Automated Detection and Classification for Packed Android Applications

Yibin Liao*, Jiakuan Li†, Bo Li*, Guodong Zhu*, Yue Yin*, Ruoyan Cai*
Department of Computer Science, University of Georgia
* {liao, bo, guodong, yin, ruoyan}@cs.uga.edu
† jiak94@uga.edu

Abstract—Android packing services provide significant benefits in code protection by hiding original executable code, which help app developers to protect their code against reverse engineering. However, adversaries take the advantage of packers to hide their malicious code. A number of unpacking approaches have been proposed to defend against malicious packed apps. Unfortunately, most of the unpacking approaches work only for a limited time or for a particular type of packers. The analysis for different packers often requires specific domain knowledge and a significant amount of manual effort.

In this paper, we conducted analyses of known Android packers appeared in recent years and propose to design an automatic detection and classification framework. The framework is capable of identifying packed apps, extracting the execution behavioral pattern of packers, and categorizing packed apps into groups. The variants of packer families share typical behavioral patterns reflecting their activities and packing techniques. The behavioral patterns obtained dynamically can be exploited to detect and classify unknown packers, which shed light on new directions for security researchers.

I. INTRODUCTION

Being the most popular mobile operating system, Android dominated smartphone market with a share of 82.8% until 2015 [1]. The rapid growth of Android application economy brings a great profits for developers [2], meanwhile, it causes a series of malicious tampering, code injection, and plagiarism issues [3]–[5]. Legitimate app developers adopt various code protection techniques to guarantee their labor and profits. Packing is one of the effective and efficient code protection techniques, and is getting an increasingly use nowadays [6].

Although packing techniques are initially designed to protect apps from being reversed, modified, and repackage, malware writers are making use of these benefits to hide their malicious code in order to evade malware detection. A huge growing percentage of packed Android malware has been reported by the AVL anti-virus team [7]. Since packers usually adopt complex anti-analysis defenses, recent anti-virus could not perform effective analysis task on packed code, and thus are not able to detect those packed malware automatically. Therefore, a number of unpacking approaches have been proposed recently [8]–[10].

All of the unpacking approaches are focusing on how to automatically recover original executable. However, packers are evolving frequently, most unpacking approaches only work for a limited time or for a particular type of packers. Then researchers have to analyze the newly packing techniques. Unfortunately, most of the analysis works require specific domain knowledge and are heavily based on manual efforts, which are slow and tedious.

In order to be effective in analyzing large amounts of packed Android apps, we need to be able to categorize them into groups and identify their respective families. In addition, such grouping criteria may be applied to new packers encountered on Android platform in order to identify whether they belong to a known packer family or constitute a novel packer strain.

The goal of our work is to identify and classify packed Android apps automatically. We studied the packing techniques used by different packers appeared recently, analyzed the difference between packed and unpacked apps. Based on the observation, we designed an automatic detection framework to identify packed apps. It performs both static and dynamic analysis on each app and compare the difference between the two analysis to determine whether this app is packed. Then we propose a two-layer classification: coarse-grained classification and fine-grained classification. Specifically, we look at which part of the code from an app is packed in coarse-grained classification, and propose the idea of detecting the typical behavioral patterns obtained dynamically to classify the packers in fine-grained classification.

Correspondingly, our contributions in this paper are:

- We have implemented a detection module that is capable of identifying packed app from unpacked one automatically. This is the first automatic detection for packed Android apps and can evade most anti-defenses of packers.
- We proposed approaches to extract the behavior pattern of packers that reflect their activities and packing techniques.
- we designed a framework to automatically classify packed apps on Android platform based on the behavior pattern of packers.

II. ANDROID PACKING SERVICES

An app is packed means its original executable files (i.e., DEX files) are hidden or transformed to an encrypted or obscured form so that we cannot easily reverse, modify, and repackage. Packing will not affect the normal execution semantics of the app. We investigated several commercial
packers that provide online packing services (Ali, Bangcle, Ijiami, Tencent, etc.), and summarize the common anti-analysis defenses used by them as the following:

1) Code Obfuscation: Obfuscation aims at creating obfuscated code that is difficult for humans to understand. Obfuscation techniques include modifying names of classes, fields, and methods, reordering control flow graphs, encrypting constant strings, inserting junk code, etc. Packers usually implement obfuscation for major functions in native code, and then invoke native code method through Java native interface (JNI).

2) Anti-debugging: Android is based on Linux kernel. In Linux, one process can attach to another process for debugging. Packers usually insert anti-debugging code stubs (e.g., attach to themselves using ptrace) to interfere dynamic analysis based tools (e.g., gdb). Moreover, advanced packers can check the running environment, such that a packed app can crash if it’s running in the emulator or rooted system.

3) Bytecode Hiding: For Android apps, Java source code is finally compiled to Dalvik bytecode and stored in DEX file. Android allows apps to load codes from external sources at runtime. To leverage this feature, packers usually encrypt original DEX file as an external file, and insert its owner DEX file as a shell or guard. During the execution, packer’s decryption stubs, which implemented in native code, will decrypt the original Dalvik bytecode and then load it into memory. Moreover, packer’s native components can modify the metadata of original DEX file through JNI during the execution. This kind of modification doesn’t affect the normal execution of the app. However, it significantly affects certain analysis tools.

