API Tracing Tool for Android-based Mobile Devices

Seonho Choi, Michael Bijou, Kun Sun, and Edward Jung

***Abstract*— We developed an Application Programming Interface (API) tracing tool for Android-based mobile devices. This tool has the capability of generating trace files by remotely monitoring and recording API calls made by an app running on a mobile device. In addition, the automatic feature of the tool allows repeated tracing tasks to be conducted without user's intervention against a collection of apps. The trace data may be effectively utilized in analyzing app's behavior and intentions, which will prove to be useful in several application domains. For example, this tool may be adopted in developing malware detection techniques based upon the API call patterns, identifying resource usages in different app components, building the tool for visualizing the app's behavior, and constructing efficient mobile forensics tools, etc. Also, different components of this tool make use of various concepts covered in core computer science courses such as Operating Systems, Computer Networks, Compilers, Data Structures, Algorithms, and Security and Forensics, which makes them to be suitable candidate for a senior capstone project. We formulated inter-related project modules based upon the components of this tracing tool, and will adopt them into the senior capstone course at Bowie State University.**

***Index Terms*—Android, API tracing, mobile security, senior capstone project, mobile forensics**

I. Introduction

Android has become a popular mobile operating system for various devices including smartphones. Android is the most widely used mobile operating system, with 81% of smartphones and 37% of tablets worldwide running this Google-made OS. To analyze Android apps behavior and intentions, we may trace the API calls made by an Android app at run time. This trace data may be utilized for a wide range of problems including malware detection technique development and mobile forensic schemes.

Even though Android is built based on Linux, the API tracing techniques developed for desktops may not be applied. Android has a different system architecture. At a lower level each application is encapsulated into a separate process and it is run by using the services provided by Linux kernel. Within each application, a virtual machine, known as a Dalvik Virtual Machine (DVM), provides a run-time environment for the Java components included in the application (app). All apps contain both Java and native components. Native components are simply shared libraries that are dynamically loaded at runtime. In the Dalvik virtual machine (DVM), a shared library named libdvm.so is then used to provide a Java-level abstraction for the app's Java components. Application developers heavily make use of the objects and methods provided by the Java Library included in the DVM. To understand or grasp the intentions of the apps, it will be more beneficial to trace/utilize the Java-level semantics that comprehend the behaviors of the Java components in the app rather than focusing on system call histories captured at the lower level.

We developed an Application Programming Interface (API) tracing tool for Android-based mobile devices. This tool has the capability of generating trace files by remotely monitoring and recording API calls made by an app running on a mobile device. In addition, the automatic feature of the tool allows repeated tracing tasks to be conducted without user's intervention against a collection of apps. The trace data may be effectively utilized in analyzing app's behavior and intentions, which will prove to be useful in several application domains. For example, this tool may be adopted in developing malware detection techniques based upon the API call patterns, identifying resource usages in different app components, building the tool for visualizing the app's behavior, and constructing efficient mobile forensics tools, etc. Also, different components of this tool make use of various concepts covered in core computer science courses such as Operating Systems, Computer Networks, Compilers, Data Structures, Algorithms, and Security and Forensics, which makes them to be suitable candidate for a senior capstone project. We formulated inter-related project modules based upon the components of this tracing tool, and will adopt them into the senior capstone course at Bowie State University.

The remainder of the paper is structured as follows. We present the related works in Section 2. Section 3 presents a methodology. In Section 4 experiments are explained. Section 5 concludes the paper.

II. Related Works

In signature-based malware detection systems malwares are detected by utilizing the sets of rules or policies [1, 2, 3, 4, 5, 6]. If an attack shows a signature exactly matching one of the known signatures, then it can be easily detected. However, this mechanism may not be effective against new malwares with unknown signatures.

In anomaly detection approaches machine learning algorithms are first used to obtain classifiers from the known

Manuscript received April 15, 2014. This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number W911NF1210060 and W911NF13110143, and by the Office of Naval Research under grant number N00014-11-1-0471.

Seonho Choi is with the Department of Computer Science, Bowie State University, Bowie, MD 20723 USA (e-mail: schoi@bowiestate.edu).

Michael Bijou is with the Department of Computer Science, Bowie State University, Bowie, MD 20723 USA (e-mail: mike.bijou@gmail.com).

Kun Sun is with the Center for Secure Information Systems, George Mason University, Fairfax, VA 22030 USA (e-mail: ksun3@gmu.edu).

