A Hidden File Extraction Scheme Defeating Malware Using Android Dynamic Loading

Hongsun Yoon[1], Hyunseok Shim[2], and Souhwan Jung[3](✉)

[1] Department of Information and Telecommunication Engineering, Soongsil University, Seoul, South Korea
ghdtjs243@gmail.com

[2] Department of Information and Communication Convergence, Soongsil University, Seoul, South Korea
ant_tree@naver.com

[3] School of Electronic Engineering, Soongsil University, Seoul, South Korea
souhwanj@ssu.ac.kr

Abstract. Recently, malicious Android applications have become intelligent to bypass traditional static analysis. Among them, which using dynamic loading techniques hide malicious code by separating DEX files. These additional DEX files can be installed together during the installation time in different directory or downloaded from the command and control server. However intelligent malwares delete the DEX files after execution to avoid analysis. Therefore, It is difficult to figure out the some of hidden behavior without extracting files used for dynamic loading. In this paper, we propose a extraction algorithms to save the loaded or deleted DEX file using Xposed. After that, verifies whether the extracted DEX file is malicious by using the proposed technique. This method allows you to analyze additional actions performed by malware through analysis. As a result, it contributes to find hidden features of Application.

Keywords: Multidex · Dynamic loading · Java reflection · ClassLoader · Android malware

1 Introduction

When comparing the market share of the portable device OS in 2018 based on data provided by Statista, Android accounts for more than 85% of the market

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-00477, Development of android security framework technology using virtualized trusted execution environment) and this work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2020-0-00952, Development of 5G Edge Security Technology for Ensuring 5G+ Service Stability and Availability).

I. You et al. (Eds.): MobiSec 2019, CCIS 1121, pp. 85–98, 2020.
https://doi.org/10.1007/978-981-15-9609-4_7

```
JavaReflection obj = new com.sample.JavaReflection;
obj.println();
```

```
Class JR = Class.forName(com.sample.JavaReflection , null)
Object obj = JR.newInstance();
Method mtd = hello.getMethod( println , null);
Mtd.invoke(obj , null);
```

Fig. 1. Java reflection sample code.

[23]. In addition, IDC's estimated smartphone market share statistics predicted that the Android OS would maintain an 85% share over the next five years [13]. Therefore, apps running on Android environments have increased dramatically, and malicious apps have also been on a steady rise according to McAfee Report [17] in the first of 2019. Nowadays, intelligent malwares avoid static and dynamic analysis through source code obfuscation, encryption, dynamic loading, and environment detection. Among the methods of analyzing an app, static analysis has the advantage of analyzing the entire malicious behavior source code to enable accurate analysis. But it cannot respond to apps that separate source codes using dynamic loading techniques. So some Android malicious apps use multiple DEX with dynamic loading techniques to avoid static analysis, which invoke additional DEX from the main DEX to perform actual malicious behavior. Because of this behavior, it is difficult to find all of code that performs in the Application.

In this paper, We propose a algorithm for extracting files dynamically loaded by Android malicious apps. With Xposed framework, we can see the API calls and extract files used to dynamic loading from malicious apps. And This allows analyst to find and analyze dynamic loading codes hidden by intelligent and advanced Android malicious apps. Following the introduction, We explain about Java reflection and Xposed in Sect. 2. After that, see details of APK structure and feature of the API hiding technique with Java reflection as a related research in Sect. 3. In Sect. 4, introduces the algorithms to extract DEX file and describes the operation process. After that, we verifies the proposal algorithms and analyze loaded DEX file. Finally, sum up the result of this paper and describes the future research and direction.

2 Background

2.1 Java Reflection

Java reflection is a feature of Java language that is used to check or change run-time behavior of applications running in Java Virtual Machines (JVM) [26]. With this, it is possible to get a loaded class and method lists in JVM and use it directly [15]. Figure 1 is an example of code using Java reflection techniques.

