2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

DyDroid : Measuring Dynamic Code Loading and Its Security Implications in Android Applications

Zhengyang Qu, Shahid Alam[†], Yan Chen, Xiaoyong Zhou[††], Wangjun Hong, Ryan Riley[†]
Northwestern University, Qatar University[†], Samsung Research America[††]

Abstract

Android has provided dynamic code loading (DCL) since API level one. DCL allows an app developer to load additional code at runtime. DCL raises numerous challenges with regards to security and accountability analysis of apps. While previous studies have investigated DCL on Android, in this paper we formulate and answer three critical questions that are missing from previous studies: (1) Where does the loaded code come from (remotely fetched or locally packaged), and who is the responsible entity to invoke its functionality? (2) In what ways is DCL utilized to harden mobile apps, specifically, application obfuscation? (3) What are the security risks and implications that can be found from DCL in off-the-shelf apps?

We design and implement DyDroid, a system which uses both dynamic and static analysis to analyze dynamically loaded code. Dynamic analysis is used to automatically exercise apps, capture DCL behavior, and intercept the loaded code. Static analysis is used to investigate malicious behavior and privacy leakage in that dynamically loaded code. We have used DyDroid to analyze over 46K apps with little manual intervention, allowing us to conduct a large-scale measurement to investigate five aspects of DCL, such as source identification, malware detection, vulnerability analysis, obfuscation analysis, and privacy tracking analysis.

We have several interesting findings. (1) 27 apps are found to violate the content policy of Google Play by executing code downloaded from remote servers. (2) We determine the distribution, pros/cons, and implications of several common obfuscation methods, including DEX encryption/loading. (3) DCL's stealthiness enables it to be a channel to deploy malware, and we find 87 apps loading malicious binaries which are not detected by existing antivirus tools. (4) We found 14 apps that are vulnerable to code injection attacks due to dynamically loading code which is writable by other apps. (5) DCL is mainly used by third-party SDKs, meaning that app developers may not know what sort of sensitive functionality is injected into their apps.

I. Introduction

Android is the dominant smartphone OS. In Q2 2015, IDC placed the worldwide market share of Android at 82.48 percent of all active smartphones [27]. However, its open nature and the wide variety of app markets also make it easier to disseminate malware or otherwise untrustworthy apps. In 2016, Mirror reported that up to 10 million Android smartphones had been infected by malicious software [21]. After realizing the severity of the malware threat, Google developed and deployed Google Bouncer [48], a tool that analyzes apps submitted to Google Play [15] and checks them for malicious behavior before publishing them. Other security vendors, such as Bitdefender [11], have released products that are deployed on the client side with static malware analysis.

While most apps are distributed as standalone Android application package (APK) files, the Android platform also supports apps dynamically loading additional binaries at runtime by making use of dynamic code loading (DCL). The usage of DCL is not regulated by the OS, and as such it opens up several possible threats. For example, it can be leveraged to evade malware detection. Our research indicates that DCL is widely used in mobile marketplaces. A thorough investigation of various security-relevant aspects of DCL is thus needed.

By using DCL, a developer can change the behavior of an app at runtime in unpredictable ways. This feature can significantly ease the deployment of malicious code. Malware authors are able to evade the security check of offline analysis systems, such as Google Bouncer, by only executing the malicious code when logical conditions are met [38]. For example, we developed an app which downloads and dynamically loads known malware over the network. This app passed the security check of Google Bouncer, thus demonstrating the practicality of such threats. Although the similar experiment has been conducted by Poeplau et al. [52], our penetration proves that this issue has not been addressed within the recent two years. Moreover, our study of malware samples deployed by DCL in the wild shows instances where the malicious behavior is triggered by the status of the runtime environment, such as availability of a network connection or the system time.

Google's content policy [16] for apps on Google Play specifies that all application updates must go through their market. This policy is not effectively enforced, however, because apps can download and dynamically load new code at runtime without using the market. In fact, our experimental and the measurement results in Section V find numerous apps in the wild that are loading remotely fetched code and violating this policy. Android lacks the ability to track the provenance of code loaded dynamically. Thus, the malicious behaviors and privacy usage in the stealthy channel of DCL are not regulated. Moreover, benign apps that improperly implement DCL can be vulnerable to code injection attacks by other apps on the device; the OS does not enforce any sort of integrity check on dynamically loaded code, and in certain circumstances it is possible for the attacker to tamper with the code to be loaded.

2158-3927/17 $31.00 © 2017 IEEE
DOI 10.1109/DSN.2017.14

While DCL can be the cause of some security problems, it can also be used to protect the intellectual property of Android developers. Some recent studies [64], [67] show that DCL and bytecode encryption can be leveraged to obfuscate an app, which makes it difficult to reverse-engineer with static/dynamic analysis tools. Some security providers, such as Bangcle [10] and Ijiami [17] provide such services to protect the intellectual property of developers, where the whole app's bytecode is encrypted and stored as a private resource, and an app container dynamically loads the bytecode after decryption.

In this work we perform a large-scale measurement of DCL usage in over 46K apps, investigating the following issues:

- **Provenance.** The loaded bytecode can be either packed as static files in the APK file or fetched stealthily from a remote server at runtime. The latter is capable of evading static/dynamic malware detection mechanisms. We are thus interested in the popularity of its usage, despite the fact that it is prohibited by Google Play.
- **Security risks/implications.** Are there any malicious behaviors hidden in dynamically loaded code? Does the usage of DCL in existing mobile apps have vulnerabilities? How is user privacy tracked in dynamically loaded code?
- **Application hardening.** DCL can be used by apps for the purpose of anti-reverse engineering. In the obfuscated app, the bytecode of the original app is encrypted and repacked. The modules of bytecode decryption, code loading, and app lifecycle construction are interposed in the original app's launching procedure. We investigate an app's pattern after obfuscation, popularity, and comparison with other common obfuscation techniques, including native code, lexical obfuscation, Java reflection, and anti-decompilation.
- **Usage in the wild.** How widely is DCL adopted in apps in marketplaces? Does the DCL usage correlate with other application attributes, e.g. number of downloads, average rating, and number of ratings? We also study the source of dynamic code loading within the app itself, whether it is the main application or a third party library. For example, a developer may integrate a software development kit (SDK) related to advertising in order to generate revenue. This SDK may use DCL to load portions of its functionality at runtime. We are interested in the entity responsible for using DCL.

We summarize the following challenges. *(1) Code interception.* We need to log the DCL event and intercept the code loaded. The files containing the binaries may be temporary, which are compiled as intermediate results and will be deleted after being merged with the app triggering the DCL behavior. The app's runtime and our code interception are concurrent in the OS. We thus need to instrument the low-level IO-related APIs to enforce mutual exclusion and intercept those loaded binaries. *(2) Provenance/entity identification.* The Android OS itself does not distinguish whether or not a file in storage is downloaded from the network, meaning that it is non-trivial to determine if a file loaded using DCL was originally sourced from the network. Detecting this case will require making use of flow analysis. In addition, the code loading may be triggered by a third-party SDK or library. Our mechanism must also be able to find out whether it was the developer or the third-party library provider who performed DCL. *(3) Obfuscation identification.* DCL is being actively used for anti-reverse engineering purposes. A proper methodology to accurately detect when an app is obfuscated in this way needs to be developed.