Edward Jung is with the Department of Computer Science and Software Engineering, Southern Polytechnic State University, GA USA (e-mail: ejung@spsu.edu).

malware behaviors [3, 7, 8, 9, 10, 11, 12]. Then, the classifier will be used at run-time to detect malwares. Although anomaly detection is able to detect new or evolved malwares more effectively compared to the signature-based approaches, it sometimes causes high false positive.

Malware detection techniques may also be classified into two different categories, static vs. dynamic. In static approaches the classifier or signatures will be obtained only from the apps' codes, which remove the necessity to collect the data by running the apps [13, 14, 15, 3, 16, 17, 18, 6]. These approaches have limitations. Metamorphic and polymorphic techniques may be applied to generate new signatures for the same virus. If polymorphic techniques are used, due to the encryption, it is difficult to generate the signatures; if metamorphic techniques are applied, all the codes will be obfuscated. In dynamic approaches, they obtain classifiers or signatures only based upon data obtained at run-time [19, 20, 21, 4, 7, 9, 10, 12]. They don't have limitations as in the static approaches, but it becomes critical how to design and conduct app running experiments to capture their behaviors or signatures in a comprehensive manner.

A few attempts were made to apply signal processing techniques to the design of intrusion detection systems for wired networks [22].

III. METHODOLOGY

The overview of our tracing tool is shown in Figure 1.

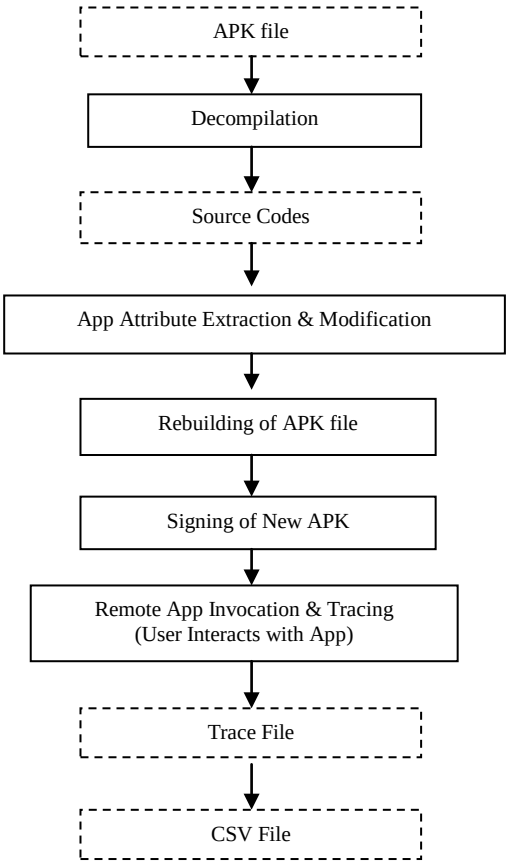

Fig. 1. Overview of our tracing tool

A. *Decompiling of APK File into Source Codes*

By using the Android apktool's decode function we need to decompile an Android APK into its constituent parts. It uses baksmali, a disassembler for Dalvik executable (.dex) files used by Dalvik Virtual Machine. Dex files themselves are created from Java class files generated from Java source code. After baksmaling completes, apktool loads the application's binary resource table, which stores or references resources defined by the application, such as xml files containing strings or layouts used by the application. It uses the resource table to decode the application's AndroidManifest.xml, which contains essential information necessary for execution of the app DVM such as the package name, the activities, services, process, and permissions used in the application. The end result of the decoding process is a series of directories containing the assets, Android library files, the application's resources, a "Smali" directory containing decompiled Smali code files, similar to the application's original source for rebuilding the application. Figure 2 shows the detailed steps and output directory structure after APK files are decompiled. Figure 3 shows an example Smali code obtained after the apktool is applied.