In the reflection code, the *Class.forName* method is to obtain class objects and create object through the *newInstance* methods. After that, set up the method contained in the class using *getMethod* and finally can call the method by Invoke call. As a result of using reflection, inside of class and method data can be changed with in run-time. It's kinds of useful function for developing, but malicious application want to use this for calling other method containing additional behaviors. So, they load malicious class and method by execute ClassLoader API And call them in run-time By doing this techniques, malicious application can avoid Static analysis.

2.2 Dynamic Analysis

Dynamic Analysis is designed to analyze Android Runtime (ART) when the target is running. Android applications can be dynamically analyzed by logging API function calls and responses to broadcast events and intents. There are many tools such as Xposed and Frida for API function invocation and response collection. There are several input generation tools for dynamic analysis, such as monkeys, but random-based methods have low code coverage. An efficient approach is needed to increase the scope of code. Therefore, accurate and efficient input generation tools are essential for dynamic analysis. Researchers designed user-interface-based methods such as DynaLog and DroidBot. The input generation tool reads and analyzes UI components to increase the scope of code application by pressing a button or typing in a text field.

2.3 Xposed

Xposed [19] is an Android application hooking tool that enables dynamic code modification while running. For example, you can hook a result of API call inside value. In case of getting phone number from calling *getline1number* method, originally the result value is your phone number. However, it can be empty using Xposed. Not only modifying return value, But can see the parameters or making exception while each app running.

In addition, for hooking the sdk information using Xposed, all the parameters and return must be same as target API signature as shown in Fig. 3. Or they might cause exception while hooking, however due to many SDK versions, it is quite difficult to match those APIs for every versions. So we manually investigated for every version of targets and separate the API according to each version, in order to match for every cases.

Figure 2 shows the difference of booting process. When Android OS system booting, zygote process is created in the init process. Zygote [11] is a key element of the Android system and contains core libraries. And using app process required class can be load in zygote. Also all applications are forked by zygote, so the applications have core libraries which zygote contains. Same with this step, the Xposed extends app process to add a jar file which named as XposedBridge to the class path, which invokes the method at a specific point during execution to enable modification of the application behavior.

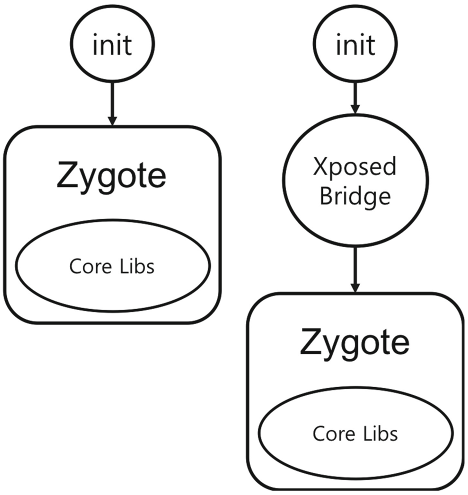

Fig. 2. Normal booting VS Xposed booting in Android

```
new HookUtil().createInstance(SmsManager.class)
    .addClassLoader(loadPackageParam.classLoader)
    .hook("sendTextMessage"
        , String.class
        , String.class
        , String.class
        , PendingIntent.class
        , PendingIntent.class)
    .setOnHookListener((param, methodName) -> {
        String dstAddress = (param.args[0] == null) ? "" :
            param.args[0].toString();
        String srcAddress = (param.args[1] == null) ? "" :
            param.args[1].toString();
        String data = (param.args[2] == null) ? "" : param.args[2].toString();
        Map<String, String> content = new HashMap<>();
        content.put("dstAddress", dstAddress);
        content.put("srcAddress", srcAddress);
        content.put("data", data);
        new Util().log(methodName, content);
    });
```

Fig. 3. SDK hooking example using Xposed

3 Related Works

3.1 APK Reverse Engineering

Figure 4 represents the file structure of the Android Package Kit (APK). APK can be extracted easily through the Android Debug Bridge (ADB) [6]. In addition, APK has following the Zip format, which makes it easy to decompress with apktool [20]. And after extraction using the apktool, there are classes.dex, resource, libraries, assets, META-INF.

