, and security of mobile systems and Apps.

HUA ZHANG (Member, IEEE) received the B.S. and M.S. degrees from Xidian University, in 2002 and 2005, respectively, and the Ph.D. degree in cryptology from the Beijing University of Posts and Telecommunications (BUPT), in 2008. She is currently an Associate Professor with the Institute of Network Technology, BUPT. Her research interests include cryptographic protocols, cloud computing security, and industrial control systems security.

JING GUO received the B.S. degree from the Beijing University of Posts and Telecommunications (BUPT), in 2004, and the Ph.D. degree in information and communication system from Tsinghua University, in 2011. She is currently a Senior Engineer with National Internet Emergency Center. Her research interests include network security situation awareness, protection of critical information infrastructure, and the IoT security.

SENMIAO WANG received the B.Eng. degree in software engineering from the Dalian University of Technology, in 2017. She is currently pursuing the Ph.D. degree with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, China. Her current research interests include mobile security and machine learning.

QIAOYAN WEN received the B.S. and M.S. degrees in mathematics from Shaanxi Normal University, Xi'an, China, in 1981 and 1984, respectively, and the Ph.D. degree in cryptography from Xidian University, Xi'an, in 1997. She is currently a Professor with the Beijing University of Posts and Telecommunications. Her current research interests include coding theory, cryptography, information security, internet security, and applied mathematics.



YIJIE SHI received the B.S. degree in radio technology from the Dalian University of Technology, Dalian, Liaoning, China, in 1997, and the M.S. degree in cryptography and the Ph.D. degree in computer science from the Beijing University of Posts and Telecommunications, Beijing, China, in 2000 and 2014, respectively. She is currently a Lecturer with the Beijing University of Posts and Telecommunications. Her major research interests include security of industrial control systems, security of cloud computing, and cryptography.