Comput Syst Sci & Eng (2018) 2: 149–155
© 2018 CRL Publishing Ltd

International Journal of
Computer Systems
Science & Engineering

A Dynamic Online Protection Framework for Android Applications*

Junfeng Xu†and Linna Zhou

University of International Relationship, 12 PoshangCun, Haidian Area, Beijing, 100091, China

At present, Android is the most popular Operating System (OS) which is widespreadly installed on mobile phones, smart TVs and other wearable devices. Due to its overwhelming market share, Android attracts the attentions from many attackers. Reverse Engineering technology plays an important role in the field of Android security, such as cracking applications, malware analysis, software protection, etc. In order to prevent others from obtaining the real codes and tampering them, this paper designs and implements a online dynamic protection framework by deploying dynamic anti-debugging technology for Android application with comprehensive utilization of encryption, dynamic loading and shell technologies. Evaluated the performance on different aspects, the proposed framework can work effectively for Android application protection. Comparing with the static protection scheme, the proposed online dynamic protection framework can prevent the android applications from cracking and malicious analysis to the utmost

Keywords: Software Protection; Android Security; Encryption

1. INTRODUCTION

Since Google released the open source Linux-based smart phone operating system (OS) Android on November 5, 2007, Android was highly favored by businesses and developers for its open source, openness and customizability. According to the reports by US-based IT research firm IDC, Android's market share will rise from 81.2% in 2015 to 82.6% in 2019[1] in 2019. However, Android's popularity also attracts a large number of attackers. According to the China mobile security status report which is released by security company 360[5], 360 Center for Internet Security cumulatively intercepted 18.74 million new malicious program samples of Android OS in 2015. Due to its low-cost of development, a large part of the malicious software are repackaged applications. Repackaging is commonly used by Android malware developers, by analyzing 1260 malware some researchers found that there were 86% repackaged applications in the given samples[9]. In order to implement re-packaging, the attackers have to firstly reverse analysis the original APK (Android Package) so that their malicious code can be injected. On the other hand, reverse analysis plays an important role in analysis of the Android malware.

Android reverse analysis includes static analysis and dynamic debugging in accordance with whether the Android program is performed. Static analysis is a procedure that analysts decompile APK with decompilation tools, then read the code obtained from decompilation to understand the principle of the application. With the rapid development of the Android application protection technology, static analysis technology was often powerless to cope with code obfuscation, dynamic loading, software reinforcement and other protection measures [10]. On analysis of 50,000 apps from Google Play and third-party application market, some researchers found that as of March 2013, there were 32.8% (16,396 APKs) applications with dynamic loading behavior[11, 7]. For obtaining runtime information of such applications, static analysis technology is even more helpless. Static analysis is severely limited under a number of circumstances and the deficits of static analysis gave rise to dynamic analysis techniques[12]. Many researchers have also given their own dy-

*This work is supported by National Natural Science Foundations of China (Grand No. 61672534, U1536207, U1736117 and U1636115).

†Corresponding Author. E-mail: fibger@foxmail.com

namic analysis methods and tools of Android applications, such as, TaintDroid [13], DroidScope[14], Aurasium[15] etc.

Dynamic debugging is a basic but very important technology in dynamic analysis of Android applications. Dynamic debugging is also called assembly-level debugging corresponding to the source-level debugging in phase of software development[2, 6]. It is a procedure that in the case of only executable Android application can be got, analysts run the program, track and analyze the assembly code, obtain intermediate results of program's execution through observing values of the register, then grasp behavior of the program, and at last understand the core algorithm of the program. In Android reverse analysis, especially for analyzing native code, dynamic debugging has a huge advantage[3]. Dynamic debugging makes software's implementation details exposed, to protect Android application from dynamic debugging, anti-debugging technology emerged. Anti-debugging can greatly enhance the difficulty of dynamic debugging software, so as to achieve effective protection for android applications[4, 8].

This paper present a new protection framework which can exchange online keys to encrypt the original APK file. After comparing with the existing anti-debugging technologies, this paper chooses a practical dynamic anti-debugging scheme according to the character of the proposed online protection framework for android applications. Deploying the new framework and the method, the protection system of Android applications can eliminate the secondary packaging feasibility and extend the life cycle of the software as long as possible.

The remainder of this paper is organized as follows. In Section 2, the current related protection technologies for anti-debugging of Android application are described. Section 3 proposes a new online dynamic framework to prevent Android applications being decompiled. Some evaluations and assessments are described to illustrate the effectiveness of the proposed framework. Finally, the concluding remarks are given in Section 5.

2. COMMON ANTI-DEBUGGING TECHNOLOGIES

To protect Android application from dynamic debugging, developers can take a variety of anti-debugging techniques and the following is an introduction to several commonly used anti-debugging techniques.