III. DESIGN

With different types of packers, and the vast range of packed apps within each type, it’s important that every packed app can be easily distinguished and unambiguously classified. Therefore, we propose to design an automatic detection and classification framework to identify and classify packed apps.

A. Detecting Packed Android Apps

Most of detections for packed apps are based on manual analysis such as identifying unique file names and native .so libraries inserted by packer. Manual analysis is slow and tedious, which only works for known packers. We propose to design a detection module that automatically identify packed apps. One of the major challenges of this work is to evade the complex anti-defenses adopted by packers. To thwart this challenge, we perform both static and dynamic analysis on each app, and compare the difference between the two analysis to determine whether this app is packed.

1) Static Analysis: In the static analysis phase, We first extract package name and launchable-activity name from the app. The package name is used as a filter in dynamic analysis and will be sent to the device. The launchable-activity name is used to start the app automatically. Then we use the decompiler tool Baksml to parse the app and convert its DEX files to a series of smali files, which contain human readable assembly language representing Dalvik bytecode. Finally we extract the class name of each class from the smali files.

2) Dynamic Analysis: In the dynamic analysis phase, we instrument Dalvik Virtual Machine (DVM) to monitor the execution of this app. This monitoring is a compilation time code injection instrumentation. We didn’t modify any of the APIs, only insert our own code to capture the class loading information. Thus, it’s very difficult to be aware of by packers’ anti-analysis measures. We also deployed the framework on a standard Android device, which can evade typical emulator detection of packers. This guarantees a very trustworthy analyzing environment.

Since each class should be loaded into memory before it can be used, we explore the class loading process to collect class information at runtime. We select Dalvik_dalvik_system_DexFile_defineClassNative as the key function for injecting instrumentation code. DEX file is parsed into a data structure called DvmDex, and then initialized to DexFile in memory. Since all class loadings will call this function, including apps running background, we first implement a filter based on the package name captured in static analysis and search the p_id of the app to be analyzed. After locating the correct DexFile object with the p_id, we obtain the class index object DexClassDef by passing DexFile as an parameter of the dexGetClassDef function, and then invoke the method dexGetClassDescriptor to extract the class name of each class.

3) Comparison: Since the class name is unique, we only extract full class name as a string for each class in the static and dynamic analysis. We sort the class name strings in order and then compare their difference from the two analysis. If we find class names that only exist in dynamic analysis, this means the app contains packed code. Otherwise, the app is unpacked.

B. Classifying Packed Android Apps

Since we have identified which Android apps are packed, we now want to categorize them into groups that reflect similar types of behaviors. We propose a two-layer classification, coarse-grained classification and fine-grained classification as following:

1) Coarse-grained Classification: In coarse-grained classification, we look at which part of code is packed. Based on our observation, many apps contain third-party libraries and only the libraries are packed. We consider this type of packed apps is partially packed or framework packed. The app which its entire code is packed will be considered fully
packed. For the fully packed apps, we can extract much more classes in the dynamic analysis than static analysis. Such that we extract 4806 classes in dynamic analysis but only 6 classes in static analysis. We evaluated 200 fully packed apps, more than 99% of them has the ratio (number of classes in static / number of classes in dynamic) less than 20%. Thus, we can set a threshold as 20% to classify framework packed and fully packed apps. If the packed app is framework packed and the framework libraries are from well-known legitimate publishers, we may ignore the analysis for that app.

2) Fine-grained Classification: As mentioned previously, packers usually encrypt original DEX file to external data format, insert decryption stubs and customized loader into the app. The original bytecode of the app will be released during the execution by the decryption stub and loaded into memory by packer’s customized loader. Different packers may implement different code releasing and loading strategies. For instance, some packers reload the DEX data into in-consecutive memory regions and modify relevant pointers that point to the data to prevent direct memory dump based unpacking; a type of packers deploy a two-layer decryption stub. It first releases a decrypted DEX, which doesn’t contain the original bytecode. However, it contains a second decryption stub responsible for decrypting original bytecode of a method once it’s invoked [8].

All behaviors performed above can be eventually represented as a sequence invocation of native activities, system calls, JNI calls, etc. Packers implement similar code release or loading strategy may result similar behavioral pattern. Such as a similar pattern of system calls, or a similar pattern of JNI calls, which make these packers belong to one type of packer. Native code in Android apps is deployed as ELF files, either executable files or shared libraries (.so files). Java code can invoke native code in the following ways [11].

**Exec methods.** Executable file can be called from Java by `Runtime.exec` and `processBuilder.start`. These methods are refer to Exec methods.

**Native methods.** Methods are declared in Java code but implemented in native shared libraries. Java Native Interface (JNI) defines a way for Java to interact with native methods.

**Native activity.** Native code is invoked in native activities using activities’ callback functions, (e.g., `onCreate` and `onResume`), if defined in a native library.