In this work we make the following contributions:

- We develop a framework, DyDroid, which combines both dynamic and static analysis in order to detect DCL and intercept the bytecode and/or native code loaded. The paths to the binaries to be loaded are pushed to a queue, and we instrument the IO-related calls to block file delete and rename operations during the phase of code interception. A flow analysis is implemented in the dynamic analysis, which captures the flow from a URL to a file. DyDroid tracks the call site of DCL behavior by retrieving the element of the Java stack trace. Using this stack trace we are able to differentiate the responsible entity launching DCL. After capturing the dynamically loaded code, DyDroid performs static analysis on the intercepted binaries in order to determine malicious behavior and privacy leakage. In addition, we summarize the general pattern of apps obfuscated with bytecode encryption/loading based on the samples from four mobile app security vendors. The obfuscation pattern involves how the three core components, the app bytecode decryption, DCL, and app lifecycle construction, are organized in an application subclass [4] as the container.
- DyDroid is capable of stable operation with little manual intervention. Various types of exceptions are automatically handled, such as device storage running out. It allows us to be the first to conduct a large-scale measurement of DCL over 46K Android apps. Our measurement tracks the provenance of DCL, including local/remote availability, and the entity. We find the 27 apps that violate the content policy of Google Play by executing the binaries downloaded from the remote server of `Baidu` [9]. Generally, over 85% of DCL is initiated by third-party SDKs or libraries. And the app popularity has the positive correlation with DCL adoption. Moreover, we conduct the first large-scale measurement of various obfuscation methods to understand their distribution, pros/cons, and implications.
- Our analysis demonstrates a number of apps in the wild that use DCL to load malware. We find 87 apps which load malicious bytecode or native code at runtime, making them undetectable to existing antivirus tools such as Google Bouncer or VirusTotal [29]. We have conducted further analysis which reveals that the execution of the malicious code in these apps is triggered by properties of the runtime environment, such as the system time, GPS service availability, and network connectivity.
- We have identified a vulnerability in a number of DCL apps that leaves them open to code injection attacks [52] by other apps on the system. We explore a variant of the code injection vulnerability, where code is loaded from the

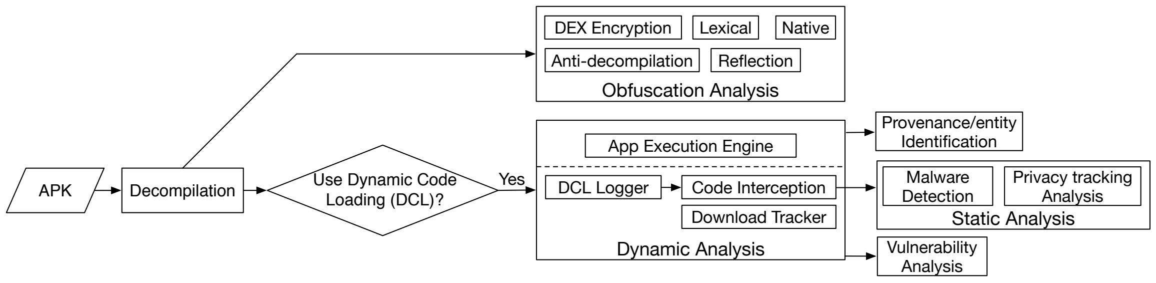

Fig. 1. System Architecture

internal storage of other apps, and we find 7 apps vulnerable to this attack.

The remainder of this paper is organized as follows: Section II presents a brief background. We cover the design of DYDROID and its implementation in Sections III and IV. Section V presents our measurement results over large numbers of real-world apps with DCL, which is followed by the relevant discussion. We have related work in Section VI. Finally, we conclude our work in Section VII.

II. BACKGROUND

Android apps are written in Java. The classes are compiled to Dalvik bytecode with the tool `dx` and further stored as one file `classes.dex` in the installation package. Each class is loaded and executed in the DVM[1]. Other than the internal static executable bytecode, Android also supports fetching external binaries dynamically. Developers use the class loader provided by Android to load arbitrary executable bytecode, which is stored in files with various formats, such as APK, JAR, ZIP, DEX, and ODEX (optimized DEX). The DEX code is then translated into a performance-optimized version, ODEX. There are two types of basic class loaders `DexClassLoader` [5], and `PathClassLoader` [6]. Apps can also load native code. The APIs in the Java Native Interface (JNI) [18] can be invoked to dynamically load native libraries in `.so` format. Android does not verify the loaded code integrity or have the ability to differentiate whether the file containing loaded binaries is originally packed in the application or downloaded from a remote server at runtime. The binaries can be accessed with diverse methods. For example, an application can even use package contexts to retrieve the classes contained in another application. However, the loading behavior will always be achieved by either using `DexClassLoader`, `PathClassLoader` for DEX code or invoking the APIs `load()`, `loadLibrary()` in the JNI for native code. All DCL goes through one of these points, which provides us with a reliable way to enforce complete mediation in intercepting the loaded code.

[1]Starting with Android 5.0 the Dalvik virtual machine was replaced by ART, an ahead of time compiler. In this work we make use of Android 4.3.1, and thus we discuss Dalvik.

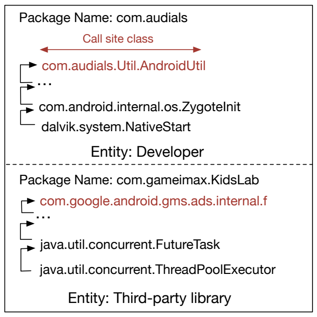
Package Name: com.audials
Call site class
com.audials.Util.AndroidUtil
...
com.android.internal.os.ZygoteInit
dalvik.system.NativeStart
Entity: Developer
Package Name: com.gameimax.KidsLab
com.google.android.gms.ads.internal.f
...
java.util.concurrent.FutureTask
java.util.concurrent.ThreadPoolExecutor
Entity: Third-party library

Fig. 2. Java Stack Trace Element

III. SYSTEM DESIGN

A. System Overview

The architecture of DYDROID is illustrated in Figure 1. An APK file will first be decompiled into an intermediate representation (IR). Then we check if the app creates the class loader to dynamically load DEX code or invokes the APIs related to native code loading. We do not verify the reachability of DCL-related code, only its existence within the app. This step simply serves as a filter to determine which apps to investigate further using our dynamic analysis. Apps containing DCL-related code are then executed and our App Execution Engine is used to log DCL events and track files downloaded remotely during execution. Using this information we are able to determine the provenance of the loaded code (local or remote) and whether the DCL is vulnerable to code injection attacks. The intercepted code will be passed to our static analysis to investigate the existence of malicious behavior and privacy leakage.

We also perform obfuscation analysis by checking the Android manifest file and the availability of basic components against a series of rules to identify whether bytecode loading and encryption are applied to obfuscate the app. The method is also designed to recognize the usage of other anti-reverse engineering techniques, including lexical obfuscation, reflection, native code, and anti-decompilation.