2.1 Code obfuscation

Code obfuscation is a technology used to hide the intent of a program and it can increase the dif-ficulty of reverse analysis. The traditional code obfuscation technology increases the difficulty of static analysis by replacing some function names with meaningless words or complicating logic of program under the premise not changing the function of program. The literature[16] proposed an integrated protection system SMOG based on obfuscation interpretation, the system confuses the Android executable code, and it changes limitations that the traditional code obfuscation techniques can only anti static analysis. In addition to the SMOG system, Obfuscator-LLVM (OLLVM) project is also of concern[17]. The project focuses on the LLVM compiler, it can be used for obfuscating native code (C/C++, etc.), thus greatly improves the difficulty to reverse native program that has already been quite difficult. The paper [18] proposes a practical tool that makes Android application have the ability of effective self-protection, OLLVM was used in this project. Moreover, there are other Android application obfuscation tools such as [19]: Proguard, DashO, Dexguard, DexProtector, ApkProtect, Shield4j, Stringer, Allitori, etc.

2.2 Emulator detection

ro.debuggable's default value in default.prop of Android emulator is ′1′, that means allowing Android to debug any program. Therefore, using Android emulator can bring a lot of conveniences for reversing Android application. The emulator detection function can eliminate these facilities: when the program is running in emulator detected, it can be terminated.

The emulator detection utilizes the differences between the emulator and real Android devices on certain properties. We can usually use the following methods to determine whether an Android program is running in the emulator: detect "*/dev/socket/qemud*", "*/dev/qemu_pipe*" these two channels (only exist in emulator); check whether goldfish driver is included in "*/proc/tty/drivers*" (only exist in emulator); detect unique files of emulator, such as "*/system/lib/libc_malloc_debug_qemu.so*", "*/sys/qemu_trace*" and so on; detect default phone numbers of emulator, such as "15555215554" "15555215556", etc.; whether IDS of the Android device is "000000000000000"; whether IMSI ID of the device is "310260000000000"; detect network operators, the network operator name in emulator is "android". However, it is the owned operator name in real device and it is empty if there is no SIM card in the device.

2.3 Shell

Shell technology is derived from the software protection technology of Windows and now it is also widely used in Android applications protection. To reverse analysis interested applications, attackers have to crack the shell of the reinforced application firstly. In shell technology of Android, the object to be reinforced can be the whole original APK or only the dex file in APK. The object is encrypted and stored in a specific location in the shell APK. When the shell APK begins to run, it decrypts the encrypted portion and gets the original APK or dex file, then dynamic loading the original application.

As mentioned above, currently, the dynamic loading technology has been commonly used in Android software development. To study the malicious behaviors of malware with dynamic payloads, ZHENG Min, et al. proposed a ptrace based Android dynamic analysis system with forward execution: DroidTrace. The system uses ptrace to monitor the system calls of the target process which is running the dynamic payloads and classifies the payloads behaviors (such as file access, network connection, inter-process communication and privilege escalation) through the system call sequence.

2.4 Anti-debugging runtime

The current runtime anti-debugging techniques based on the following principles:

2.4.1 Ptrace

Ptrace is an important system call in Linux and it provides a mechanism by which a parent process may observe and control the execution of another process. Many debugging tools such as IDA, GDB implement dynamic debugging Android program by means of ptrace. An important feature of ptrace is that a process can be traced only by one process. So, we can call *ptrace (PTRACE_TRACEME, 0, 0, 0)* in our own program to achieve the effect of simple anti-debugging.

2.4.2 Debugging check

Read the value of TracerPid in ’$/proc/[pid]/status$’, if it is $'0'$, that means the program is not being debugged, otherwise, the value represents PID of the debugger. Another way is to read the value of ’$/proc/[pid]/wchan$’ [20], when it is ’$ptrace\_stop$’, the program is being debugged.

2.4.3 Time difference

When tracking and debugging Android applications by dynamic debugging tools such as IDA, GDB, because of the need to step through the key code, execution time of the program will be far greater than the time under normal circumstances. So time difference detection code can be inserted into the key code segment, if we find that the time difference exceeds a certain threshold, that means the program is being debugged.

2.4.4 Parent process detection

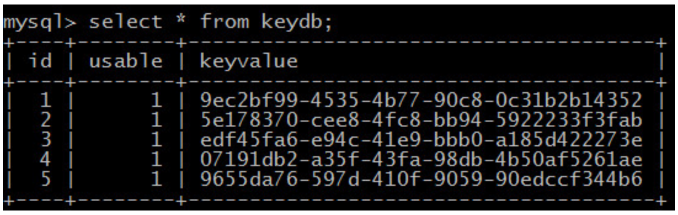
Examine PPID of the target process, determine whether it is PID of the android_server, gdb-server, etc.

3. ANDROID APPLICATION PROTECTION FRAMEWORK BASED ON ANTI-DEBUGGING AND NETWORK KEY

In this paper, we design and implement an Android application reinforcement scheme based on anti-debugging and network key with comprehensive utilization of encryption, anti-debugging and dynamic loading techniques.

The scheme involves three objects [21]: the original APK to be reinforced, namely $OriginalAPK$; shell APK, namely $UnShellAPK$; reinforcing tool, namely ShellTools. $OriginalAPK$ is reinforced on the remote server of network, each reinforced APK has its own unique identifier ID, and we use the KEY corresponding to the ID to encrypt $OriginalAPK$. The KEY table maintained in server-side is shown in Fig. 1.