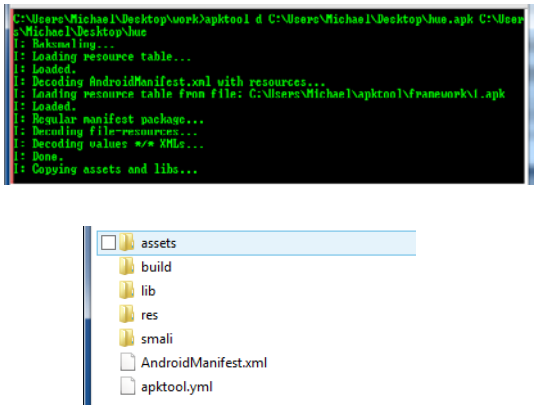

Fig. 2. APK file is decompiled into a directory-structured collection of source codes

```
.class Lcom/google/ssearch/SearchService$1;
.super Ljava/util/TimerTask;
.source "SearchService.java"


# annotations
.annotation system Ldalvik/annotation/EnclosingMethod;
    value = Lcom/google/ssearch/SearchService;->provideService()V
.end annotation

.annotation system Ldalvik/annotation/InnerClass;
    accessFlags = 0x0
    name = null
.end annotation


# instance fields
.field final synthetic this$0:Lcom/google/ssearch/SearchService;


# direct methods
.method constructor <init>(Lcom/google/ssearch/SearchService;)V
    .locals 0
    .parameter

    .prologue
    .line 1
    iput-object p1, p0, Lcom/google/ssearch/SearchService$1;->this$0:Lcom

    .line 355
    invoke-direct {p0}, Ljava/util/TimerTask;-><init>()V

    return-void
.end method
```

Fig. 3. Example smali source codes obtained from the apktool with decompilation option

B. *App Attribute Extraction & Modification*

Next, various apk attributes are extracted mainly from the AndroidManifest.xml file obtained in the previous decoding step. The names of the package, activities, and services are extracted along with the permission information. These will be stored in a temporary file for later reference and the AndroidManifest.xml file will be modified by introducing necessary attributes such as "android:debuggable" in some of the xml elements. Figure 4 shows an example AndroidManifest.xml file.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.program1.buttons"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.program1.buttons.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Fig. 4. Example AndroidManifest.xml from which the app attributes are extracted.

C. *Rebuilding of APK File*

Then the modified APK file is created by using the apktool with "build" option. It rebuilds the application by applying the smali, the reverse of baksmali, to the previously extracted and modified set of assets in the directory structure. The detailed steps are given in Figure 5.

```
C:\Users\Michael\Desktop\work>apktool build C:\Users\Michael\Desktop\hue C:\User
s\Michael\Desktop\hue-b.apk
I: Checking whether sources has changed...
I: Smaling...
I: Checking whether resources has changed...
I: Building resources...
I: Copying libs...
I: Building apk file...
```

Fig. 5. Rebuilding of APK fie.

D. *Signing of New APK File*

The Android system requires that all installed applications be digitally signed with a certificate whose private key is held by the application's developer [23]. The Android system uses the certificate as a means of identifying the author of an application and establishing trust relationships between applications. The certificate is not used to control which applications the user can install. The certificate does not need to be signed by a certificate authority: it is perfectly allowable, and typical, for Android applications to use self-signed certificates.

The important points to understand about signing Android applications are [23]:

- All applications *must* be signed. The system will not install an application on an emulator or a device if it is not signed.
- To test and debug your application, the build tools sign your application with a special debug key that is created by the Android SDK build tools.
- When you are ready to release your application for end-users, you must sign it with a suitable private key. You cannot publish an application that is signed with the debug key generated by the SDK tools.
- You can use self-signed certificates to sign your applications. No certificate authority is needed.
- The system tests a signer certificate's expiration date only at install time. If an application's signer certificate expires after the application is installed, the application will continue to function normally.
- You can use standard tools — Keytool and Jarsigner — to generate keys and sign your application APK files.
- The Android system will not install or run an application that is not signed appropriately. This applies wherever the Android system is run, whether on an actual device or on the emulator.

We used jarsigner [24] to "sign" the application. Jarsigner is part of the basic Java Development Kit, distributed by Oracle. In our experiment, we used a single, self-signed certificate, generated through the Java SDK's keytool, to sign tested applications after decoding and rebuilding them.

E. *Remote App Invocation & Tracing*

All of the remaining steps in the androiddebug.bat process will involve using the Android Debug Bridge [25]. The Android Debug Bridge, or adb, is a command line tool that lets a computer communicate with an instance of the Android emulator or an Android device connected via USB. It consists of a client and server, which run on the development computer, and a daemon, which runs on an emulator instance or Android device. The server manages communications between the client and the daemon; the client executes adb commands from command line; the daemon provides an endpoint for the server to establish a connection to on the device or emulator. Figure 6 shows a typical debugging environment for Android, and Figure 7 provides the overview of our tracing tool.