B. Dynamic Analysis

To completely capture loading events, we modify the Android framework. All DCL events an app can use go through `DexClassLoader` or `PathClassLoader` in the DVM,

TABLE I
RULES OF DOWNLOAD TRACKER

source: URL, sink: File	
URL:	
	URL → InputStream
InputStream:	
	InputStream → InputStream
	InputStream → Buffer
Buffer:	
	Buffer → InputStream
	Buffer → OutputStream
OutputStream:	
	OutputStream → Buffer
	OutputStream → OutputStream
	OutputStream → File
File:	
	File → File
	File → InputStream

or `load()` or `loadLibrary()` in the JNI. As such, we instrument these methods to record the following information: (1) path to the loaded file with various formats, e.g., so, APK, ZIP, JAR, DEX; (2) path to the directory storing the optimized version of the DEX code; (3) the call site class of the DCL (the class where the class loader is created). We determine the call site by analyzing the stack trace [20]. An example of this analysis can be found in Figure 2. We record the classes of the sequence of objects whose methods are called when the class loader is initialized, and the top element of the stack trace is the call site class, which is used to figure out whether the developer or a third party library provider launched the DCL. Our DCL logger skips the system binaries, such as native libraries in `/system/lib`, which are provided by security-trusted OS vendors and are thus not in our scope.

When a DCL event is captured, the path to the file being loaded is stored in a queue and logged. In some third-party libraries, such as the Google Ads library, we observed that the files loaded are temporary, meaning they are deleted after the load. As such, fully intercepting the loaded binaries requires enforcing mutual exclusion. We modify the methods related to file deleting and renaming in `java.io.File` to ensure that the delete and rename operations silently fail for files in our queue of dynamically loaded binaries. This ensures the dynamically loaded code remains available for later analysis.

To investigate if a loaded file is packed locally or fetched remotely at runtime, our dynamic analysis includes taint tracking regarding file downloads. As shown in Table I, URL and File are modeled as source and sink. We first instrument the class `URL` to record all the URLs initialized, and the class `URLConnection` with its subclasses, such as `HttpURLConnection`, `HttpsURLConnection`, and `FtpURLConnection`, to track the flow from URL to InputStream. Next, we instrument the constructors, and the methods `read()` and `write()` of the classes `InputStream`, `Reader`, `OutputStream`, `Writer`, including their subclasses in the package `java.io.*`, to track the flows among InputStream, Buffer, OutputStream, and File. The copy and renaming operations are considered as the flows among Files. Each object is represented by type and hash code [19]. In the data flow graph, we search the paths from a URL to a File.

In order to increase the chance that our dynamic analysis engine triggers the DCL event, we employ Fuzz testing [56], [49], [45], [28]. Specifically, a sequence of events is generated and triggered automatically as inputs to UI elements, which invoke the callback functions and Android framework. We utilize the fuzzing tool Monkey [28], which runs on top of a device running the instrumented version of Android 4.3.1. We verify that the DCL-related APIs in Android 7.1 do not change significantly from Android 4.3.1. `DexClassLoader` and `PathClassLoader` remain the same and ART uses DEX to load. The class `Runtime` only adds an API (load0) to load native code. We only need to add hooking to one API to adapt to the latest version of Android. Our system modification thus works well on newer versions of Android.

a) Provenance/entity Information: Poeplau et al. had shown that it was feasible to evade Google Bouncer with DCL [52]. Our experiment indicates that the issue has not been fixed in the recent two years. We prepared a malicious app App$_M$, that is derived from known malware [22]. We submitted this app to Google Play and it was rejected by Google Bouncer. We then implemented a new app App$_L$, which can dynamically load App$_M$ from a server at runtime. The server decides whether or not to send App$_L$ the link to the copy of App$_M$. The app App$_L$ was approved and released on Google Play. We should note that we disabled the malware delivery at the server side during application review and after release. We thus make sure no end user is affected by the malware.

Google has a content policy [16] that apps should not using side channels other than the standard updates to modify the APK binary code. In other words, when using DCL it is only legitimate to load code already packaged into the installation package. Remote fetching new code is not allowed. However, we still found some apps fetching binaries from a remote server at runtime. This technique can ease the application updates for developers. For example, a normal application update can be packed as a DEX file and be pushed to devices instantly when it is ready, bypassing lengthy application review on the store. However, loading the code fetched remotely brings malware authors a stealthy channel to deploy malicious code after app approval by the store. Given the limitation of offline analysis systems [51], [62], the malware detection deployed on mobile marketplaces can be evaded easily, where the malicious code is actually fetched and executed after the application's public release. Moreover, the Android OS currently cannot tell whether the file to be loaded is fetched remotely. Thus, the existing Android ecosystem lacks a mechanism to enforce Google's policy. The DCL logger and download tracker of DyDroid records the provenance information for remotely downloaded files, meaning we can identify which DCL apps are loading code remotely and thus violating the policy.

In addition to tracking local/remote provenance, we can also determine if the DCL event was triggered by the app itself or a third party library included with it. In Java, packages organize classes into namespaces. Classes in the same package can access the package-private and protected members of each other. Android apps inherit this organizational pattern. Each app has a unique application package name that includes the classes from developers, while the third-party libraries are organized with different package names. As shown in Figure 2, the package name can be used to determine if the DCL event was triggered by the main app or a third party library.

b) Vulnerability Analysis: When studying DCL we noticed a potential vulnerability depending on where apps load their dynamic code from. If bytecode is being loaded, then the parameter `dexPath` in the constructors of `DexClassLoader` and `PathClassLoader` specifies the list of files containing bytecode to be loaded. If native code is being loaded, the parameter `libName` of API `loadLibrary()` represents the name of the library containing the loaded code. It will be passed to function `mapLibraryName()` to get the path to the library file given the runtime environment, and the API `load()` does the real job of loading code from the library file.

Under Android, the responsibility for verifying the integrity of the file being loaded is on the developer, who is generally more concerned with functionality than security. Thus, if the loaded code is located on a space writable by other parties, then other apps can replace the file with another, and cause the code to be loaded in the context of the vulnerable app.

Poeplau et al. [52] have previously discussed the problem of dynamic code loading from external storage. As such, part of our analysis checks for this vulnerability in our application set. In addition, we identify another variant of this vulnerability. During vulnerability analysis, we check if the path of the file loaded falls into either of the following categories:

- **External storage.** Prior to Android 4.4, any app is able to modify the contents of external storage without declaring special permissions. This means that if an app performs DCL from a file on external storage (for example, in /mnt/sdcard/[2]), any other app can replace that file. After Android 4.4, apps must declare a special permission in order to write to external storage. This is a common permission, however, and it would not be unusual for an app to have it.
- **Internal storage of other apps.** Android provides each app private internal storage where only that app can create files. However, we have observed that other apps can dynamically load binaries from the private internal storage of other apps. While it unclear why an app developer would want to do this, we noticed that some do. As such, we flag this situation as a potential vulnerability in the apps that load files from the internal storage of other apps, e.g. from /data/data/otherAppPackageName/.

C. Static Analysis

a) Malware Detection: The dynamic code intercepted by our system can be bytecode or native code. Most malware detection systems for Android, however, only operate on bytecode. As such, in order to perform malware detection of our captured samples we make use of the publicly available malware analysis system DroidNative [32], [14], which is able to analyze both bytecode and native code binaries. DroidNative translates the binaries compiled for various platforms, such as ARM and x86, to the platform independent Malware Analysis Intermediate Language (MAIL) [31]. MAIL provides a high-level representation of the disassembled binary program, which includes the specific information such as control flow information, function/API calls and patterns. Given the issue of zero-day malware and malware variants, DroidNative utilizes a learning-based method, and trains a classifier based on the annotated control flow graphs (ACFG) of malware. In the evaluation with traditional malware variants, DroidNative achieves the detection rate of 99.48%. In DyDroid, we train DroidNative with 1,240 apps from 19 malware families which are collected from two sources [69], [22]. We then use the system to detect malware samples from among the dynamically loaded code we intercept. Specifically, DroidNative conducts a subgraph matching on the ACFG and flags a malware when the degree of match is over 90%. When a sample is flagged as malware, we manually verify it in order to reduce the possibility of false positives.