The id field in the table will be written to the dex file of $UnShellAPK$ by Shell Tools. The usable field represents whether the reinforcement id is available, "1" represents that it is available and "0" represents not. The key value field is a random string generated by the class UUID, which is used as the encryption key. When $UnShellAPK$ obtains the reinforcement key corresponding to its own ID from the Server, if it connects the database at the Server directly, the username and password of the database should be written in $UnShellAPK$, which is clearly undesirable. To solve this problem, we use WampServer as the server in our scheme. WampServer is a Windows web development environment, it allows us to create web applications with Apache2, PHP and a MySQL database. $UnShellAPK$ uses PHP as a medium to access the database which can avoid the exposure problem of username and password of database at Server.



```
mysql> select * from keydb;
+----+--------+--------------------------------------+
| id | usable | keyvalue                             |
+----+--------+--------------------------------------+
|  1 |      1 | 9ec2bf99-4535-4b77-90c8-0c31b2b14352 |
|  2 |      1 | 5e178370-cee8-4fc8-bb94-5922233f3fab |
|  3 |      1 | edf45fa6-e94c-41e9-bbb0-a185d422273e |
|  4 |      1 | 07191db2-a35f-43fa-98db-4b50af5261ae |
|  5 |      1 | 9655da76-597d-410f-9059-90edccf344b6 |
+----+--------+--------------------------------------+
```

Figure 1 Key table maintained in the server

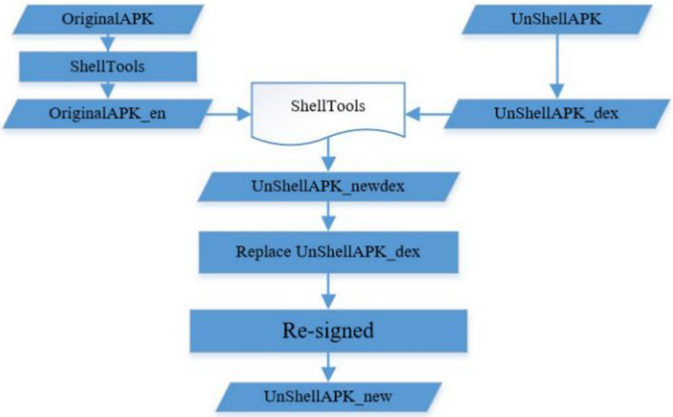


Figure 2 Generating flow of the reinforced APK

The generating flow of the reinforced APK is shown in Fig. 2, specific implementation process is: Compile $UnShellAPK$ and generate the file classes.dex (at the moment $UnShellAPK$ is not executable for lacking of payload), namely $UnShellAPK\_dex$. Then encrypt the $OriginalAPK$ by Shell Tools with XOR and RC4 encryption algorithm and get $OriginalAPK\_en$, the encryption key of RC4 is acquired from the database at server. The file $OriginalAPK\_en$ itself and its file size, reinforcement ID will be sequentially written to the end of the file $UnShellAPK\_dex$. Next, according to the file format requirements of dex file, fix in turn the dex file size, SHA-1 hashes, adler32 check value (avoiding the Error [INSTALL_FAILED_DEXOPT]). After completing the repair we get $UnShellAPK\_newdex$, then replace the original $UnShellAPK\_dex$ with it, delete the original signature file and resigned, finally we can get the reinforced $APKUnShellAPK\_new$.

The execution flow of the UnShellAPK_new in client is shown in Fig. 3.

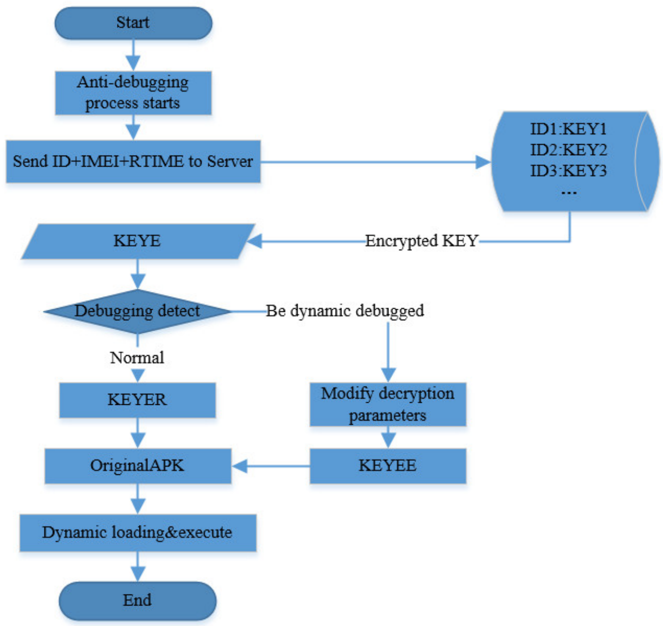


Figure 3 Execution flow of the UnShellAPK_new

UnShellAPK_new firstly starts an anti-debugging process and the process starts a child process which reads the value of TracerPid in the file "$/proc/\$PID/status$" in a certain interval of time. If finding that the value is not 0, it will kill the current process. The anti-debugging algorithm is as follows.

