Quokka

ACADEMIC PAPER

Analysis of Content Copyright Infringement

in Mobile Application Markets

Analysis of Content Copyright Infringement in Mobile Application Markets*

Ryan Johnson, Nikolaos Kiourtis, Angelos Stavrou
Kryptowire LLC &
Center for Assurance Research Engineering, George Mason University
{*rjohnso8, nkiourti, astavrou*}*@gmu.edu*

Vincent Sritapan
Department of Homeland Security
Science and Technology Directorate
Vincent.Sritapan@hq.dhs.gov

***Abstract*—As mobile devices increasingly become bigger in terms of display and reliable in delivering paid entertainment and video content, we also see a rise in the presence of mobile applications that attempt to profit by streaming pirated content to unsuspected end-users. These applications are both paid and free and in the case of free applications, the source of funding appears to be advertisements that are displayed while the content is streamed to the device.**

In this paper, we assess the extent of content copyright infringement for mobile markets that span multiple platforms (iOS, Android, and Windows Mobile) and cover both official and unofficial mobile markets located across the world. Using a set of search keywords that point to titles of paid streaming content, we discovered $8,592$ Android, $5,550$ iOS, and $3,910$ Windows mobile applications that matched our search criteria. Out of those applications, hundreds had links to either locally or remotely stored pirated content and were not developed, endorsed, or, in many cases, known to the owners of the copyrighted contents. We also revealed the network locations of $856,717$ Uniform Resource Locators (URLs) pointing to back-end servers and cyber-lockers used to communicate the pirated content to the mobile application.

I. Introduction

Over the past few years, we have experienced the advent of mobile devices capable of receiving and displaying streaming content. Everyday users are becoming more comfortable using their phones and tablets to view their favorite shows and movies. The convenience of the device, as well as the availability of the content, has made it much easier to experience video on a variety of devices. In fact, a recent residential pay-to-view study by J.D. Power and Associates indicates a clear shift in the way consumers watch streaming content. As depicted in Figure 1, phones and tablets each drew more than a 30 percent share in terms of viewing preference. Similar trends towards use of mobile devices for video content viewing are supported by Nielsen [1] and IDG [2].

However, as the mobile devices became the new vehicle for content dissemination, there has been a rise in the number of mobile applications that attempt to profit by illicitly delivering copyrighted content to the end-users. In many cases, these illicit mobile applications are taking advantage of the current market gap where providers of paid content have not developed or made available an application of their own to distribute their content through mobile devices. In other cases, they attract the users by offering the mobile application and streaming content for free in exchange for in-app advertisement and purchases both not easily monitored by the application market.

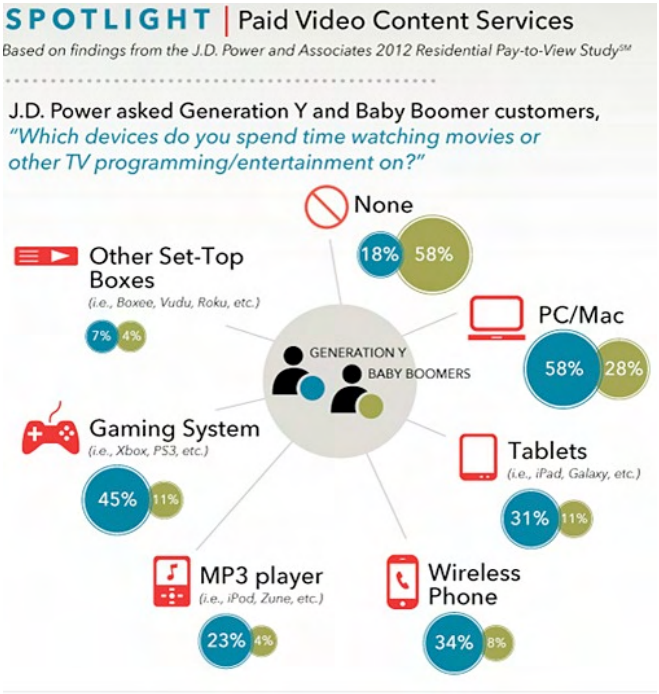

Fig. 1: J.D. Power and Associates survey for paid video content delivery mechanisms. The total number exceed 100 percent because users were allowed to select multiple options for where they view content.

In this paper, we are presenting a study of streaming content copyright infringement for mobile markets that span multiple platforms and both official and unofficial mobile markets located across the world. We were able to automatically and continuously collect application binaries and meta-data while surveying Google Android, Apple iOS, and Microsoft Windows mobile markets. Using a set of search keywords that point to titles of paid streaming content, we discovered $8,592$ Android, $5,550$ iOS, and $3,910$ Windows mobile applications that matched our search criteria. Out of those applications,

*This research was partly supported by the Science and Technology Directorate of Department of Homeland Security, USA. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not represent the views of the funding sponsor.

hundreds had links to either locally or remotely stored pirated content and were not developed, endorsed, or, in many cases, known to the owners of the copyrighted contents. We also revealed the network locations of $856,717$ Uniform Resource Locators (URLs) pointing to back-end servers and cyber-lockers used to communicate the pirated content to the mobile application.

Our work is the first of its kind to perform a large scale study across different platforms that covers more than one hundred mobile application markets for a period of a year: January to December of 2014. To perform this analysis, we collected and analyzed data for more than 4 million mobile applications. Our findings point out that there are many mobile applications that attempt to confuse users to pay for pirated content without revealing anything about the source or license of said content. Although large-scale in terms of platforms and market locations, the analysis in this paper is by no means exhaustive because the landscape of the pirated applications keeps changing. However, the presented results offer examples of an old form of electronic crime that is making its debut in a new set of market places.