We then go further, and for each intercepted file containing malicious code, we validate whether the loading event can be reproduced under a variety of runtime environment configurations. First, we set the system time to be before the app's release date. Second, we enable airplane mode but intentionally re-enable the WiFi connection. Third, we enable airplane mode to disable all Internet connectivity. Finally, the location service is disabled.

b) Privacy Tracking Analysis: Previous related studies [69], [61] found that Android apps frequently transmit sensitive data to unknown destinations without user consent. However, the severity of this problem remains unclear within DCL. As such, we conduct a static data-flow analysis on intercepted DEX code.

Our data-flow analysis leverages the public system FlowDroid [33], which achieves the high precision 86% and recall 93% in data leak detection. FlowDroid requires the application installation package as input. The manifest file and layout resource are used to locate the app entry points. While we only have the DEX binaries intercepted. Unlike a whole app that has the well-defined components interacting with the system, the loaded code interacts with the app, and an arbitrary class can be the entry point to the loaded libraries. We thus modify FlowDroid regarding the definition of program entry point and remove its dependency on the manifest file and layout resources.

Felt et al. [41] surveyed 3,115 smartphone users about 99 risks and asked the participants to rate how upset they would be if a given risk occurred. Specifically, their survey covered 11 data types regarding user privacy. Additionally, we combine the data types reported in other mobile privacy tracking studies [39], [65], [70], as listed in Table X. The 18 types of privacy are classified into 5 categories:

[2]The example paths to external storage and internal storage are based on the observation in the Android device, where we conduct our measurement.

- **Location.** Android provides the APIs that can be invoked to fetch user's real-time location.
- **Phone identity.** The smartphone identifiers (IMEI, IMSI, ICCID) can be used to recognize the device's identity.
- **User identity.** The user identifiers (phone number, device accounts) can be used to track user's identity.
- **Usage pattern.** The system's `PackageManager` APIs support fetching the apps and packages installed on device. Third parties are strongly motivated to track this type of data. For example, advertisement providers can infer user's interests from the installed apps and selectively push customized ad content.
- **Content provider.** Content providers control the access to a structured set of data. Android has a series of default content providers to manage the private user data, e.g. bookmark in browser, address book, and call history.

For the categories location, phone identity, user identity, and usage pattern, our data-flow analysis checks the invocation of related system APIs and callback functions as the source of privacy tracking. Content provider is identified by URI [7] and organized as an *SQLite* database with schema and table definitions. We thus look up the URI mapped with each privacy-sensitive content provider as the source of data flow. We use the comprehensive list of sinks in the SuSi project [55], which was discovered by a learning approach.

D. Obfuscation Analysis

In addition to the dynamic and static analysis components of DyDroid, we also analyze obfuscation techniques applied to the apps. Based on our observation of obfuscated app samples served by various providers, e.g., Bangcle, Ijiami, 360 [1], and Alibaba [2], we found that these services share a common design based on application rewriting, where code loading and encryption are actively used with the purpose of anti-reverse engineering. An application subclass is implemented as a container. When the application process is started, this container is instantiated before any of the application's components. The class loader created in the container loads the bytecode of other components from an encrypted file packed as a local resource, and the customized code decryption runs before the actual code load. Thus, it is impossible to reverse-engineer the bytecode through static analysis. In addition, some tricks are applied to make dynamic analysis more difficult, e.g., for one app, three distinct processes are started and attach the `ptrace` system call [24] in a loop to prevent the execution from being tracked and controlled externally.

When all the following rules are fulfilled, we identify the obfuscated app with DCL applied based on the decompiled IR as illustrated in Figure 1.

- The attribute `android:name` is defined in the `application` tag in the application's manifest file and a class loader is instantiated in this class. This is the name of the class that executes before any other components of the app. This class (container) injected via application rewriting performs as the new entry point of the whole app. It invokes the added native code to decrypt the original bytecode of the app. Moreover, the bytecode is loaded at runtime and the app lifecycle is constructed within this class.
- Not all the application components declared in the manifest file are found in the decompiled code, and a file in the format that supports bytecode storage is found locally. The decompilation tool used by us is designed for the app organized in the general pattern, where the bytecode is stored in the file `classes.dex`. Thus, the obfuscated DEX code stored as a resource (normally in the `assets` folder) cannot be found and decompiled by the reverse engineering tool. However, all the components to be invoked at runtime need to be declared in the manifest file. We thus treat this mismatch as an identification rule.
- The job of decryption is normally implemented in native code for the sake of security. The application container class that is discussed above uses the JNI to load the local `.so` file to decrypt the bytecode. Although the code decryption may be implemented in Java within the application container class, the decryption process will be exposed to attackers, who can reverse engineer the application container class. In our dataset we did not find any examples of using Java to do the code decryption.

Our mechanism to detect obfuscation techniques includes several methods in parallel with DCL, such as lexical obfuscation, and anti-decompilation. We intend to deliver the compressive measurement results regarding the app obfuscation usage in the current mobile marketplace.

Lexical obfuscation is the process where the identifiers of classes, fields, and methods in the bytecode are replaced with meaningless ones, and thus we need to judge whether each identifier makes sense regarding semantics. We implement a parser to extract the identifiers. We compare the identifiers against a language database constructed from `DBpedia` [12], which dumps Wikipedia for the purpose of Natural Language Processing (NLP). If the identifiers in an application do not correspond to actual words, then we assume the app has been lexically obfuscated. ProGuard [23] has been integrated into Android IDE to provide the lexical obfuscation functionality. One may argue that the ProGuard identifier assignment scheme is rather repetitive and simple to identify, which is straightforward to be used to identify the usage of lexical obfuscation. However, there are several other mobile security vendors having such a functionality, such as Allatori [3], where the app is obfuscated by methods other than the simple identifier renaming. Our method is thus able to recognize the obfuscation usage comprehensively.

Reflection allows a running program to retrieve information about itself and the runtime environment, which can be used to instantiate arbitrary classes, invoke methods, and alter data fields. As with native code, although developers may have various purposes of using these techniques, such as performance improvement, accessing private fields and methods, they do increase the bar of reverse engineering, because they make analyzing the program statically very difficult. But dynamic analysis is still able to recover the execution of the

TABLE II
DYNAMIC ANALYSIS SUMMARY OUT OF 40,849 APPS FOR BYTECODE AND 25,287 APPS FOR NATIVE CODE

	DEX	Native
Failure	**495 (1.21%)**	**330 (1.31%)**
Rewriting failure	454 (1.11%)	133 (0.53%)
No activity	8 (0.02%)	13 (0.05%)
Crash	33 (0.08%)	184 (0.73%)
Exercised	**40,354 (98.79%)**	**24,957 (98.69%)**
Intercepted	16,768 (41.05%)	13,748 (54.37%)

apps obfuscated by this method. We determine if reflection is applied by checking the existence of the related APIs of the package `java.lang.reflect`. Moreover, the usage of native code can be identified by confirming with the output of our dynamic analysis.