Algorithm 1: *anti_debug* Function

```
$getpid()$ ;
if $fork() == 0$ then
    $ptrace(PTRACE\_TRACEME, 0, 0, 0)$ ;
    while $True$ do
        Get pointer fp of the file 'proc$PIDstatus' ;
        while Read the contents of the file from fp line by line do
            if Read the contents of 'TracerPid' then
                Get the value of TracerPid ;
                if TracerPid != 0 then
                    Debugger detected, kill the process ;
                $break$ ;
        $sleep(CHECK\_TIME)$ ;
```

This method should be modified with $'\_\_attribute\_\_((constructor))'$ so that it can be compiled into the section $init\_array$ by the compiler, which will ensure that the method is executed at the very beginning.

After $UnShellAPK\_new$ starts the anti-debugging process, it sends its own reinforcement $ID + IMEI + RTIME$ (request time) to the Server to request the KEY required in decryption. The server firstly checks that whether the received IMEI is "000000000000000", if so, that indicates the program is running in an emulator, server returns a fixed invalid key at this time. If the IMEI is not "000000000000000", the server look-up table of its database to find the KEY corresponding to the ID, after this, encrypts the KEY with MD5 value of $ID+IMEI+RTIME$ using RC4 encryption algorithm, then, send obtained KEYE to the client. The client makes debugging detection once again, if it finds that the program is being dynamic debugged, it will change the value of RTIME to CTIME (current time) by a native method. When $UnShellAPK\_new$ use $ID + IMEI + RTIME$ (actually CTIME) to decrypt KEYE it gets KEYEE, and it will obtain an invalid $OriginalAPK$ when decrypting with KEYEE, so $UnShellAPK\_new$ will fail to dynamic loading the $OriginalAPK$. If the running environment of the program is normal, it will use $ID + IMEI + RTIME$ to decrypt KEYE to get KEYER, then decrypt $OriginalAPK\_en$ with KEYER, we can get the right $OriginalAPK$, so $UnShellAPK\_new$ now can dynamic loading it.

This scheme has two designs of anti-debugging. The first is at the beginning of the program where the anti-debugging process starts. The second is when KEYE got, if the check result is not normal, the application no longer actively kills itself, but makes the $OriginalAPK$ obtained by the attacker not invalid by modifying the decryption parameters, and ultimately fail to dynamic loading $OriginalAPK$. This is mainly based on the following considerations: when the attackers reverse analysis the reinforced program, they often regard exiting of the program as an important feature, narrow range step by step, and finally find the anti-debugging code, then try to get rid of the anti-debugging code. If we do not immediately exit the program when being dynamic debugged detected, it can increase the difficulty that attackers find the anti-debugging code.

Under normal circumstances, UnShellAPK_new loads its own components after startup of the context class Application. In order to read $OriginalAPK\_en$ and decrypt it before $UnShellAPK\_new$ loads its own components, we need to define the class $ProxyApplication$ that inherits from the class Application, complete the following tasks in the class:

1) Separate $OriginalAPK\_en$ from the dex file of $UnShellAPK\_new$, and decrypt it to get $OriginalAPK$, and get so library files in $OriginalAPK$.

2) To be able to dynamic loading $OriginalAPK$, in the method $attachBaseContext()$ (this method is executed prior to the method $onCreate()$)of the class $ProxyApplication$, we need to generate a class loader($DexClassLoader$) that can dynamic loading $OriginalAPK$, and its parent class loader should be the class loader of $UnShellAPK$(the system default class loader: $PathClassLoader$), then replace the original $PathClassLoader$ with $DexClassLoader$. The main difference between $PathClassLoader$ and $DexClassLoader$ is that $PathClassLoader$ can only load the classes of APK which have been installed in local device, while $DexClassLoader$ have no the limit. Specific code are shown as follows:

```
Object currentActivityThread =
RefInvoke.invokeStaticMethod (
"android.app.ActivityThread", "currentActivityThread",
new Class[] { }, new Object[] { });
String packageName = this.getPackageName ();
ArrayMap mPackages = (ArrayMap)
RefInvoke.getFieldOjbect ( "android.app.ActivityThread",
currentActivityThread, "mPackages");
WeakReference wr = (WeakReference) mPackages.get
(packageName);
DexClassLoader dLoader = new DexClassLoader
(apkFileName, odexPath, libPath, (ClassLoader)
RefInvoke.getFieldOjbect ( "android.app.LoadedApk",
wr.get (), "mClassLoader"));
RefInvoke.setFieldOjbect ("android.app.LoadedApk",
"mClassLoader", wr.get(), dLoader);
```

RefInvoke is a tool class that implements reflection calls.