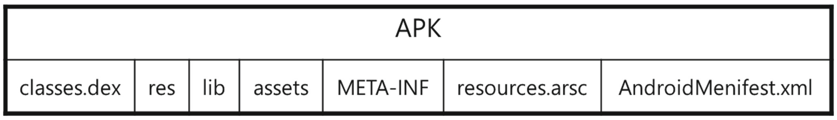

Fig. 4. Android APK structure

APK Structure. Classes.dex is a file that aggregates all class files and converts them into byte codes for Android Dalvik virtual machines to recognize. Res is a folder in which all non-compiled resources exist. Resources include image files, xml files, and so on. Lib is the folder where the library is collected. This folder contains so-files compiled for each process created with Native Development Kit (NDK) [9]. Assets is a folder that contains information about applications that can be managed by Assets Manager. META-INF is a folder related to signatures. Inside of MANIFEST.MF, there are CERT files. These files store signed values using SHA1 and base64 [14]. Also, signature files can be decoded using the keytool. Resources.arsc is a file that records information about resource files. The type and id information of various resource files of resfolder is stored. AndroidMenifest.xml [8] is a xml file for managing applications. The file specifies the application's permission settings, Android component information (e.g., Service, Int, Activity, Receiver, Provider), and Android version. Table 1 shows the overall structure of APK.

Among those of files, the classes.dex is actually executed file in Android system. It contains compiled source code inside, and we can decompile this file using dex2jar [18] and Jd-Gui [1]. With this static analysis tool, It is possible to get readable source code of APK. Attackers use these methods to extract code and then put malicious code inside to attack. Apktool also has the ability to repackage modulated code easily again. Once repackaged APK file is distributed via the Third-party, then it will be the malware.

3.2 API Hiding Technique

The API hiding technique is based on source code. Developers do not want to break down their own applications. Thus, there are many kinds of hidden methods in the Android world, and we have already mentioned one of the hidden methods, the Java reflection, in Sect. 2. Malicious applications also use this technique by using the name of protecting malicious behavior source codes. Therefore, for proper analysis, the files that apply to Java Reflection must be extracted and analyzed before drawing the appropriate call graph [22].

4 Proposed Scheme and Implementation

In this section, We will talk about How we implement our extraction system. Normally, Malicious applications use dynamic loading techniques using DexClassLoader with executable some files in device, such as DEX, JAR, ZIP, APK, etc.

Table 1. Dynamic loading API method and parameter.

Name	Description
classes.dex	Files converted class files into byte codes for recognition within the Android Dalvik virtual machine
res	A directory aggregates non-compile images and xml resources
lib	Directory contains library files, which are compiled with NDK
Meta-INF	A directory related to signature. It contains MANIFEST.MF, CERT.SF, and store signature encrypted with SHA1 and base64
resources.arsc	File that record information about resource files. Store types and ids of resource files located in res directory
assets	A directory aggregates application's information that can be managed by AssetsManager
AndroidManifest.xml	An xml file for managing applications, specifying the application's permissions, component information such as content, services, and activity, and information about the SDK version

Those files can be installed when APK downloading time or comes from remote locations while runtime. Also, they can use reflection to execute sub loadable files. However, intelligent malwares dynamically loads the files and then delete it to avoid being analyzed. So in this paper, we propose extraction algorithms to solve the problems of dynamic loading behavior which contains prevention of delete files case.

4.1 Dynamic Loading API and Java Reflection

Originally, DEX dynamic loading technique is used to cover up DEX file's limitation. In one DEX file, it cannot contain method over 65536 [28]. It means when we develop a application, we cannot use the number of method more than that. So Android allow developers to use multiple DEX files in on application. And it can be loaded dynamically in run-time. However, malicious applications using dynamic loading techniques for hiding source code with same as normal application.

Class.forName is the most important method for Java reflections. The method is used to extract the DEX with a dynamic loading technique and then place the class in the Class object. This feature allows you to detect which class the application actually runs. Through the class extracted from the Class.forName method, you can find out the class that actually was loaded, and hook the getDeclared-Methods method to get a list of the methods that were called directly by dynamic loading. This allows you to identify the method names in the class using the Java reflection technique, which can not be confirmed by static analysis, and extract information about the constructor and field.