Anti-decompilation techniques hinder the reverse engineering tools by making the code appear invalid to them. For example, the programming language pattern lacking the one-to-one mapping from DEX bytecode to the target language. When we decompile the Android apps to IR, we record the apps obfuscated with anti-decompilation techniques.

IV. IMPLEMENTATION

We leverage the open source tool `baksmali` [26] to unpack and decompile the installation package into the IR `smali`. The log of our dynamic analysis and the dumped loaded code are stored in the external storage of the device. If the application does not declare the Android permission WRITE_EXTERNAL_STORAGE, we will rewrite and repack the decompiled version with the permission added to the manifest file. Our DCL logger and code interception rely on instrumenting the constructors of `DexClassLoader` and `PathClassLoader`, the APIs `load()` and `loadLibrary()` in the JNI, and the APIs related to file deleting and renaming in `java.io.File`. The download tracker involves instrumenting the constructor of the class `URL` and the method `getInputStream()` of the class `URLConnection`, including its subclasses. Moreover, the flow among InputStream, Buffer, OutputStream, and File are tracked through the constructors and the methods `read()` and `write()` of the classes `InputStream`, `Reader`, `OutputStream`, `Writer`. We write a script in `Python` to parse the output of download tracker and construct the flow graph of file download.

V. MEASUREMENT

In this section, we will introduce our measurement data set. We then present our measurements results, which mainly answer the following questions. (1) What are the apps loading code in the remote fetch manner that is prohibited by Google Play, and who is the responsible entity? (2) How is the DCL used for app hardening, specifically, obfuscation? (3) What are the security risks/implications of DCL in the marketplaces? The dynamic analysis runs on the Samsung Galaxy Nexus device with the fuzzing tool Monkey running on top of it.

TABLE III
DCL V.S. APPLICATION POPULARITY BASED ON 58,739 APPLICATIONS; NUMBER OF DOWNLOADS; NUMBER OF RATINGS, AVERAGE RATINGS

	#Downloads	#Ratings	Rating
DEX	60,010	2,448	3.91
Without DEX	52,848	2,318	3.77
Native	288,995	8,668	3.82
Without Native	75,127	1,119	3.79

TABLE IV
RESPONSIBLE ENTITY OF DCL OUT OF 16,768 APPS FOR BYTECODE AND 13,748 APPS FOR NATIVE CODE

	3rd-party (#Apps)	Own (#Apps)	3rd-party & Own (#Apps)
DEX	16,755 (99.92%)	50 (0.30%)	37 (0.22%)
Native	11,834 (86.08%)	2,280 (16.58%)	366 (2.66%)

A. Data Set

We randomly collected 58,739 apps and the metadata, such as description, rating, the number of downloads, from Google Play in November 2016. The data set includes 42 application categories. 58,685 apps are successfully unpacked and decompiled into the IR. Those apps which fail in the reverse-engineering procedure are obfuscated. The decompiler crashes and does not generate the `smali` code. We find out that 46K apps have DCL operations in the decompiled IR, where 40,849 apps initialize class loaders for loading DEX code, and 25,287 apps invoke related APIs in JNI for loading native code. We note that the DCL may not be actually executed at runtime. We try to avoid blindly exercising app, given the heavy cost of dynamic analysis.

B. Results

The results of our dynamic analysis are summarized in Table II. The app will be rewritten and repacked with the permission of writing to external storage added, if it is not declared, so as to log the DCL. The anti-repackaging technique is applied to some apps, which crashes `apktool`. Moreover, the fuzzing tool cannot exercise those apps without any *Activity* component. Finally, apps may also crash at runtime due to the implementation fault by developers. We overall successfully exercise 40,354 apps for bytecode and 24,957 apps for native code, among which the DCL of 16,768 apps and 13,748 apps are actually executed and the loaded code are successfully intercepted, separately. We note that the loading of system library is not included in our scope, which is provided by security-trusted OS vendors.

By mining the log of DCL from mobile advertisement vendors, such as AdMob, we find the general pattern of the file path to the bytecode loaded by the advertisement libraries "`/data/data/AppPackageName/cache/ad*`". Within the 16,768 apps whose DCL events are captured and loaded bytecode are intercepted, we find out 15,012 apps execute the binaries related to mobile advertisement. Those files are generated intermediately and will be deleted after being merged with the apps which start the DCL behaviors.

TABLE V
APPS FETCHING BINARIES FROM REMOTE SERVERS

Package name
com.ipeaksoft.pitDadGame, com.xy.mobile.shaketoflashlight
org.madgame.Idom, com.yb.sex.cartoon5
com.jianhui.FJDazhan, com.quwenba.i9300manual
com.rhino.itruthdare, com.xiangqi.fanapp.a1521
com.huijia.moyan, org.mfactory.three.bubble
com.huijia.zuoqingwen, apps.simple.recipe
com.xiangqi.fanapp.a1284, com.ioteam.numbertest
com.avpig.acc, air.com.qqqf.xxywszzy2a
com.seven.chuanyueqinggong, com.game.knyds
air.com.qqqf.xxnjyybdc123456, com.seven.tiancantudou
com.conpany.smile.ui, com.classicalmuseumad.cnad
com.seven.chuanyuegongting, com.seven.mengrushenj
com.nexusgame.popbirds, com.XTWorks.lolsol
com.Long.ButtonsShowAndroid

a) Dynamic Code Loading in Mobile Marketplaces: The number of downloads, the number of ratings, and the average rating are used to quantify the application popularity in marketplaces. From Table III, we can see that the apps with DCL are more popular than the complementary set. There are various factors, which affect the application popularity, and we cannot assert there is any causal relation between usage of DCL and application reputation. However, given the high popularity, the security risks of DCL, such as evading malware detection, code injection vulnerability, and privacy tracking, can thus easily affect large numbers of end users.

b) Provenance/entity Identification: We identify if the third-party or developer is the responsible entity who launches DCL. The results are summarized in Table IV. For both DEX and native code, the third-party SDKs and libraries of over 85% are the actual entities to load code at runtime. Given the difficulty of reverse-engineering the code dynamically loaded, protecting the intellectual property is the possible motivation of deploying third-party libraries using DCL.

With the download tracker in our dynamic analysis, we find out the 27 apps in Table V, which execute the binaries downloaded from remote servers at runtime. For example, the app `com.classicalmuseumad.cnad`[3] downloads two files in the formats JAR and APK from the domain `http://mobads.baidu.com/ads/pa/`. All the DCL events of loading code in the remote fetch manner are initialized by the advertisement related third-party libraries from `Baidu` [9]. The update mechanism of mobile marketplace is a reasonable explanation of the measurement results. Application developer fully controls the update release. SDK vendors cannot predict whether the most up-to-date version of library will be included. In other words, the mobile market channel is not dependable. Fetching the DEX code from a remote server allows the third-party SDK providers to modify the libraries without any constraint, which is prohibited by the content policy of Google Play because it eases the deployment of malware. However, the existing Android OS lacks the ability to track the source of loaded code and is not effective to enforce the policy.