3) Replace the Application object of *UnShellAPK_new* with the Application object of *OriginalAPK*. For this purpose, we need to write the application class name of *OriginalAPK* to the *meta − data* tag of application tag in the file *AndroidManifest.xml*, namely:

$< meta - dataandroid : name = $"*APP_CLASS_NAME*" *android* : *value* = "*com.demo.originalapk.OriginalApplication*"$/ >$

then in the method *onCreate*() of *UnShellAPK_new*, get *ApplicationInfo* object by the method *this.getPackageManager*().*getApplicationInfo*() and further obtain the class name. When we get the class name, we can execute *OriginalAPK* by calling the method *onCreate*() in the *OriginalApplication* object, the key code is as follows:

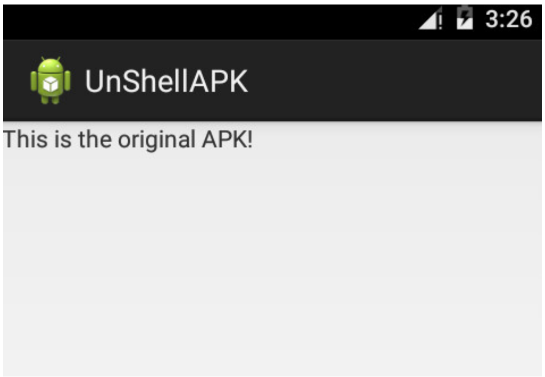
```
Application app = (Application) RefInvoke.invokeMethod(
"android.app.LoadedApk", "makeApplication",
loadedApkInfo, new Class[] { boolean.class,
Instrumentation.class }, new Object[] { false, null });
RefInvoke.setFieldOjbect("android.app.ActivityThread",
"mInitialApplication", currentActivityThread, app);
app.onCreate();
```

At this point OriginalAPK begins its normal execution.

4. EVALUATIONS AND ASSESSMENTS OF THE FRAMEWORK

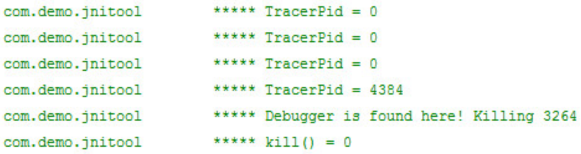
4.1 Effectiveness of the program's normal execution

In order to verify the effectiveness of the proposed scheme, the experiment is carried out, normal execution of the reinforced APK is shown in Fig. 4, the displaying interface is the *MainActivity*'s interface of the *OriginalAPK*.



This is the original APK!

Figure 4 Normal execution of the reinforced APK



```
com.demo.jnitool        ***** TracerPid = 0
com.demo.jnitool        ***** TracerPid = 0
com.demo.jnitool        ***** TracerPid = 0
com.demo.jnitool        ***** TracerPid = 4384
com.demo.jnitool        ***** Debugger is found here! Killing 3264
com.demo.jnitool        ***** kill() = 0
```

Figure 5 The value of TracerPid changes before and after *gdbserver*'s attaching

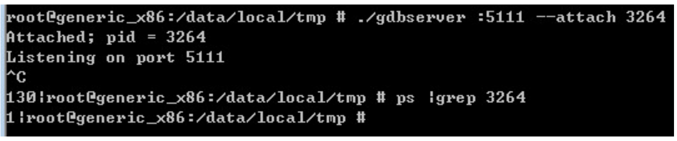
4.2 Anti-debugging validation

We can verify effectiveness of anti-debugging of the proposed scheme by using *gdbserver* to attach *UnShellAPK*, as shown in Fig. 5 and Fig. 6. Before attached by *gdbserver*, the value of *TracerPid* is "0", application performs normally. While after the attaching, the value of *TracerPid* becomes 4384, that is the PID of *gdbserver*. Then *UnShellAPK* is killed, "$kill() = 0$" means that the method is executed successfully. Then search for the process with PID 3264 in the process list we can see that the process no longer exists.

4.3 Comparison of reverse test results

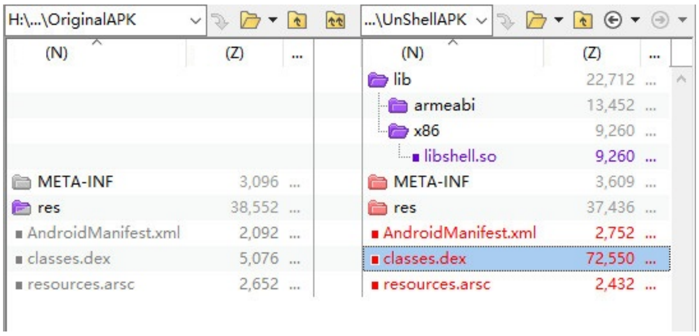
4.3.1 Comparison of APK file structure before and after reinforcement

The comparison of APK file structure before and after reinforcing is shown in Fig. 7, as what can be seen from the figure, the reinforced APK file directory includes a lib directory which does not exist in *OriginalAPK*, anti-debugging and decryption method in this scheme exist in the file libshell.so in form of native code. It also can be seen that the size of the file classes.dex changed greatly, this is due to the encrypted *OriginalAPK* stored in the file.



```
root@generic_x86:/data/local/tmp # ./gdbserver :5111 --attach 3264
Attached; pid = 3264
Listening on port 5111
^C
130|root@generic_x86:/data/local/tmp # ps |grep 3264
1|root@generic_x86:/data/local/tmp #
```