Table 2 shows the method for dynamic loading. DEX Path is a parameter used for loadable DEX file path. Optimized Directory is location of created an ODEX file [25]. The Optimized Dalvik Enable (ODEX) is an executable optimized DEX for each system that runs. It generated when application is built. Library Search Path is a parameter used to set the library which related to loaded DEX. Parent is parent classloader.

4.2 Hook with Magisk and EdXposed

In previous Sect. 2, the Xposed tool can hook Android API method in runtime. With this tool, you can easily hook classloader's class or method. If you hook the classloader then we can check the path of loadable DEX file and get it with dynamic analysis. Using Xposed in device give us a lot of benefits. But, there is prelimitation to use the Tool. Each android app is separated by sendboxing technique, cannot access or execute the other app's private data, storage and components. So, the Xposed require root privilege to hooking application method. And another Key point of Xposed hooking is developers should have to use exact method name and parameter types for hooking API.

Magisk [24] is a tool developed by topjhonwu and is used for Android device rooting. Unlike conventional rooting, it is possible to provide root access without changing or replacing the image of the existing system. It can also be linked to external programs to provide various functions together. The main point of the Magisk is mirroring original system. First, mirroring system directories to specific directories and change root mount points. Then, reboot android device. After that, automatically changed the root directory to the new mount point and create the /system, /data, and /cache directories as subdirectories based on the mirrored directory. The directory is mirrored to the existing system, where changes and manipulations are carried out and applied together. The biggest advantage of using Magisk is that it can bypass the SafetyNet provided by Google [5]. Google's SafetyNet is a fairly powerful environmental detection API that collects information about the environment in which the app runs and authenticates itself. This allows the integrity of the system to be verified, and all the rooting and emulator detection [7] are possible. Therefore, using the attest function among the SafetyNet APIs, you can accurately detect the device environment and obtain confirmation from Google for the integrity of the device. In addition, in cases other than the previously used stock boot image, Android Open Source Project (AOSP) [3] build and use cannot pass through SafetyNet.

Among the existing studies, dynamically loaded files were being extracted in various ways. In particular, the approach of building a new OS by changing AOSP [27] or using various tools for memory analysis [29] has become more likely to not work correctly if the app is using SafetyNet, and in future papers, it may be necessary to consider how to analyze apps using SafetyNet. Therefore, using the Magisk created to bypass SafetyNet can proceed with the correct detection when using other supported tools. Xposed can also bypass SatetyNet and hook APIs by using Edxposed, an open source that is changed for use by Magisk.

Table 2. Dynamic loading API method and parameter.

Method	Parameter
DexClassLoader	(dexPath, optimizedDirectory, librarySerchPath, parent)
BaseDexClassLoader	(dexPath, optimizedDirectory, librarySerchPath, parent)
PathClassLoader	(dexPath, librarySerchPath, parent)
OpenDEXFile	(sourceName, outputName, flag)

On the other hand, the SafetyNet API is also a kind of API. You can change the failed result to Success by hooking the result value of API used for Test. Simple implementation is possible using existing Hooking tools. However, Google also has an algorithm to verify the results received from SafetyNet using backend server to ensure that the values are not forged. If the app is implemented to validate and operate the result values of the attest API on the designated server, the Magisk is the only way to bypass SafetyNet as a result of the investigation so far. Therefore, the Magisk tool was installed at the actual terminal to hook up the dynamic loading API, and Edxposed [2] was installed to configure the environment.

4.3 File Extraction Algorithms

Figure 5 is the dex file extraction algorithm proposed in this paper. In the previous section, we identified the APIs offered by Android to perform dynamic loading and configured an environment with Magisk and Xposed to extract DEX files. After applying the developed environment to the actual device, use the monkey tool, a program that automatically executes the app, to operate the app randomly. Monkey [4] is a tool that can help you turn over Activity by randomly clicking on the UI of the Android app. Therefore, if dynamic loading-related APIs and reflection-related APIs are called during the execution of the app to be analyzed, refer to the first parameter of the function to check the location of the DEX file loading. In such cases, the loaded path is then recorded and the algorithm is constructed so that it can be extracted at once at the end of the analysis.