To achieve this scale of collection and analysis, we built a framework that collects and processes mobile applications' binaries and meta-data including the location of back-end servers that stream pirated content. In our study, we had to address many challenges in terms of mobile application analysis and back-end server content analysis. Furthermore, an increasing number of mobile applications employ different forms of code obfuscation to thwart static and dynamic analysis in an effort to mask their operations. Code obfuscation is often coupled with the use of network and file encryption to prevent analysis that would reveal the extent of the available pirated content and the location of all the back-end streaming servers. We also encountered another form of obfuscation: the use of local, non-U.S. markets to deploy applications that are shielded from any attempts to access the mobile application or even content from U.S. network locations or devices. Finally, to validate our empirical findings, we manually inspected many of the mobile applications. In addition, we communicated all of our findings to the owners of the copyrighted content for take-down or other legal action.

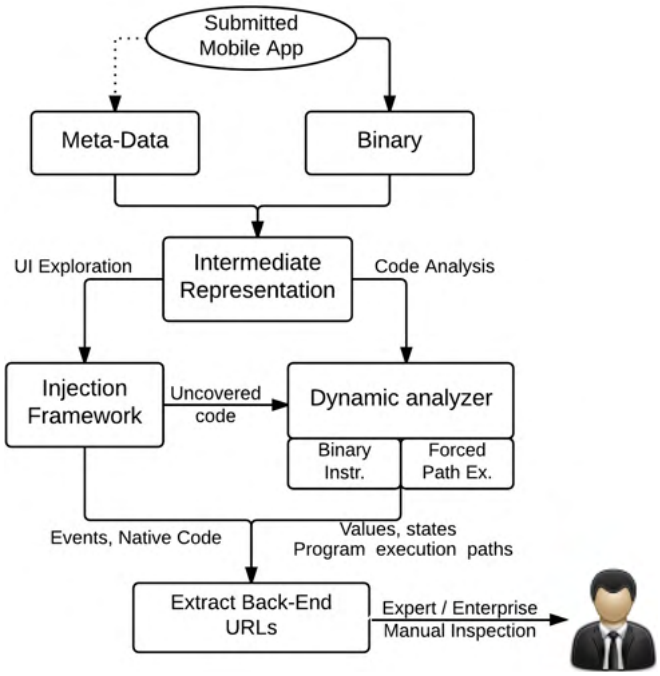

Fig. 2: Overall analysis workflow. We collect mobile packages and meta-data and we generate an intermediate representation that is passed to an injection framework that performs dynamic analysis and binary instrumentation eventually extracting the back-end URLs.

II. Analysis Framework Overview

Our mobile application analysis employs a wide-range of static and dynamic analysis techniques to enable the automated large-scale analysis of mobile applications and the extraction of back-end server information and streaming content to a database. This automated analysis process is also complemented with manual inspection to avoid any false positives (*i.e. mobile applications that were flagged as serving copyrighted material without permission*). Figure 2 provides an overview of the mobile analysis workflow.

In our application analysis, we must contend with a resource-asymmetric environment (lightweight devices, relatively thin communication with servers that can do more expensive computation, intermittent connectivity, etc.). Thus, we split the analysis into two parts: the injection framework that handles native portions that take on the device through the injection engine and the dynamic analysis portion that can run on x86 architectures by tagging and analyzing the code. For each platform, we have created a list of mobile Application Programming Interface (API) method calls that we target due to their functionality. We interpose on these targeted Android API calls using an injection framework to obtain, inspect, and possibly modify the parameter and return values. In addition, the specific instance of the object which has had its method called is also examined for non-static methods. The targeted platform and OS API calls can be hooked before and after one of the targeted API calls executes which allows us to modify the input and the output of the hooked mobile API method call. We can also provide our own implementation for the hooked API method call instead of allowing the actual implementation of the mobile API method call to execute.

A. Analysis Framework Implementation

Our mobile application analysis system only requires the mobile binary package or file of a mobile application along with the meta-data. After creating the intermediate representation, the data are passed to the dynamic analyzer and the injection framework (see Figure 2). The dynamic analyzer system provides analysis that will feed into the run-time program instrumentation: when the execution hits the target code, the system switches to controlled execution mode and

would continue untill it reaches a branch. For the encountered branch, the system follows all paths. The goal of the dynamic analyzer system is to traverse the effective control graph of the entire program, including third-party libraries. In addition, we use dynamic instrumentation to determine concrete values and program states that can serve as starting points for fault injection and vulnerability discovery in all code areas. As a complementary technique, we use concrete values from runtime execution as shortcuts in the dynamic execution analysis. The goal of both approaches is to permit a more complete exploration and automated analysis of the application code.

The main components of the framework are a single controller module and a plurality of execution modules. The controller module coordinates the operation of the execution modules and maintains shared data-structures (e.g., binary tree, results, etc.). The execution modules operate independently and concurrently take different execution paths through an application component. An execution module exists for a single unique execution path through an application component. Upon completion of a path, the controller module reinitializes the execution module so it can take an new path through the application component, assuming all paths have not been taken and that the time limit, if one exists, has not elapsed. The output from processing each application component is: (1) a method call graph, (2) control flow graph, (3) the output of an in-order traversal of the binary tree, (4) a list of the jump values taken for each execution through the application component, and (5) a list of relevant behaviors of the application component.

For Android specifically, we have published the general structure of the framework previously in [3], [4]. We briefly describe it here to provide some context on the addition we performed so we can scale our analysis. We use open-source software called apktool [5] to generate a human-readable representation of the Dalvik bytecode for an application. This format is called smali [6] and each smali file generally corresponds to a Java class. Nested Java classes will have their own smali file. The analysis framework reads, parses, and executes the instructions from the smali files. The framework contains a Java implementation for executing each of the 226 Dalvik instructions as they are represented in the smali files [7].