[3]https://play.google.com/store/apps/details?id=com.classicalmuseumad.cnad

TABLE VI
#APPS USING OBFUSCATION TECHNIQUES OUT OF 58,739 APPLICATIONS

Technique	#Apps (%)
Lexical	52,836 (89.95%)
Reflection	30,664 (52.20%)
Native	13,748 (23.40%)
DEX encryption	140 (0.24%)
Anti-decompilation	54 (0.09%)

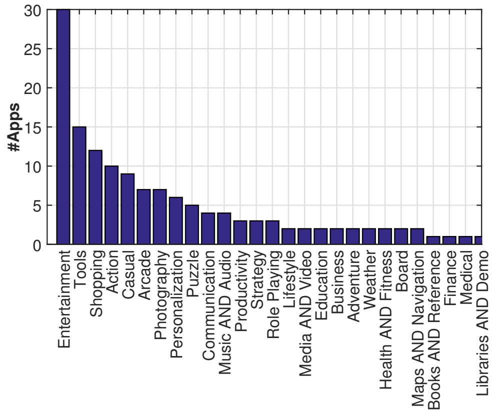

Fig. 3. #Apps with DEX Encryption v.s. Application Category

c) Obfuscation Analysis: The feature of DCL can be used to harden mobile apps. Based on our observation of application samples from mobile application security providers and the general pattern after being obfuscated, we detect the app shielded by DCL and bytecode encryption. Moreover, we also design the method to identify the usage of common obfuscation techniques. Table VI lists how widely each technique is adopted in the apps within our data set.

89.95% apps use the lexical obfuscation. The high adoption rate is expected, as this functionality is included in `ProGuard` and served within Android IDE for free [23]. Even with the high popularity, lexical obfuscation just makes the source code not human readable. For reflection and native code, though they may be used for other purposes, such as performance improvement, accessing private fields, they do increase the difficulty of reverse engineering. 52.20% apps adopt reflection and 23.40% apps include native code.

The adoption rate of DEX encryption method is still low 0.24%. DEX encryption has the decryption functionalities inside native layer, and developers may have the compatibility concern, given the Android fragmentation issue [8]. It is also possible that this technique is relatively new and does not have enough market penetration. Given the 140 apps using DEX encryption, we measure its distribution across application categories, which is illustrated in Figure 3. The categories `Entertainment`, `Tools`, and `Shopping` of apps play a dominant role. We further investigate the functionalities of apps in these categories. The apps in the category of entertainment provide the functionalities of controlling smart TV, where the TV vendors are motivated to protect the communication

TABLE VII
MALWARES DETECTED IN DCL

	Family	#Apps	Sample App (#Downloads)
DEX	Swiss code monkeys	1	com.sktelecom.hoppin.mobile (10,000,000)
	Adware airpush minimob	2	com.oshare.app (10,000)
Native	Chathook ptrace	84	com.com2us.tinyfarm.normal.freefull.google.global.android.common (10,000,000)

TABLE VIII
MALICIOUS CODE LOADED IN VARIOUS CONFIGURATIONS OVER 91 FILES

Configuration	#Files intercepted (%)
System time	72 (79.12%)
Airplane mode/WiFi ON	56 (61.54%)
Airplane mode/WiFi OFF	53 (58.24%)
Location OFF	70 (76.92%)

between smartphone and TV from being reverse engineered. The apps in the category of tools are antivirus apps and those in the category of shopping include the sensitive functionalities of payment, which are both obfuscated for security.

Anti-decompilation disables the reverse-engineering tool `apktool` by using its implementation bug. As `apktool` keeps evolving, the percentage of apps with anti-decompilation capability remains low 0.09%. Next, we discuss the security risks and implications of DCL.

d) Malware Detection: Our measurement investigates the malware hidden in DCL. Overall, we find that 87 apps dynamically load malicious binaries in three malware families from 91 static files [4]. All the detection results are verified by one of the authors manually with the following method so as to guarantee that there is no false positive. DroidNative outputs the ACFG match of the testing binary with the training malware sample. A testing sample will be flagged as malicious if over 90% ACFG of a malware training sample has the parallel match with its ACFG. In most cases, the identified testing samples only differ from the matched malicious samples in the memory addresses, which depend on where the app is loaded. We note that because these malware samples are loaded dynamically, existing detection systems do not detect them. All of these apps are publicly released on Google Play, which means they pass the security verification of Google Bouncer. Moreover, we submitted the malicious samples to VirusTotal [29] (a service that integrates various antivirus products) for scanning and it failed to detect them.

We find the apps loading malicious code in three malware families, and the results are listed in Table VII. For each family, one sample application package name is given for the sake of brevity. We share the full results with all malicious apps in our technical report [5]. One app loads the malicious DEX code in the `Swiss code monkeys` family. It adds the malicious code as a service, and sends IMEI, phone number, and IMSI to a remote site. A remote user is able to send and execute a command, such as app installation, website navigation, adding browser bookmark, sending text message, and blocking test message response. Two apps are found to execute the malicious bytecode in the family `Adware airpush minimob`, where mobile advertisement is pushed to the device via notification. Moreover, shortcuts are placed on users' home screens and browser settings are changed to redirect homepage. There are total 84 apps loads malicious native code in the family `Chathook ptrace`, which mainly targets the two popular chatting apps `QQ` [25] and `WeChat` [30] with millions of downloads. The malicious app tries to get the root privilege first. Then, it attaches the system call `ptrace` to the two apps as the superuser, controls the two apps, and hooks the Java methods related to the chatting window. Finally, the malware leaks the chat history to a remote server.

We further investigate the malicious loading event can be reproduced with different configurations of runtime environment. The results are listed in Table VIII. 19 files of malicious code are not loaded when the system time is set before the app's release date, which can be used to bypass the check during the app review phase. Moreover, we also observe the hide of malicious behaviors when connection or location service is not available, where those logical conditions make it more difficult to detect the malware loaded dynamically.

e) Vulnerability Analysis: The app that loads code from a space writable by other parties is vulnerable to code injections. We classify those apps with risky DCL into two categories: (1) private storage of other apps, (2) public external storage. The results are listed in Table IX. We note that all the vulnerable apps are manually confirmed to make sure that even developer fails to enforce integrity verification on the loaded code. We also check the manifest files of those apps in the second category and make sure they do support the OS version lower than 4.4. 14 apps are found to have risky usage of DCL. Three vulnerable apps have over the 1M number of downloads. Both developers and OS vendors should pay attention to security regulation of DCL.

7 of them load native code from the internal storage of other apps. 6 apps load the native code from the file `libCore.so` in the internal storage of the app *com.adobe.air*. The developers of these apps are different from that of the app *com.adobe.air*, and they blindly trust the integrity of the library provided by the Adobe developer, which introduces the extra attack surface for code injection. Another app *com.devicescape.usc.wifinow* loads the library `libdevicescape-jni.so` from the app *com.devicescape.offloader*, which share the same developer.