On the other hand, there are many malicious codes that erase files after dynamic loading. As described earlier, the source code acquisition is difficult and is one of the factors hindering the analysis. Therefore, if an app deletes a loaded file using APIs that delete a specific file, it can block it using Xposed and copy it to another path for storage.

(1) Monitoring API Calls which use ClassLoader
(2) Hook the dynamic loading methods
(3) Extract DEX file location in DEX Path parameter

(4) Check DEX files are deleted or not
(5) If delete executed, then prevent delete command
(6) Extract DEX file until analysis finished

By default, extracted DEX files are stored in the application's default path. Apps can store and retrieve data without authorization in the /data/data/packageName directory, which is the location granted during installation process. If stored in an external storage device, they can be used only with permission. On the other hand, the data/local/tmp path can be used as a directory provided to store temporary files. Therefore, extracted DEX files are designed to be moved to a temporary path, stored, and analyzed.

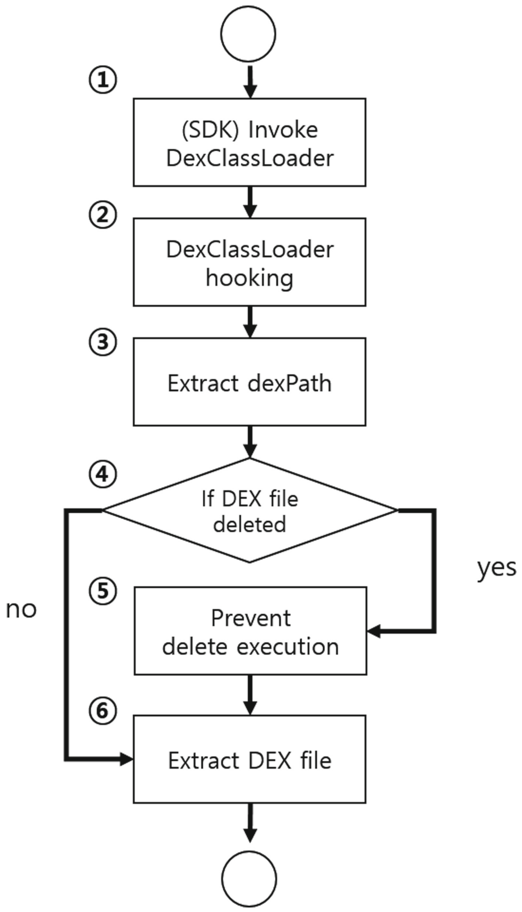

Fig. 5. File extraction algorithms in SDK area.

5 Evaluation

In this section, the performance of the implemented algorithms is evaluated in several ways. The devices used in the evaluation process were analyzed through Android Nexus 5 (Android version 5.1). AMAaaS-supplied application was used to collect the apps that would be tested first. AMAaaS [21] is a web-based Android analysis platform that provides basic static analysis information and providing APIs that run and analyze apps using Android container environments as a result of dynamic analysis. API information and sequence executed can be checked using this method.

Application that was provided by AMAaaS is an application set that was collected for one year from January 2018 to January 2019. Users uploaded the app to analyze the app and check the malicious code. The total number of apps collected is 1,323. The application was classified as Benign and Malware for later analysis, and uploaded to Virustotal [16] for verification. And the app was executed for 3 min and the results were analyzed using logs generated in the process. Monkey tool was applied to collect as many logs as possible, and the analysis results are as follows.

5.1 Performance Result

The content that was carried out before the classification of collected apps as malicious or not is the dynamic loading ratio of apps. Although the apps collected are not actually apps developed for that year, users can upload them through the device, so they can confirm that they were recently used. Follow by Table 3 about 25% of the apps collected used dynamic loading techniques. Nowadays, most apps are using many dynamic loading methods to avoid basic static analysis, including malicious apps, and they have been able to see approximate numbers through the analyzed values.