The Dalvik Virtual Machine (VM) uses registers to represent primitive data types and refer to objects. We created a custom Java data type to represent the registers in our framework, so they can be used in the Dalvik instructions. Each Dalvik instruction has its own implementation that utilizes our custom class implementing registers and the framework's internal data structures. A single line of Java code can translate into multiple lines of smali code since it is at the bytecode level. The registers in Dalvik bytecode are designated by a lowercase letter v or p followed by a decimal number. The translation of a Java statement for deleting all pictures from a rooted mobile device into equivalent lines of code in the smali format. For instance, a mobile application would only need the android.permission.WRITE_EXTERNAL_STORAGE permission to perform such an action.

B. Defending Against Obfuscation

In Android applications, the Java reflection API and dynamic class loading can be used to obfuscate the behavior of an application[8]. Dynamic class loading can enable the insertion and execution of mobile code to be performed at runtime[9]. At runtime, the application may download some bytecode, dynamically load the class, and call its methods reflectively. Static analysis may fail to identify any malicious activity that could be contained in the dynamically loaded class. A static analysis tool would most likely be able to identify the use of the dynamic class loading and reflection, but the target of a reflective call can be difficult or impossible to resolve using static analysis. It is possible to create a Java program that does not explicitly call any constructors directly. All objects could be created via the `java.lang.reflect.Constructor.newInstance(java.lang.Object[])` method call. In addition, all method calls could be performed using the `java.lang.reflect.Method.invoke(java.lang.Object, java.lang.Object[])` API call. This would certainly make any manual analysis of the source code or bytecode much more time consuming.

The injection framework will intercept API method calls from native code, Java reflection, and dynamic class loading. These are some methods that may be used to try to obfuscate the functionality from static analysis and dynamic analysis via bytecode rewriting. These situations can generally be handled by bytecode rewriting, but it would likely introduce more complexity. Reflective method calls can be chained in order to try to obfuscate the ultimate target of the reflective method call. This is accomplished by calling `java.lang.reflect.Method.invoke(java.lang.Object, java.lang.Object[])` resulting in a reflective call to `java.lang.reflect.Method.invoke(java.lang.Object, java.lang.Object[])` that reflectively calls `java.lang.reflect.Method.invoke(java.lang.Object, java.lang.Object[])` which has the ultimate target of `java.lang.Runtime.exec(java.lang.String[])`. The injection framework will ultimately intercept the call to `java.lang.Runtime.exec(java.lang.String[])` after the reflective calls are executed.

Dynamic class loading can be used to load dex bytecode from a jar, apk, or dex file. The injection framework will intercept API calls from dynamically loaded code. When using bytecode rewriting, any dex files that are loaded at runtime must be rewritten before execution of the application continues so that the logging code is inserted. This can be accomplished by intercepting certain constructor calls for dynamic class loading (e.g., dalvik.system.DexClassLoader), so that the dex bytecode can be instrumented prior to being loaded. This can be done at runtime, but there is generally a significant performance penalty from doing so. This only needs to be done once for each file containing dex bytecode if the optimized dex file is maintained from the previous time it was loaded.

Native code is an appropriate location to obfuscate API usage since it is not as easily reverse engineered as Dalvik bytecode. If the API usage is in native code and it is not obfuscated, then some API calls may still be able to infer statically by examining the rodata (i.e., read-only data) segment when examining the output of using objdump on a native library written in C. This objdump executable comes with the Android Native Development Kit (NDK)[10] and the otool[11] or LLVM toolchain[12] for iOS. In Figure 3 we present source

code written in C to use an android.content.Context object passed in as a parameter to query the text messages on a mobile device. The android.database.Cursor object will be returned to the calling method where it can be used to obtain the text messages from the device.

```
JNIEXPORT jobject JNICALL
Java_com_kryptowire_randomapp_SMS_getSMSMessages(JN
IEnv * env, jobject  obj, jobject context)
{
    jclass uri_cls = (*env)->FindClass(env,
"android/net/Uri");
    jmethodID methodId = (*env)-
>GetStaticMethodID(env, uri_cls, "parse", "(Ljava/
lang/String;)Landroid/net/Uri;");
    const char sms_uri[] = "content://sms";
    jstring contentBuf = (*env)->NewStringUTF(env,
sms_uri);
    jobject uri = (*env)-
>CallStaticObjectMethod(env, uri_cls, methodId,
contentBuf);
    jclass cls_context = (*env)->FindClass(env,
"android/content/Context");
    jmethodID contentResolverMid = (*env)-
>GetMethodID(env, cls_context,
"getContentResolver", "()Landroid/content/
ContentResolver;");
    jobject cr = (*env)->CallObjectMethod(env,
context, contentResolverMid);
    jclass cls_cr = (*env)->FindClass(env,
"android/content/ContentResolver");
    jmethodID queryMid = (*env)->GetMethodID(env,
cls_cr, "query", "(Landroid/net/Uri;[Ljava/lang/
String;Ljava/lang/String;[Ljava/lang/String;Ljava/
lang/String;)Landroid/database/Cursor;");
    jobject cursor = (*env)->CallObjectMethod(env,
cr, queryMid, uri, NULL, NULL, NULL, NULL);
    return cursor;
}
```

Fig. 3: De-obfuscating native code using JNIEXPORT and JNICALL methods and examining the read-only data segment of the objectdump command (see Figure 4).