Figure 6 Attach to the program with *gdbserver*



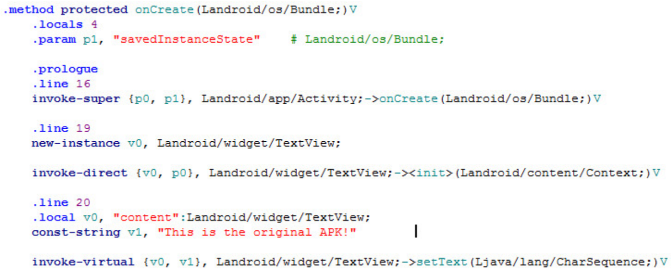
H:\...\OriginalAPK

(N)	(Z)	...
META-INF	3,096	...
res	38,552	...
AndroidManifest.xml	2,092	...
classes.dex	5,076	...
resources.arsc	2,652	...

...\UnShellAPK

(N)	(Z)	...
lib	22,712	...
armeabi	13,452	...
x86	9,260	...
libshell.so	9,260	...
META-INF	3,609	...
res	37,436	...
AndroidManifest.xml	2,752	...
classes.dex	72,550	...
resources.arsc	2,432	...

Figure 7 Comparison of APK file structure before and after reinforcing



```
.method protected onCreate(Landroid/os/Bundle;)V
    .locals 4
    .param p1, "savedInstanceState"    # Landroid/os/Bundle;

    .prologue
    .line 16
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

    .line 19
    new-instance v0, Landroid/widget/TextView;

    invoke-direct {v0, p0}, Landroid/widget/TextView;-><init>(Landroid/content/Context;)V

    .line 20
    .local v0, "content":Landroid/widget/TextView;
    const-string v1, "This is the original APK!"

    invoke-virtual {v0, v1}, Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V
```

Figure 8 Snippet of MainActivity in OriginalAPK

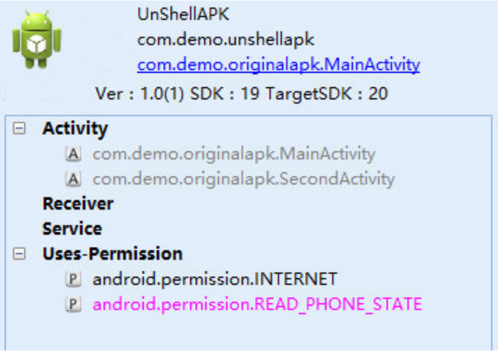
4.3.2 Reverse OriginalAPK

Reverse $OriginalAPK$ by reverse analysis tools we can get the smali code shown in Fig. 8, from which we can get the implementation details of $OriginalAPK$.

4.3.3 Reverse UnShellAPK

Project information obtained about UnShellAPK is shown in Fig. 9.

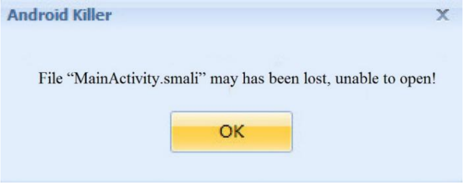
But when we click on the entry class $com.demo.originalapk.MainActivity$, the reverse tool reports an error of smali file missing, as is shown in Fig. 10. That is because $OriginalAPK$ exists in $Un-ShellAPK$ in an encrypted form, the reverse tool are not able to resolve the class information of $OriginalAPK$ correctly.



UnShellAPK
com.demo.unshellapk
com.demo.originalapk.MainActivity
Ver : 1.0(1) SDK : 19 TargetSDK : 20

- Activity
  - com.demo.originalapk.MainActivity
  - com.demo.originalapk.SecondActivity
- Receiver
- Service
- Uses-Permission
  - android.permission.INTERNET
  - android.permission.READ_PHONE_STATE

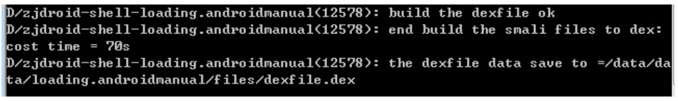
Figure 9 Project information of UnShellAPK



Android Killer

File “MainActivity.smali” may has been lost, unable to open!

Figure 10 Reverse tool report file missing error



```
D/zjdroid-shell-loading.androidmanual(12578): build the dexfile ok
D/zjdroid-shell-loading.androidmanual(12578): end build the smali files to dex:
cost time = 70s
D/zjdroid-shell-loading.androidmanual(12578): the dexfile data save to =/data/da
ta/loading.androidmanual/files/dexfile.dex
```

Figure 11 Dump dex file

4.4 Crack shell test using ZjDroid

4.4.1 Introduction of ZjDroid

ZjDroid is a dynamic reverse analysis module based on the Xposed framework and reverse analysts can accomplish the following tasks [22] by ZjDroid:

- Dump dex file in memory;
- Memory BackSmali based on Dalvik key pointer;
- Crack reinforced applications effectively;
- Dynamic monitoring sensitive API;
- Dump data in the specified memory area;
- Get dex information loaded by application;
- Get loaded classes information of the specified dex file;
- Dump java heap information in Dalvik;
- Dynamic run lua script in the target process.