Table 3. Dynamic loading rate and extraction rate.

	Number (#)	Percentage (%)	Extraction success (Percentage)	Extraction fail (Percentage)
Dynamic load	322	24.3	259 (80.4%)	63 (19.6%)
Single DEX	1001	75.7	-	-
Total	1323	100	259	63

Subsequently, the collected apps were actually activated and dynamic files could be extracted. The analysis found that approximately 80% of apps were able to perform static analysis but could actually be powered up and subject to extraction. Among the apps collected, many apps were unable to run due to contamination or tampering with the DEX file if the **AndroidManifest.xml** file did not exist. In addition, we identified apps that do not run because the value of the **Minimum SDK** set in the app is higher than the actual device. With the exception of these, 259 applications were used for actual testing.

Table 4. Deleted rate after execution.

	Number (#)	Percentage (%)
Deleted	24	9.3
Alive	235	90.7
Total	259	100

Table 5. Malicious rate of loading DEX.

	DEX state	Number (#)	Percentage (%)
Malicious	Deleted	24	60.2
	Alive	132	
Benign	Deleted	0	39.8
	Alive	103	
Total		259	100

We used 259 selected apps to see if dynamic loading actually takes place and then summarize the statistical results on how many actual deletion of loaded files takes place. According to the Table 4 approximately 10% of apps or less were performing commands to delete dynamically loaded files. If you check the results, you can see that about 1% app actually deletes it to protect the source code or to hide malicious behavior based on the entire app that is not a very high number.

To check the distribution of apps that were last deleted, we divided the Benign app and the Malicious app into tables. Table 5 shows that among apps classified as real benign, dynamic loading is performed and the results are not deleted. It was finally possible to confirm that all 24 apps that were deleted were only done in applications that were separated by malicious.

6 Limitation

The original goal of the this paper was to identify malicious behavior using dynamic loading techniques among apps classified as benign. Benign apps analyzed using the proposed method are using dynamic loading techniques but have not been deleted. The previous analysis confirmed that the apps that proceed with deletion were malicious applications with high probability. On the other hand, the extracted files were verified using the virus total using the extracted results for files loaded by the benign app, and the malicious behavior was not found.

The first problem is if the code are not executed which call dynamic load then it cannot extract the loaded file. Monkey tools used to increase code coverage cannot currently bring higher code cover compared to other tools such as UI-automation [10] and DroidBot [12]. However, the data set what we use could be extracted and stored because dynamic loading techniques were used immediately when apps were executed.

Second problem is that if the application has not yet been found, but the code to find the Magisk app and stop the operation is inserted, the extraction is not possible for the app. Magisk app basically offers a technique called magisk-hide and root-hide. But if you look at the source code of github, you can see which files exist in which path. This information is fully detectable, especially su-file and services are installed and used inside the data directory.

7 Conclusion

In this paper, for applications using dynamic loading techniques, the DEX file extraction method is designed and implemented using Magisk and Xposed. Previously, it is impossible to analyze if changing Android OS or extract dynamic files using emulator when using SafetyNet. Therefore, direction was provided to solve this problem, and the application of deleting loaded DEX files was also implemented to limit deletion behavior and extract target files. Subsequently, it was finally confirmed that most of the apps that perform the acts were implemented in applications that include malicious behavior.

On the other hand, the app did not solve the shortcomings of dynamic analysis that must be executed to extract the application's dynamic loading file, and there are disadvantages that cannot be analyzed if the app implements code that detects the Magisk app itself and determines its operation. Nevertheless, if the code was executed, the entire loaded file could be extracted and the source code obtained without any problems. It is also expected that the detection and extraction of the actual device will enable the execution and analysis of as many applications as possible, thus contributing to detecting malicious behavior that could not be analyzed in static analysis.