Below is a partial output of running objdump -s thelibrary.so. We have only included a small sample of the rodata section. The strings that are used as parameters to functions from native code can consequentially be seen as they show up in the C source code which is shown above. Examining the output, one can see the `android/net/Uri.parse.(Ljava/lang/String;)Landroid/net/Uri;`, which is the fully qualified API call. The period in the previous API call represents a value of 0x00, that is how objdump represents non-printable characters. The platform API usage in native code could be obfuscated by deriving some of the strings instead of hard-coding them in the code.

C. Exercising the User Interface

Our framework is completely automated. We utilize a shell script and the Mobile Debugging Bridge (ADB) to perform the necessary file transfer and command execution on the device. Our framework does require the device to be rooted, so that we can access files on the internal storage of the device. For complete automation, we use Monkey [13] to exercise the (User Interface) UI to get a sampling of an application's functionality. The option for complete automation

```
Contents of section .rodata:
 3884 4e617469 76654c6f 67730044 69642069  NativeLogs.Did i
 ..........................................................
 3d24 72657475 726e476d 61696c00 616e6472  returnGmail.andr
 3d34 6f69642f 6e65742f 55726900 70617273  oid/net/Uri.pars
 3d44 6500284c 6a617661 2f6c616e 672f5374  e.(Ljava/lang/St
 3d54 72696e67 3b294c61 6e64726f 69642f6e  ring;)Landroid/n
 3d64 65742f55 72693b00 616e6472 6f69642f  et/Uri;.android/
 3d74 636f6e74 656e742f 436f6e74 65787400  content/Context.
 3d84 67657443 6f6e7465 6e745265 736f6c76  getContentResolv
 3d94 65720028 294c616e 64726f69 642f636f  er.()Landroid/co
 3da4 6e74656e 742f436f 6e74656e 74526573  ntent/ContentRes
 3db4 6f6c7665 723b0061 6e64726f 69642f63  olver;.android/c
 3dc4 6f6e7465 6e742f43 6f6e7465 6e745265  ontent/ContentRe
 3dd4 736f6c76 65720071 75657279 00284c61  solver.query.(La
 3de4 6e64726f 69642f6e 65742f55 72693b5b  ndroid/net/Uri;[
 3df4 4c6a6176 612f6c61 6e672f53 7472696e  Ljava/lang/Strin
 3e04 673b4c6a 6176612f 6c616e67 2f537472  g;Ljava/lang/Str
 3e14 696e673b 5b4c6a61 76612f6c 616e672f  ing;[Ljava/lang/
 3e24 53747269 6e673b4c 6a617661 2f6c616e  String;Ljava/lan
 3e34 672f5374 72696e67 3b294c61 6e64726f  g/String;)Landro
 3e44 69642f64 61746162 6173652f 43757273  id/database/Curs
 3e54 6f723b00 636f6e74 656e743a 2f2f736d  or;.content://sm
 3e64 73005445 4c455048 4f4e595f 53455256  s.TELEPHONY_SERV
```

Fig. 4: Read-Only data segment (.rodata) output from using the objdump command in Android. Similar information can be obtained by the use of the otool or LLVM toolchain for iOS.

allows the analysis to scale. Monkey is not a complete solution since it randomly injects UI events on the device. There is also the option for allowing for a human user to control the analysis process and exercise the application as long as they desire. This generally enables a more complete view into the application's functionality due to an intelligent agent exercising the application.

D. Exclusion List

We have two modes of targeting specific applications. Generally, injection frameworks require the device to be rebooted whenever a change to the code to interact with the injection framework is made. We use an exclusion list of applications to ensure that the hooked API method calls in certain applications are exempt from the logging process. It is possible to isolate a specific package name by writing it to a file to be checked by the injection framework or hard-coding it, but this requires that the device be rebooted whenever the package name changes or that a certain amount of time passes. To create the exclusion list, we first create a baseline of package names that are installed on the device. These are the required applications and OS processes that should generally be excluded unless otherwise desired. This should be a static and complete list so that any installed application (*i.e., the target application*) has its API method calls logged since it is not part of the exclusion list. After analysis of the targeted application has completed, the application is removed from the device. Then the next application can be installed and analyzed without rebooting the device. The exclusion list is implemented as a hash set of strings that represent the package name of applications and processes to be excluded. This only needs to be checked once by each process and then a static Boolean variable is set to true or false indicating whether that application is excluded from logging or not. Subsequently, to check if the application is exempt from logging, it can quickly just check a Boolean variable to determine it.

E. Tracking Data

We have two different methods for capturing the network and file I/O performed by a mobile application. The first is by hooking API method calls for I/O streams before or after they have been executed and capturing what is being written or read. This can have performance penalties, especially if the I/O is not buffered. The analysis framework incurs overhead for writing a log entry. When a log entry is created for reading or writing a single byte from an I/O stream, the performance starts to degrade. The analysis framework is considerably more efficient, when reading and writing streams are buffered. To minimize the latency that is introduced when analyzing I/O-heavy applications, we also have a mode to offload the logging of I/O-related API calls from the injection framework. For this approach, we simply do not hook the API calls to read from the I/O streams. We then utilize a proxy to perform a Man In The Middle (MITM) attack on the connection, so that we are able to intercept the data sent over the network and even from SSL/TLS connections. This offloads the network I/O onto another program that captures the data. Some applications use certificate pinning to try to prevent a MITM attack on a connection. We make an attempt to obviate this by hooking various Java methods (*e.g.,* `javax.net.ssl.TrustManagerFactory`) to allow any certificate to be accepted for the connection instead of the one that is pinned.