7 of them use world read/writable external storage to cache the bytecode loaded. For example the app *com.longtukorea.snmg* stores its bytecode file

[4]One app may have multiple malicious files to load.
[5]http://zyqu.info/DyDroid_DSN.pdf

TABLE IX
VULNERABLE APPLICATIONS DETECTED. APPS IN THE CATEGORY OF EXTERNAL STORAGE ARE VERIFIED AS SUPPORTING THE OS VERSIONS LOWER THAN 4.4

	Category	#Apps	Package name (#Downloads)
DEX	Internal storage of other applications	0	
	External storage (< Android 4.4)	7	com.longtukorea.snmg (1,000,000) com.felink.android.launcher91 (1,000,000) com.ycgame.cf1en.gpiap (100,000) com.fitfun.cubizone.love (100,000) com.fkccy.view (100,000) com.trustlook.fakeiddetector (10,000) com.leduo.endcallsms (100)
Native	Internal storage of other applications	7	com.devicescape.usc.wifinow (1,000,000) com.renren.and02506 (100,000) air.air.com.hi4o.game.Subway_Rushers (10,000) air.com.fire.ane.test.bubblecrazy (10,000) com.renren.wan.war (10,000) air.com.fire.ane.test.ANETest (1,000) com.moeapps (100)
	External storage (< Android 4.4)	0	

TABLE X
PRIVACY TRACKING IN DYNAMICALLY LOADED CODE BASED ON 16,768 APPLICATIONS (L: LOCATION, PI: PHONE IDENTITY, UI: USER IDENTITY, UP: USAGE PATTERN, CP: CONTENT PROVIDER), BROWSER: READ HISTORY & BOOKMARK

Data type	Categ	#Apps	Exclusively 3rd-party (%)
Location	L	254	251 (98.82%)
IMEI	PI	581	576 (99.14%)
IMSI	PI	27	25 (92.59%)
ICCID	PI	8	6 (75.00%)
Phone number	UI	12	10 (83.33%)
Account	UI	23	23 (100.00%)
Installed applications	UP	32	28 (87.50%)
Installed packages	UP	235	231 (98.30%)
Contact	CP	1	1 (100.00%)
Calendar	CP	76	73 (96.05%)
CallLog	CP	32	32 (100%)
Browser	CP	1	1 (100%)
Audio	CP	5	5 (100%)
Image	CP	74	72 (97.30%)
Video	CP	31	31 (100%)
Settings	CP	16,482	16,441 (99.75%)
MMS	CP	1	1 (100%)
SMS	CP	1	1 (100%)

`yayavoice_for_assets_2015101201.jar` in the public directory `/mnt/sdcard/im_sdk/jar/`. Malicious parties can exploit these vulnerabilities by replacing the original file with arbitrary binaries. One app with only the permission of writing to the SD card can misbehave with all the permissions declared by the vulnerable app granted.

f) Privacy Tracking Analysis: We investigate 18 types of privacy tracked in the loaded DEX code with our static analysis. The results are listed in Table X. As we mentioned above, there are 15,012 apps loading the Google Ads library, which has strict control of user privacy and only reads the device settings. However, the remaining 1,756 apps heavily leak various types of user privacy. About 30% apps leak the user's IMEI through DCL. Some highly sensitive types of data, such as location, and installed packages are retrieved in more than 10% apps. As for the responsible entity, the majority of those privacy leakages are exclusively invoked by third-party libraries. The integrated SDK/library is a black-box for the developer, who is not clear about the security risks introduced.

C. Discussion

Using a fuzzing tool in dynamic analysis may have a code coverage problem. We observe that advertisement libraries initialize most of the DCL events and the DCL events are triggered when the app is launched. Our observation matches the results in MAdScope [50]. Thus using monkey is enough regarding the purpose of our measurement.

Regarding the privacy tracking in DCL, users may know and accept it when installing the application. Without this differentiation, it is not possible to know if it is a violation of the promised privacy or not. Deciphering the purpose of privacy tracking is still an open question.

VI. RELATED WORK

a) Dynamic Code Loading & Measurement: Gibler et al. [42] design the system AndroidLeaks, which performs a static analysis to check user privacy leakage among large-scale Android applications. It does not support the analysis of dynamically loaded code. Grace et al. [44] conduct a measurement regarding the privacy and security risks in the advertisement libraries of Android applications, where DCL is defined as a risky flag. Other than simply focusing on advertisement libraries, we include the DCL invoked by developer her/him-self and the third-party libraries for various purposes. Zhauniarovich et al. [68] investigate the usage of DCL and reflection in application update, where the native code is not considered in the security model. DEX encryption together with dynamic loading has been recently found in the application of anti-reverse engineering, and some studies investigate recovering the obfuscated applications [64], [67].

However, there is no study to uncover its usage cases within the Android applications in current marketplaces, such as popularity, general obfuscation pattern, and pros/cons. Rastogi et al. [59] have the system design similar to us, which focuses on the mobile advertisement measurement on the App-Web interface, while we explore the DCL usage. Poeplau et al. [52] find out the vulnerabilities in the usage of loading external code with a static analysis approach, but they do not further analyze the security implications of the loaded binaries. Our code analysis is its superset, which includes five security-related aspects and allows us answer the 3 critical questions missing there. Our dynamic analysis framework allows us to precisely investigate the provenance, entity, and content of DCL. It additionally reveals that the loading of malicious code is triggered by properties of the runtime environment, such as system time and network connectivity. Falsina et al. [40] propose a code verification protocol and a drop-in library to reduce the vulnerabilities in DCL.

b) Program Analysis: RiskRanker [43] determines that DCL is taking place by static analysis. Its Dalvik code execution scheme is not able to analyze code loaded from sources other than local package, e.g. remote fetch. Zhang et al. [66] introduce a learning-based approach to analyzing the dependency of dynamic network requests. Crowdroid [35] is deployed in a crowdsourcing manner to detect Android malware using dynamic analysis. Because it applies low-level system call interposition, the analysis is not fine-grained due to the loss of context in Android middleware. Specially, it cannot differentiate the bytecode in the original application with that additionally loaded. TaintDroid [39] tracks the taint propagation at runtime, which aims at privacy leakage detection. Its implementation is based on DVM modification, which thus cannot handle native code. DroidBox [13], which combines TaintDroid and modifications of Android's code libraries, is able to log sensitive events at runtime, such as file read/write, loading class through `DexClassLoader`. It shares the same limitation with TaintDroid on analyzing native code. Some other dynamic analysis approaches can be adopted in our measurement [34], [63], [60], [58], which reconstructs both low-level OS-specific and high-level Android-specific behaviors. Those methods introduce heavy latency in behavior reconstruction. Our approach intercepts the dynamically loaded code, and passes it to the cheap static analysis.

c) Other Android Security Problems.: There are also some previous studies on other Android security problems, which are complementary to this paper. Rastogi et al.[57] demonstrate that obfuscation is widely used in Android malware for the purpose of bypassing detection. Cao et al. [36] propose and implement a systematic solution to the static analysis of Android framework and generating the Android API summaries. AppShield [53] enables the enforcement of arbitrary access control policies without the dependencies on OS, which resolves the data/privacy leakage issue in the enterprise scenario. AutoCog [54] leverages a learning-based approach to automatically verify whether the reason for in-app privacy usage is fully stated in the application description. SafePay [37], with backward compatibility, is proposed to address the issue of credit card information leakages via smartphone. Jin et al. [47], [46] design privacy-preserving solutions in the data collection for mobile sensing networks.