To address the diversity of packers’ behavior and detect the typical behavioral patterns for specific type of packer, we propose to design a native code level instrumentation framework that records all events and operations executed from within native code, such as invoked system calls, native-to-Java communications and Binder transactions as following:

- To monitor system calls, we proposed to implement a Linux kernel module to capture the invoked system calls. Other tools such as `strace` cannot perform effective analysis because of the packer’s anti-debugging.
- To monitor native-to-Java communications including calls to Exec methods, calls to Native methods, native activity callbacks, we proposed to instrument `libjavacore`, `libdvm`, `libandroid_runtime` respectively.
- To monitor Binder transactions, we propose to instrument `libbinder` to capture the class of the remote functions being called and the number that identifies the function.

To classify the packed apps, we can use similar idea we propose in [12]. First, for each app instance $i$, we can convert the behavior patterns into sequential strings according to the execution time, say sequence for system call monitoring $sc_{i}$, sequence for Native-to-Java monitoring $nj_{i}$, and sequence for binder transaction monitoring $bt_{i}$. Such as $S_{sc(i)}$, $S_{nj(i)}$, and $S_{bt(i)}$. Take system call monitoring for example, if the system call sequence of $ith$ instance $sc_{i}$ is read→write→write, then $S_{sc(i)}$ can be written as “RRW”.

The normalized Levenshtein distance can be then used to compute the distance between behavior patterns of app instance $i$ and $k$. We define the normalized Levenshtein distance as following. If we have two strings $S_1$ and $S_2$, the normalized Levenshtein distance $D(S_1, S_2)$ equals to the minimal operations taken to transform $S_1$ to $S_2$, divided by $\max(length(S_1), length(S_2))$. For example, if $S_{sc(i)}$ = “RRW” and $S_{sc(k)}$ = “RRWW”, one operation will be taken to transform $S_{sc(i)}$ to $S_{sc(k)}$, i.e. adding an extra “W”. Then we have the normalized Levenshtein distance between $S_{sc(i)}$ and $S_{sc(k)}$: $D(S_{sc(i)}, S_{sc(k)}) = \frac{1}{\max(length(S_{sc(i)}), length(S_{sc(k)}))} = \frac{1}{2}$. Finally, the overall distance between app instance $i$ and
k will be $D_{total(i,k)} = w_{sc} \cdot D(S_{sc(i)}, S_{sc(k)}) + w_{nj} \cdot D(S_{nj(i)}, S_{nj(k)}) + w_{bt} \cdot D(S_{bt(i)}, S_{bt(k)})$, where $w_{sc}$, $w_{nj}$, and $w_{bt}$ is the corresponding weights applied to distances of behavior patterns of system call, Native-to-Java and binder transaction. Given $n$ app instances, a $n \times n$ distance matrix between each app instance is built in this way and single-linkage hierarchical clustering algorithm can be applied to split app instances into clusters (sets of packed apps).

IV. IMPLEMENTATION

A. Architecture

As shown in Fig. 1, detection module takes a set of Android apps as input, and output a set of packed apps. The static analysis is performed in a desktop computer running Ubuntu Desktop 14.04. We use Baksmai 2.1.0 as a basis for the static analysis. The instrumented DVM is deployed in a Nexus 7 tablet running Android 4.4.3. In the dynamic analysis, Apps will be automatically installed and launched by triggering the launchable-active defined in the manifest file. To save the storage space, apps will be automatically deleted after class names have been extracted.

For the coarse-grained classification, we can maintain a white-list database for existing known packed frameworks as a filter. The unknown framework packed apps and fully packed apps will be further classified in the following fine-grained classification. All three monitor components will be deployed in another Android device. Each monitor component will log the corresponding behavior pattern of the packed app and feed the pattern to the classifier. Packed apps will be classified into groups reflecting similar type of behavior eventually.

B. Benefits

With the framework, it is possible to:

- Reduce the large volume of packed Android apps to be evaluated when analyzing different packers.
- Reflect the specific actions performed by packer and infer the packing techniques without manual analysis.
- Detect unknown packers and reveal their representative behavioral features.

V. CONCLUSION AND FUTURE WORK

In this work, we investigated a series of Android packing services appeared recently, studied the packing techniques adopted by packers and the difference between packed apps and unpacked apps. Our study showed that the complexity of packing techniques and packers’ evolution makes many of the packed apps hard to be detected and analyzed efficiently with existing tools.

Based on our findings, we implemented a detection module for packed Android apps by combining static and dynamic analysis to evade the anti-defenses of packer, and proposed coarse-grained and fine-grained classifications. We designed an automated framework that performs native code level instrumentation to extract the discriminative behavior pattern and categorizes packed Android apps into groups.

Current ongoing work includes kernel module implementation for system call monitoring and instrumentation for native-to-Java communication monitoring.

ACKNOWLEDGMENT

This work is partially supported by National Science Foundation (NSF) under award No. 1319115 and a gift award from Intel Corp.

REFERENCES