We offload file I/O streams by post-processing the log file after the analysis has concluded. We have a list of constructors and method calls from the API that are used for reading and writing files. The parameters to certain constructors (*e.g.* `java.io.FileInputStream.(java.io.File)`) and method (*e.g.* `android.content.ContextWrapper.openFileOutput(java.lang.String,int)`) are examined to obtain the full path to files that are read or written by the application. The list of accessed files is generated as the log is parsed during post-processing. Thereafter, the files accessed by the application are pulled off the device. We first copy the file to the SD card and then pull it from the device using ADB. We also have a mode to be more aggressive about preserving files by making files more persistent that they would otherwise be. We can hook calls such as java.io.File.delete() and provide our own implementation that runs prior to the actual call and simply returns true and does not delete the file. The same can be done for temporary files by creating an actual file, so it will not be automatically deleted. This can help to preserve files so that they are copied for later analysis. In addition, the private directory on internal storage for the application (*e.g., /data/data/com.facebook.katana*) is also pulled from the phone since this is where an application tends to have the majority of its files. As the analysis is occurring, the APK is unzipped so that files from the assets folder can be accessed. An exclusion list of files is checked before a file or directory is pulled from the phone. For example, we prevent the entire /mnt/sdcard/DCIM/Camera directory from being pulled even though it is accessed to have its contents listed. The same goes for the /mnt/sdcard directory as well as others.

F. Logging Methods

There are multiple methods to transfer the log entries from within the application to long-term storage. The possible logging methods are writing to the following locations: logcat, a named pipe, a file, and a memory-mapped file. The most efficient method is writing to a memory-mapped file. The events are written to the file in a sequential order due to the use of a lock to ensure mutual exclusion when writing to the file. The use of threads complicates the sequencing of the API calls. Nonetheless, a relative ordering of API calls can be examined to give more context to what multiple API calls may be doing in aggregate. For example, one might observe the application querying the SMS messages, writing them to a file, encrypting the file, and sending it over the network. We also record call the `java.lang.System.identityHashCode(java.lang.Object)` on all objects to identify common objects across various API calls.

Writing the output to logcat is convenient since you do not have to setup a file or pipe to write to, but writing to the mobile log is slow and also results in a log file that contains log entries from all other processes that must be removed. The size of an entry in logcat has a maximum size of 4KB, so large entries must be paginated. Writing to logcat also incurs the overhead of writing binary data to a different representation such as hexadecimal. In other methods, the binary data from byte arrays can be written in binary and converted to a String representation when the log file is post-processed. Writing to a memory-mapped file is the most effective for writing large amounts of data to a file. There is a trade-off between the size of the memory-mapped file and performance.

G. Finding the targeted API call

The smali format [6] is a representation for the disassembled dex bytecode format that the Dalvik Virtual Machine (VM) uses. In some cases, we are able to identify the exact line of code in the smali file in which the targeted API method call occurred. We obtain the smali files using baksmali, which is used for reverse engineering an applications classes.dex file. Each smali file generally corresponds to a Java file, although inner classes will have their own smali file. Each smali file also contains the package to which the class belongs, and the location of smali files is hierarchical according to the package name.

The smali format has .line directive that is used to populate the line numbers in stacktraces when an exception is encountered. Below is a snippet of code from the com/mvl/core/MapViewActivity.smali class showing use of the .line directive.

Listing 1: "Smali code from MapViewActivity.smali file

```
.line 343
const/4 v5, 0x1

invoke-virtual {v3, v12, v5}, Landroid/
    location/LocationManager;->
    getBestProvider(Landroid/location/
    Criteria;Z)Ljava/lang/String;

move-result-object v4

.line 344
.local v4, "provider":Ljava/lang/String;
if-eqz v4, :cond_a8

.line 345
```

```
    invoke-virtual {v3, v4}, Landroid/location/
        LocationManager;->getLastKnownLocation(
        Ljava/lang/String;)Landroid/location/
        Location;

    move-result-object v16

    .line 346
    .local v16, "l":Landroid/location/Location;
    if-eqz v16, :cond_9e

    .line 347
    move-object/from16 v0, p0

    move-object/from16 v1, v16

    invoke-virtual {v0, v1}, Lcom/mvl/core/
        MapViewActivity;->onLocationChanged(
        Landroid/location/Location;)V
```

Below is a callstack called from within n mobile application. Once a targeted API call is hooked, then the injected code runs and the call stack can be examined. Usually, the location of the hooked API method call is in a predictable location. From the instance of the call (*i.e.* `android.accounts.AccountManager.getAccountsByType`), the callstack can be searched downward for the first concrete smali file. The callstack provides the fully qualified method call which enables us to locate the corresponding smali file. From there, the file is opened and the possible matches of methods are searched for the corresponding line number from the line directive. Then searching commences in all methods within the matching that have a matching line directive for the targeted Mobile API call. When the best match is found, the actual line number of the smali file can be determined.

```
android.accounts.AccountManager.getAccountsByType
    (Native Method)
com.facebook.katana.platform.
    FacebookAuthenticationUtils.a(
    FacebookAuthenticationUtils.java:128)
com.facebook.katana.authlogin.
    AccountManagerAuthComponent.d(
    AccountManagerAuthComponent.java:47)
com.facebook.katana.server.handler.
    Fb4aAuthHandler.a(Fb4aAuthHandler.java:452)
com.facebook.katana.server.handler.
    Fb4aAuthHandler.a(Fb4aAuthHandler.java:285)
com.facebook.katana.server.handler.
    Fb4aAuthHandler.a(Fb4aAuthHandler.java:223)
com.facebook.fbservice.service.BlueServiceQueue.e
    (BlueServiceQueue.java:360)
com.facebook.fbservice.service.BlueServiceQueue.d
    (BlueServiceQueue.java:58)
com.facebook.fbservice.service.BlueServiceQueue$3
    .run(BlueServiceQueue.java:280)
java.util.concurrent.Executors$RunnableAdapter.
    call(Executors.java:422)
java.util.concurrent.FutureTask.run(FutureTask.
    java:237)
com.facebook.common.executors.
    ListenableScheduledFutureImpl.run(
    ListenableScheduledFutureImpl.java:58)
android.os.Handler.handleCallback(Handler.java
    :733)
android.os.Handler.dispatchMessage(Handler.java
    :95)
android.os.Looper.loop(Looper.java:136)
android.os.HandlerThread.run(HandlerThread.java
    :61)
```