4.4.2 Demo of crack shell using ZjDroid

Next we use ZjDroid to crack shell of an application reinforced by $ijiami$. Prerequisite of using ZjDroid is that the Android device has been "rooted", and the Xposed Framework has been installed, in which the ZjDroid module option is checked. First, we use the command:

$adblogcat-szjdroid-shell-loading.androidmanual$

to open the log output platform of ZjDroid, we use the command:

$ambroadcast-acom.zjdroid.invoke--eitarget12578--escmd'action:dump\_dexinfo'$

to get Info about currently loaded dex files of the APK, we can find that there is only one item:

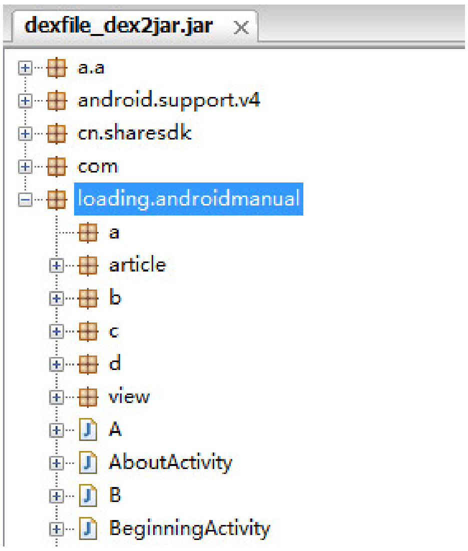
$filepath:/data/app/loading.androidmanual-1.apk$

so, we use the command:

am broadcast -a com.zjdroid.invoke –ei target 12578 –es cmd ’action:backsmali, "dex-path":"/data/app/loading.androidmanual-1.apk"’

to start dex file dumping, we can get results shown in Fig. 11

Log Info shows "build the dexfile ok", that means we dump the dex file successfully. Then we export the file dexfile.dex



dexfile_dex2jar.jar ×

- a.a
- android.support.v4
- cn.sharesdk
- com
- loading.androidmanual
  - a
  - article
  - b
  - c
  - d
  - view
  - A
  - AboutActivity
  - B
  - BeginningActivity

Figure 12 Open the jar file with JD-GUI

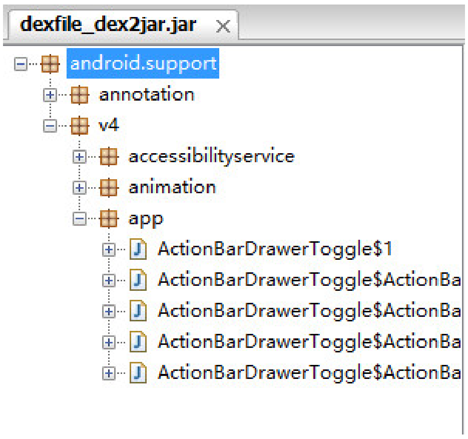
from the directory specified by Log, and convert the dex file to jar file by the tool dex2jar, open the jar file with JD-GUI, we find that the file can be opened successfully, as is shown in Fig. 12. It proves that we have restored the original dex file and cracked shell of the APK successfully.

4.4.3 Crack shell of the APK reinforced by proposed scheme

The same as the last section, first get Info about currently loaded dex files of APK, as is shown in Fig. 13. The second item is the shell part, we need only to dump the first part payload.apk using the backsmali command, then convert the obtained dexfile.dex to jar file. The opened jar file is shown in Fig. 14, we can see that ZjDroid did not extract any valid class files of $OriginalAPK$. That's because $OriginalAPK$ exists in $UnShellAPK\_new$ in the encrypted form and after decryption it is performed in the way of dynamic loading. So, ZjDroid can't get any valuable file of $OriginalAPK$, the scheme proposed is proved effective.

5. CONCLUSIONS

In this paper, we studied anti-debugging technology in Android reverse, proposed and implemented a scheme of reinforcing Android application based on anti-debugging and network key. In the proposed scheme, the strategies can be improved when finding that the application is being dynamic debugged to achieve a better effect of anti-debugging, it will be the direction of our future work. In addition, as mentioned by responsible person of a well-known Android application reinforcement company: "The most difficult part is that growing and enhancing security policy conflicts with the user experience." How to protect Android application better against dynamic debugging with a smaller affect to the user experience will be the focus of future research.