VII. Conclusion

The unpredictability of DCL challenges the related security and accountability analysis. In this paper, we build the system DyDroid, which is capable of intercepting DCL events and saving copies of the loaded bytecode and native code. We conduct a large-scale measurement of the DCL component of over 46K apps to investigate three critical questions missing in previous studies: (1) provenance, which includes the code's remote/local availability, and responsible entity; (2) app hardening, where DCL is used for the purpose of app obfuscation; (3) security risks/implications, which contains the malware detection, vulnerability analysis, and privacy tracking analysis. The apps that are found to use DCL in the remote fetch manner show that there is no existing solution to the enforcement of the related content policy. DCL is mainly used by third-party SDKs, indicated that the developer may not be aware that it is occurring. Given its stealthiness, DCL is also a channel to deploy malware, and we observe the real samples where the actual loading is controlled with logical conditions, such as system time. The security verification of DCL is needed from the app developer and OS vendors, given the apps vulnerable to code injection, which load binaries writable by other parties.

Acknowledgment

This paper was made possible by the U.S. National Science Foundation under Grant CNS-1408790 and the Defense Advanced Research Projects Agency (DARPA) under Grant FA8650-15-C-7561. We give our special thanks to Vaibhav Rastogi at the University of Wisconsin-Madison for his efforts on paper revising and the rebuttal period, Michael Grace at Samsung Research America for his support to this research project, and all the anonymous reviewers for their comments.

References

[1] 360 jiagu. http://jiagu.360.cn/.
[2] Ali jaq. http://jaq.alibaba.com/safety/.
[3] Allatori Java string renaming document. http://www.allatori.com/doc.html#properties-renaming.
[4] Android Application Class. https://developer.android.com/reference/android/app/Application.html.
[5] Android DexClassLoader. http://developer.android.com/reference/dalvik/system/DexClassLoader.html.
[6] Android PathClassLoader. http://developer.android.com/reference/dalvik/system/PathClassLoader.html.
[7] Android URI. http://developer.android.com/reference/android/net/Uri.html.
[8] Android's biggest problem is operating system fragmentation. http://o.canada.com/technology/personal-tech/androids-biggest-problem-is-operating-system-segmentation.
[9] Baidu. http://www.baidu.com/.
[10] Bangcle. http://www.bangcle.com/.
[11] Bitdefender Antivirus Software. http://www.bitdefender.com/.
[12] DBpedia - A Wikipedia based Natural Language Processing Database. http://dbpedia.org/datasets.
[13] Droidbox - android application sandbox. https://github.com/pjlantz/droidbox.

[14] DroidNative. https://bitbucket.org/shahid_alam/droidnative.
[15] Google Play. https://play.google.com/store?hl=en.
[16] Google Play Developer policy. https://play.google.com/about/developer-content-policy.html.
[17] Ijiami. http://www.ijiami.cn/.
[18] Java Native Interface. http://developer.android.com/ndk/samples/sample_hellojni.html.
[19] Java Object HashCode. https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode().
[20] Java StackTraceElement. https://docs.oracle.com/javase/7/docs/api/java/lang/StackTraceElement.html.
[21] Millions of Android smartphones are infected with malware - here's how to keep yours virus-free. http://www.mirror.co.uk/tech/millions-android-smartphones-infected-malware-8382366.
[22] Mobile Malware Dump. http://contagiominidump.blogspot.com.
[23] ProGuard. http://developer.android.com/tools/help/proguard.html.
[24] ptrace. http://man7.org/linux/man-pages/man2/ptrace.2.html.
[25] QQ. https://play.google.com/store/apps/details?id=com.tencent.mobileqq&hl=en.
[26] Smali: An assembler/disassembler for Android's dex format. http://code.google.com/p/smali/.
[27] Smartphone os market share. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.
[28] UI/Application exerciser Monkey. http://developer.android.com/tools/help/monkey.html.
[29] VirusTotal. https://www.virustotal.com/.
[30] WeChat. https://play.google.com/store/apps/details?id=com.tencent.mm&hl=en.
[31] S. Alam, R. N. Horspool, and I. Traore. Mail: Malware analysis intermediate language: a step towards automating and optimizing malware detection. In *SIN*, 2013.
[32] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi. Droidnative: Automating and optimizing detection of android native code malware variants. *computers & security*, 65:230–246, 2017.
[33] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269, 2014.
[34] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *IEEE MALWARE*, 2010.
[35] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *SPSM*, 2011.
[36] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.
[37] Y. Cao, X. Pan, and Y. Chen. Safepay: Protecting against credit card forgery with existing magnetic card readers. In *CNS*, pages 164–172. IEEE, 2015.
[38] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang. Profiling user-trigger dependence for android malware detection. *Computers & Security*, 49:255–273, 2015.
[39] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX OSDI*, 2010.
[40] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi. Grab'n run: Secure and practical dynamic code loading for android applications. In *ACSAC*, pages 201–210. ACM, 2015.
[41] A. P. Felt, S. Egelman, and D. Wagner. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *ACM SPSM*, 2012.
[42] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. *Trust and Trustworthy Computing*, 2012.
[43] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *MobiSys*, 2012.
[44] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec*, 2012.
[45] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *MobiSys*, 2014.
[46] H. Jin, L. Su, B. Ding, K. Nahrstedt, and N. Borisov. Enabling privacy-preserving incentives for mobile crowd sensing systems. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 344–353. IEEE, 2016.
[47] H. Jin, L. Su, H. Xiao, and K. Nahrstedt. Inception: Incentivizing privacy-preserving data aggregation for mobile crowd sensing systems. In *Proceedings of the 17th international symposium on mobile Ad Hoc networking and computing, MobiHoc*, volume 16, pages 341–350, 2016.
[48] H. Lockheimer. Android and security, February 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html.
[49] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013.
[50] S. Nath. Madscope: Characterizing mobile in-app targeted ads. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 59–73. ACM, 2015.
[51] J. Oberheide. Dissecting android's bouncer, June 2012. https://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/.
[52] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, 2014.
[53] Z. Qu, G. Guo, Z. Shao, V. Rastogi, Y. Chen, H. Chen, and W. Hong. Appshield: Enabling multi-entity access control cross platforms for mobile app management. In *International Conference on Security and Privacy in Communication Systems*, 2016.
[54] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1354–1365. ACM, 2014.
[55] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.
[56] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *CODASPY*, 2013.
[57] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *ASIACCS*, pages 329–334. ACM, 2013.
[58] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, and Y. Chen. Uranine: Real-time privacy leakage monitoring without system modification for android. In *International Conference on Security and Privacy in Communication Systems*, pages 256–276. Springer, 2015.
[59] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *NDSS*, 2016.
[60] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.
[61] K. Tian, D. Yao, B. G. Ryder, and G. Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 262–271. IEEE, 2016.
[62] R. Whitwam. Circumventing Google's Bouncer, Android's anti-malware system, June 2012. http://www.extremetech.com/computing/130424-circumventing-googles-bouncer-androids-anti-malware-system.
[63] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security Symposium*, 2012.
[64] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Research in Attacks, Intrusions, and Defenses*. 2015.
[65] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *CCS*, 2013.
[66] H. Zhang, D. D. Yao, and N. Ramakrishnan. Causality-based sensemaking of network traffic for android application security. In *AISec*, pages 47–58. ACM, 2016.
[67] Y. Zhang, X. Luo, and H. Yin. Dexhunter: toward extracting hidden code from packed android applications. In *Computer Security–ESORICS*. 2015.
[68] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In *CODASPY*, 2015.
[69] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. *IEEE Symposium on Security and Privacy*, 2012.
[70] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*. 2011.