The example stacktrace, shown above, is from the Facebook mobile application which has a package name of com.facebook.katana. The method being hooked is the `android.accounts.AccountManager.getAccountsByType (java.lang.String)` method. This method is called from the line 128 of the a method of the `com.facebook.katana.platform.FacebookAuthenticationUtils` class. The line 128 corresponds to the line number in the Java source code. The method calls do not have parameter values, so one must be aware of that and try to make the best match possible in the smali file.

H. Limitations of the Analysis Process

One needs to be careful when using an injection framework. It is easy to create an infinite loop of hooking a call from the application and then using a call that is to be hooked in your logging code. This can create a boot loop or a `java.lang.StackOverflowError` exception. The code used in conjunction for the injection framework needs to be thoughtfully constructed and tested.

The application being analyzed may be able to detect that is being executed under supervision or that an injection framework is active on the device. Some ways it could try to tell is by checking for all installed applications on the device and looking for known package names that belong to injection frameworks. It may be possible to examine the callstack of an executing thread and look for the presence of certain injection methods. There should be various ways to detect them.

The injection framework only allows direct methods to be hooked. This means that the method to be hooked has to reside in the class specifically being hooked and not a virtual method where the actual implementation resides in a superclass. In addition, interfaces cannot be hooked since it needs to hook the concrete implementation that fulfills the interface.

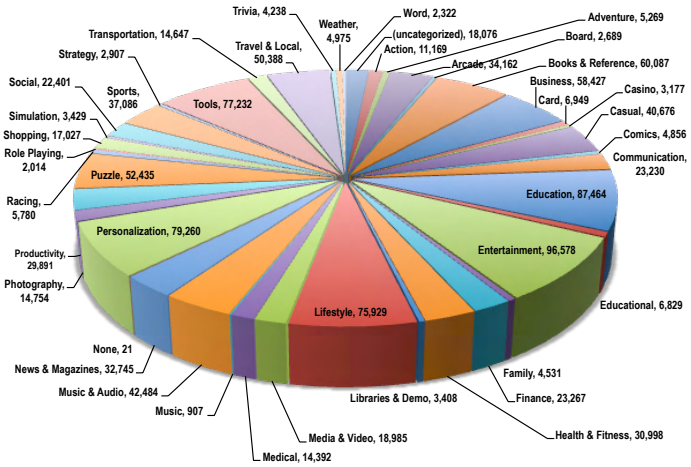

Fig. 5: A chart of mobile application in Android Google Play market for each of the categories that are present in the market.

III. Analysis Results

In this section, we discuss our empirical results from using our analysis framework to expose mobile applications that

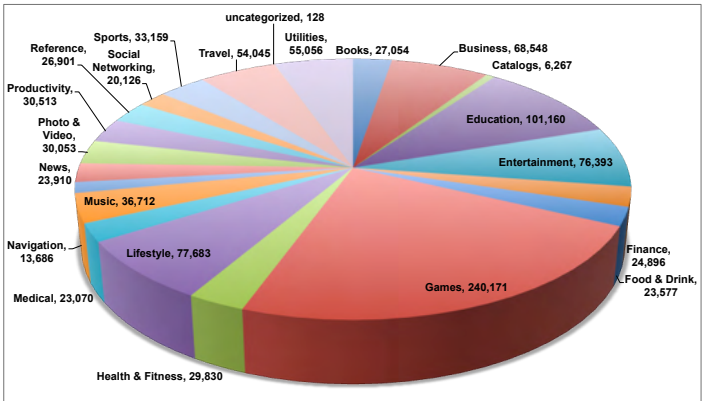

Fig. 6: Apple iTunes store mobile applications for each of the categories that are present in the market.

Total Apps Discovered	Google Play	Apple App Store	Windows Phone
U.S. & Foreign Stores	**8,592**	**5,550**	**3,910**
Brazil only	91	48	7
China only (Chinese Simplified)	41	61	20
China only (Chinese Traditional)	74	45	12
India only	49	0	0
Italy only	83	62	23
Russia only	125	63	13
South Korea only	32	37	2
Total Apps avalable in Foreign Stores only (All Countries except U.S.)	**495**	**316**	**77**

Fig. 7: Infringing Mobile Apps across all stores and marketplaces.

were streaming copyrighted content without the permission of the rightful owner and with the intent to make profit either by selling the mobile application itself or by displaying advertisements to the end-user while the content on the mobile device when the application was in use.

Overall, we surveyed more than one hundred official and unofficial markets including Apple's iTunes, Google Play, Windows Mobile, and Amazon Appstore, a list with a representative set for the 3rd-party markets we surveyed is included in the Appendix A.

We were also able to collect data from all the official market places. Figure 5 shows a list of mobile applications present as of December 2014 in the Android app store where as Figure 6 represents the list of iOS applications in the Apple iTunes store.

Our main result is shown in Figure 7. In this table, we discovered $8,592$ Android, $5,550$ iOS, and $3,910$ Windows mobile applications that matched our search criteria. Out of those applications, hundreds had links to either locally or remotely stored pirated content and were not developed, endorsed, or, in many cases, known to the owners of the copyrighted contents. Notice, that depending on where the market is being accessed in terms network location, the local markets produced different results in our search for mobile applications. For instance, in Google Play Russia, we were able to discover 125 more applications that were serving infringing content to end-users.