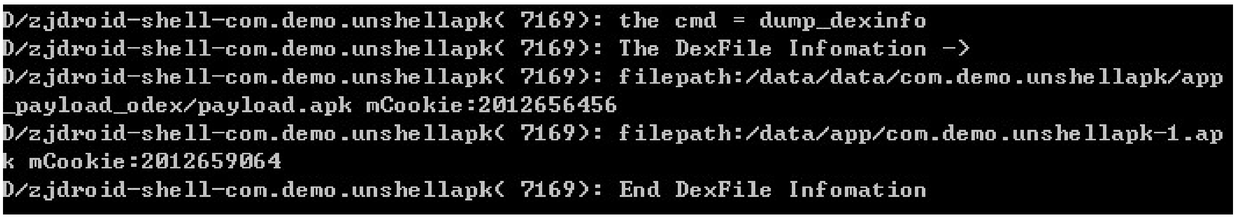
dexfile_dex2jar.jar ×

- android.support
  - annotation
  - v4
    - accessibilityservice
    - animation
    - app
      - ActionBarDrawerToggle$1
      - ActionBarDrawerToggle$ActionBa
      - ActionBarDrawerToggle$ActionBa
      - ActionBarDrawerToggle$ActionBa
      - ActionBarDrawerToggle$ActionBa

Figure 14 Payload dex file dumped by ZjDroid

REFERENCES

1. H. Bagheri, A. Sadeghi, J. Garcia and S. Malek, COVERT: Compositional Analysis of Android Inter-App Permission Leakage, in IEEE Transactions on Software Engineering, 2015, 41(9):866-886.
2. Y. Jing, G. J. Ahn, Z. Zhao and H. Hu, Towards Automated Risk Assessment and Mitigation of Mobile Applications, in IEEE Transactions on Dependable and Secure Computing, 2015, 5: 571-584.
3. Lei Cen; Christoher S. Gates; Luo Si; Ninghui Li, A Probabilistic Discriminative Model for Android Malware Detection with Decompiled Source Code, IEEE Transactions on Dependable and Secure Computing, 2015.
4. Suleiman Y. Yerima; Sakir Sezer; Igor Muttik, High accuracy android malware detection using ensemble learning, IET Information Security, 2015.



```
D/zjdroid-shell-com.demo.unshellapk( 7169): the cmd = dump_dexinfo
D/zjdroid-shell-com.demo.unshellapk( 7169): The DexFile Infomation ->
D/zjdroid-shell-com.demo.unshellapk( 7169): filepath:/data/data/com.demo.unshellapk/app
_payload_odex/payload.apk mCookie:2012656456
D/zjdroid-shell-com.demo.unshellapk( 7169): filepath:/data/app/com.demo.unshellapk-1.ap
k mCookie:2012659064
D/zjdroid-shell-com.demo.unshellapk( 7169): End DexFile Infomation
```

Figure 13 Dexinfo of the reinforced APK

5. http://zt.360.cn/1101061855.php?dtid=1101061451&did=11015 93997.
6. Jaya Bhattacharjee, Anirban Sengupta, Chandan Mazumdar and Mridul Sankar Barik, A two-phase quantitative methodology for enterprise information security risk analysis. International Journal of Computer Systems Science and Engineering, 2012, 29(1):809-815.
7. Shuchih Ernest Chang and Anne Yenching Liu, Information security in practices: Exploring privacy and trust in computer and inernet surveillance. International Journal of Computer Systems Science and Engineering, 2016, 31(2).
8. Yifan Chen, Xiang Zhao, Jiuyang Tang, Weiming Zhang and Haichuan Shang,Taxi-taking recommendation using real-time trajectories: an online query based approach. International Journal of Computer Systems Science and Engineering, 2016, 31(2).
9. Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, 2012, pp. 95-109.
10. Bartel, J. Klein, M. Monperrus and Y. Le Traon, "Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges and Solutions for Analyzing Android," IEEE Transactions on Software Engineering, 2014, 40(6):617-632.
11. M. Zheng, M. Sun and J. C. S. Lui, "DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability," 2014 International Wireless Communications and Mobile Computing Conference (IWCMC), Nicosia, 2014, pp. 128-133.
12. J. Schutte, R. Fedler and D. Titze, "ConDroid: Targeted Dynamic Analysis of Android Applications," 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, Gwangiu, 2015, pp. 571-578.
13. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taint-droid: an information-flow tracking system for realtime privacy monitoring on smartphones, In Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10, 2010.
14. L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis, In Proceedings of USENIX Security'12, 2012.
15. R. Xu, H. Sadi, and R. Anderson. Aurasium: practical policy enforcement for android applications, In Proceedings of USENIX Security 2012, 2012.
16. J. Shu, J. Li, Y. Zhang and D. Gu, Android App Protection via Interpretation Obfuscation, Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on, Dalian, 2014, pp. 63-68.
17. https://github.com/obfuscator-llvm.
18. M. Protsenko, S. Kreuter and T. Muller, Dynamic Self-Protection and Tamperproofing for Android Apps Using Native Code, Availability, Reliability and Security (ARES), 2015 10th International Conference on, Toulouse, 2015, pp. 129-138.
19. http://fuzion24.github.io/android/obfuscation/ndk/llvm/o-llvm/2014/07/27/android-obfuscation-o-llvm-ndk/
20. J. Xu, L. Zhang, D. Lin and Y. Mao, Recommendable Schemes of Anti-decompilation for Android Applications, 2015 Ninth International Conference on Frontier of Computer Science and Technology, Dalian, 2015, pp. 184-190.
21. http://blog.csdn.net/jiangwei0910410003/article/details/48415225
22. http://bbs.pediy.com/showthread.php?p=1303746.