References

1. Dupuy, E.: JD-GUI (2019). https://github.com/java-decompiler/jd-gui. Accessed May 2019
2. ElderDrivers: EdXposed (2019). https://github.com/ElderDrivers/EdXposed. Accessed May 2019
3. Google: Android open source project (2004–2019). https://source.android.com/n. Accessed May 2019
4. Google: Monkey (2016–2019). https://developer.android.com/studio/test/monkey. Accessed May 2019

5. Google: SafetyNet (2017–2019). https://developer.android.com/training/safetynet/attestation. Accessed May 2019
6. Google: Android debug bridge (2019). https://developer.android.com/studio/command-line/adb?hl=ko. Accessed May 2019
7. Google: Android virtual device (2019). https://developer.android.com/studio/run/managing-avds. Accessed May 2019
8. Google: Androidmanifest.xml (2019). https://developer.android.com/guide/topics/manifest/manifest-intro?hl=ko. Accessed May 2019
9. Google: NDK (2019). https://developer.android.com/ndk. Accessed May 2019
10. Google: UI Automator (2019). https://developer.android.com/training/testing/ui-automator. Accessed May 2019
11. Google: Zygote (2019). https://blog.codecentric.de/en/2018/04/android-zygote-boot-process/. Accessed May 2019
12. Honeynet: DroidBot (2019). https://github.com/honeynet/droidbot. Accessed May 2019
13. IDC: Smartphone market share (2019). https://www.idc.com/promo/smartphone-market-share/os. Accessed March 2019
14. Kanwal, M., Thakur, S.: An app based on static analysis for Android ransomware. In: 2017 International Conference on Computing, Communication and Automation (ICCCA), pp. 813–818. IEEE (May 2017)
15. Li, L., Bissyandé, T.F., Octeau, D., Klein, J.: Reflection-aware static analysis of android apps. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 756–761. IEEE (September 2016)
16. C.S.I. Limited: Virus total (2011–2019). https://www.virustotal.com/. Accessed May 2019
17. McAfee: McAfee mobile threat report q1 (2019). https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf. Accessed March 2019
18. Panxiaobo: Dex2jar (2019). https://sourceforge.net/projects/dex2jar. Accessed May 2019
19. rovo89: Xposed (2019). https://repo.xposed.info/module/de.robv.android.xposed.installer. Accessed May 2019
20. Ryszard Wiśniewski: APKTool (2010–2019). https://ibotpeaches.github.io/Apktool/install/. Accessed May 2019
21. S4URC: AMAaaS (2018–2019). https://amaaas.com/. Accessed May 2019
22. Shan, Z., Neamtiu, I., Samuel, R.: Self-hiding behavior in android apps: detection and characterization. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 728–739. IEEE (May 2018)
23. Statista: Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2018 (2019). https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems. Accessed March 2019
24. topjohnwu: Magisk (2018–2019). https://github.com/topjohnwu/Magisk/releases. Accessed May 2019
25. Wan, J., Zulkernine, M., Eisen, P., Liem, C.: Defending application cache integrity of Android runtime. In: Liu, J.K., Samarati, P. (eds.) ISPEC 2017. LNCS, vol. 10701, pp. 727–746. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72359-4_45
26. Wikipedia: Java virtual machine, 2019 (2019). https://en.wikipedia.org/wiki/Java_virtual_machine. Accessed March 2019
27. Wong, M.Y., Lie, D.: Tackling runtime-based obfuscation in Android with TIRO. In: 27th USENIX Security Symposium, pp. 1247–1262 (2018)

28. Yang, W., et al.: AppSpear: bytecode decrypting and DEX reassembling for packed Android malware. In: Bos, H., Monrose, F., Blanc, G. (eds.) RAID 2015. LNCS, vol. 9404, pp. 359–381. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26362-5_17
29. Zhang, Y., Luo, X., Yin, H.: DexHunter: toward extracting hidden code from packed Android applications. In: Pernul, G., Ryan, P.Y.A., Weippl, E. (eds.) ESORICS 2015. LNCS, vol. 9327, pp. 293–311. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24177-7_15