The top pirated movies sorted by the number of mobile applications that were streaming their content along with their copyright owner and the total number of applications and URLs that were carrying its content is depicted in Figure 8. Notice that many of the movies are older than a year and that there are many back-end URLs that support their streaming. This is due to the fact that mobile applications attempt to load-balance their streaming content and identify the closest server optimizing the latency and user load. In many cases, some of the back-end URLs serve as a backup in case one or more of them become unreachable or taken-down because of copyright infringement.

Title	Year	Copyright Material Owner	URLs	Apps
Iron Man 3	2013	Walt Disney Studios Motion Pictures	588	84
Despicable Me 2	2013	Universal City Studios LLC	407	78
Escape Plan	2013	Summit Entertainment	559	75
White House Down	2013	Sony Pictures Entertainment Inc.	483	75
Turbo	2013	Twentieth Century Fox Film Corporation	430	75
World War Z	2013	Paramount Pictures Corporation	447	73
Life of Pi	2012	Twentieth Century Fox Film Corporation	320	73
Warm Bodies	2013	Summit Entertainment	312	73
Gravity	2013	Warner Bros. Entertainment Inc.	442	72
Red 2	2013	Summit Entertainment	385	72
RoboCop	2014	Sony Pictures Entertainment Inc.	371	72
Parker	2013	SND	398	71
Elysium	2013	Sony Pictures Entertainment Inc.	428	70
The Wolverine	2013	Twentieth Century Fox Film Corporation	585	69
Captain Phillips	2013	Sony Pictures Entertainment Inc.	522	69
The Host	2013	Chockstone Pictures	503	69
The Hunger Games	2012	Lions Gate Entertainment	461	69
The Conjuring	2013	Warner Bros. Entertainment Inc.	400	69
Frozen	2013	Walt Disney Studios Motion Pictures	394	69
The Croods	2013	DreamWorks Animation	262	69
Ride Along	2014	Universal City Studios LLC	492	67
Epic	2013	Twentieth Century Fox Film Corporation	473	67
Olympus Has Fallen	2013	Nu image Films & Millenium Films	339	67
2 Guns	2013	Universal City Studios LLC	331	67
Grown Ups 2	2013	Sony Pictures Entertainment Inc.	472	66
The Lone Ranger	2013	Walt Disney Studios Motion Pictures	402	66
Last Vegas	2013	Universal City Studios LLC	298	65
The Expendables 2	2012	Nu image Films & Millenium Films	284	65
Carrie	2013	Sony Pictures Entertainment Inc.	493	64
Pacific Rim	2013	Warner Bros. Entertainment Inc.	462	64

Fig. 8: Top 30 pirated content ranked by the number of mobile applications that were streaming the content.

Another interesting measurement is the geo-location of the most popular back-end server that provide pirated streaming content to the mobile applications throughout the globe. In Figure 9, we have plotted the locations of the most notable ones on a map. A list with the top 40 domains and servers and their country of origin is shown in Figure 10.

IV. Related Work

The forced path execution techniques for Android applications have been discussed in our previous work [3], [4]. However, neither the workflow nor the obfuscation avoidance techniques have been presented for Android. The workflow is completely new and it covers both iOS and Windows Mobile platforms, which was not the case in any of our previous work.

Forced path execution for binaries have been extensively discussed in [14], [15], [16] and some of the obfuscation techniques for Android malware have appeared in [17], [18],

Fig. 9: Map with the geo-location of the back-end streaming servers for pirated content.

Host Name	urls	country_name	continent_code
www.youwatch.org	695768	Switzerland	EU
www.enlacespepito.com	484167	Spain	EU
vk.com	290318	Russian Federation	EU
www.exashare.com	227648	Netherlands	EU
31.7.60.50	122186	Switzerland	EU
www.youtube.com	108180	United States	NA
www.putlocker.com	85179	United States	NA
www.sockshare.com	66263	Costa Rica	NA
api.video.mail.ru	65483	Russian Federation	EU
filenuke.com	65208	United States	NA
daclips.com	59940	Moldova, Republic of	EU
movpod.net	59554	Moldova, Republic of	EU
plist.vn-hd.com	55554	Vietnam	AS
allmyvideos.net	49784	Netherlands	EU
gorillavid.in	49775	Moldova, Republic of	EU
gorillavid.com	48001	Moldova, Republic of	EU
redirector.googlevideo.com	35905	United States	NA
adf.ly	34751	United States	NA
sharesix.com	34051	United States	NA
played.to	29098	United States	NA
daclips.in	28019	Moldova, Republic of	EU
movpod.in	27457	Moldova, Republic of	EU
tv.zing.vn	27231	Vietnam	AS
www.movshare.net	26630	Switzerland	EU
www.vidhog.com	23037	United Kingdom	EU
www.nowvideo.eu	23004	Netherlands	EU
www.videoweed.es	21820	Netherlands	EU
thefile.me	21483	United States	NA
www.novamov.com	21326	Netherlands	EU
www.uploadc.com	20457	Germany	EU
www.zalaa.com	19836	Germany	EU
sharerepo.com	19644	Netherlands	EU
mightyupload.com	19449	United States	NA
www.divxstage.eu	19009	Netherlands	EU
www.firedrive.com	17824	Costa Rica	NA
stagevu.com	17565	Netherlands	EU
nosvideo.com	17463	Germany	EU
www.tudou.com	16784	China	AS
bestreams.net	14990	Ukraine	EU
videomega.tv	13498	United States	NA

Fig. 10: Top 40 domains that serve pirated streaming content to mobile applications and their country of origin.

[19]. However, we are not aware of any large-scale study that attempted to analyze mobile applications in terms of copyright infringement.

Another related form of execution, called concolic testing [20], [21], [22] was used to facilitate software and unit testing in cases where code coverage was important and symbolic execution was not possible. Our work is inspired by those techniques but we do not apply them directly because we have to analyze code that is mixed: Java bytecode and native code for Android, objective c and HTML for iOS and Visual C++ and Visual C# for Windows Mobile.

V. Conclusion

We have presented a study of infringing content for mobile applications using an automated framework for analysis of mobile applications. The process of collecting and analyzing mobile applications has many challenges including the attempts to obfuscate the application code and encrypt the network traffic. We present our approach that employs dynamic analysis and code instrumentation to bypass obfuscation techniques and allow the analysis of apps at scale.

Our empirical results show that the copyright infringement of mobile apps is non-negligible with cyber-lockers that span many domains and countries around the globe. In addition, there is clear evidence that this is not a platform-specific problem but rather a systemic issue that requires serious consideration as the number of mobile users that consume copyrighted content increases. We believe that our study is one of the first to shed light on the extent of the problem and offer a global view across all major mobile platforms.

References

[1] Nielsen, "Mobile Consumer Report," 2013.

[2] IDG Global, "Mobile Survey."

[3] R. Johnson, Z. Wang, A. Stavrou, and J. Voas, "Exposing software security and availability risks for commercial mobile devices," in *Reliability and Maintainability Symposium (RAMS), 2013 Proceedings-Annual*. IEEE, 2013, pp. 1–7.

[4] R. Johnson and A. Stavrou, "Forced-path execution for android applications on x86 platforms," in *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*. IEEE, 2013, pp. 188–197.

[5] Google, "Android apktool: A tool for reengineering Android apk files," http://code.google.com/p/android-apktool/.

[6] ——, "Smali - an assembler/disassembler for Android's dex format." http://code.google.com/p/smali/.

[7] ——, "Dalvik bytecode," https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[8] M. Hirzel, A. Diwan, and M. Hind, "Pointer analysis in the presence of dynamic class loading," in *In ECOOP*, 2004, pp. 96–122.

[9] H. Kaiya and K. Kaijiri, "Specifying runtime environments and functionalities of downloadable components under the sandbox model," in *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, 2000, pp. 138–142.

[10] Google, "Android Native Development Kit (NDK)," https://developer.android.com/tools/sdk/ndk/index.html.

[11] "Otool-NG bibtex — endnote — acm ref @inproceedingsLattner:2004:LCF:977395.977673, author = Lattner, Chris and Adve, Vikram, title = LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, booktitle = Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, series = CGO '04, year = 2004, isbn = 0-7695-2102-9, location = Palo Alto, California, pages = 75–, url = http://dl.acm.org/citation.cfm?id=977395.977673, acmid = 977673, publisher = IEEE Computer Society, address = Washington, DC, USA, ," https://github.com/gdbinit/otool-ng.

[12] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[13] Android developers. ui application exerciser monkey. [Online]. Available: http://developer.android.com/guide/developing/tools/monkey.html

[14] Y. Nadji, M. Antonakakis, R. Perdisci, and W. Lee, "Understanding the prevalence and use of alternative plans in malware with network games," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 1–10.

[15] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 2014 USENIX Security Symposium, San Diego, CA (August 2014)*, 2014.

[16] M. Graziano, C. Leita, and D. Balzarotti, "Towards network containment in malware analysis systems," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 339–348.

[17] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 329–334.

[18] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting android reuse studies in the context of code obfuscation and library usages," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 242–251.

[19] M. Protsenko and T. Muller, "Pandora applies non-deterministic obfuscation randomly to android," in *Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on*. IEEE, 2013, pp. 59–67.

[20] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 416–426.

[21] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*. Springer, 2006, pp. 419–423.

[22] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for c/c++ using concolic execution," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 132–141.

APPENDIX
TRACKED THIRD-PARTY MARKETS

Tracked Third Party Markets:

1Mobile
91mobiles
92Apk
AlternativeTo
Amazon
Andapponline
Android Downloadz
Android Freeware
Android games room
Androidblip
AndroidPit
AndroidTapp
AndroLib
Anzhi Market
ApkSuite
AppBrain
AppCake
AppChina
AppCity
Appitalism
Appolicious
AppsEgg
AppsFire
AppsLib
AppsZoom
AppTown
Appzil
Aptoide
AT&T
Baidu App Store
Barnes & Noble
Blackmart Alpha
Brophone
Brothersoft
Camangi
Cisco Market
CNET
CoolApk
Cydia
D.cn Games Center
EOE Market
ESDN
F-Droid
Fetch
GetJar
YAAM
Gfan
Good Ereader
Hami
Handango
Handmark
Handster
HiApk
Hyper Market
iMedicalApps
Insyde Market
Lenovo App Store
LG World
MerkaMarket
Mikandi
Millet App Store
Mob.org
Mobango
Mobile9
mobiles24
Mobilism
Moborobo
MplayIt
N-Duo
Naver NStore
AppAddict
DTAThemes
Insanelyi
BiteYourApple
Sinful iPhone
iF0rce
Ryan Petrich
FilippoBiga
Pushfix
HackYouriPhone
HASHBANG Productions
XSellize
iSpazio
ZodTTD
ModMyi
BigBoss
Popcorn Time
iPhoneCake
Saurik Cydia Repo
Karen Pineapple Repo
XBMC iOS Repo
Wei Feng Public Source
25PP
3K Assistant