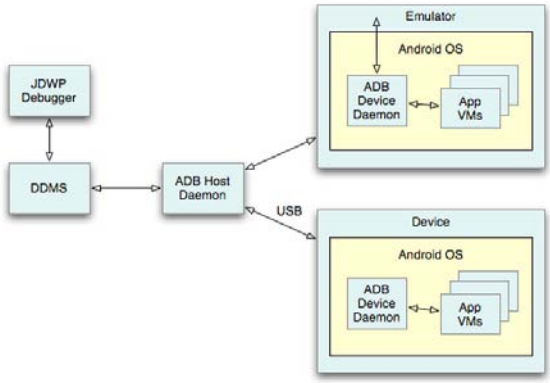

Fig. 6. Typical debugging environment for Android

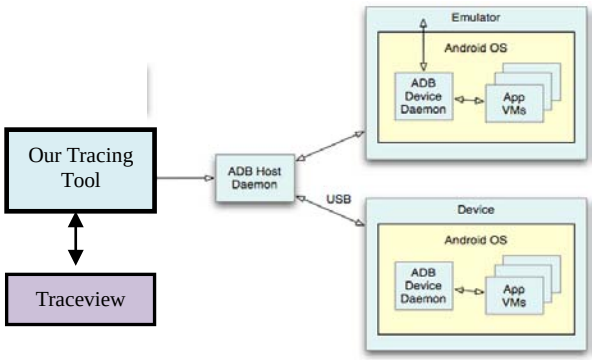

Fig. 7. Our tracing tool is interfacing with the ADB Host Daemon.

Android Debug Bridge (adb) has a variety of abilities that can aid in Android development. It can install applications, pull files from a device or emulator, push files to it, list devices or emulators connected to the development computer, or start a remote shell on the device or emulator to execute commands directly on the operating system. adb is commonly used by enthusiasts to install APKs downloaded to their computers directly to their connected devices. It is also used for testing and debugging during Android application development and troubleshooting device issues.

The command adb shell will start a remote Unix shell on a connected Android device or emulator instance, which will allow a user to run commands on an emulator or device. The shell can access binaries stored in the Android file system in /system/bin/. The most commonly used binaries in this area are the activity manager (am) and the package manager (pm). Activity manager will allow a user to issue commands to start activities, kill processes, broadcast intents, and other system actions. Activity manager is quite heavily used in our batch file to carry out the tracing and other tasks. Package manager can be used to perform actions and queries on installed applications on a device or emulator.

After an application being tested is decoded, edited, rebuilt, and signed, adb install is used to install the modified application onto a physical device connected by USB (for our experiments, we used two types of device, the Samsung SGH-i747 [Galaxy SIII] and the Samsung SGH-I857 [DoubleTime]). A remote shell is started on the device using adb shell, and we use "am start" option to start the newly installed application, and use "adb shell ps" to generate a list of the running processes on the device, including the spawned children processes by our modified application.

From this list of processes our tool selects the ones that were generated by the application to be traced by using the package name and class path names obtained in the attribute extraction step. For each process id obtained, it will create a remote profiling process by issuing the command

START /b adb shell am profile pid start

This command will call the activity manager and instruct it to start profiling for the given pid. Every START invocation runs the command given in its parameter and returns immediately, thus allowing multiple invocations of adb processes for multiple pids. This is typically utilized for concurrent and asynchronous execution of commands. Profiling process execution will allow us to monitor and record the details on its execution including what methods are invoked and what resources are being utilized at which times. Each profiler process is run for user provided duration – e.g., five minutes, during which time the user interacts with the running app on the device, while recording the execution details. After the profiling duration has elapsed, our tool sends the "adb shell am profile stop" command to stop profiling, and wait another 60 seconds, to give enough time for the device to flush out the buffered profiling data to a trace file in a microSD card. The microSD card used in our experience is a consumer-grade card, commonly used in mobile phones, rated for speeds of 2 MB/s data transfer. Since trace files can run anywhere from a few megabytes to up to a hundred, we tested with different file sizes to find out the safe delay, and found out that 60 seconds waiting time is necessary. The trace file is a file containing the binary and textual trace data itself and keys which provide a mapping from identifiers in the binary to thread and method names [25]. Figure 8 shows the example of multiple processes created from one App's execution.

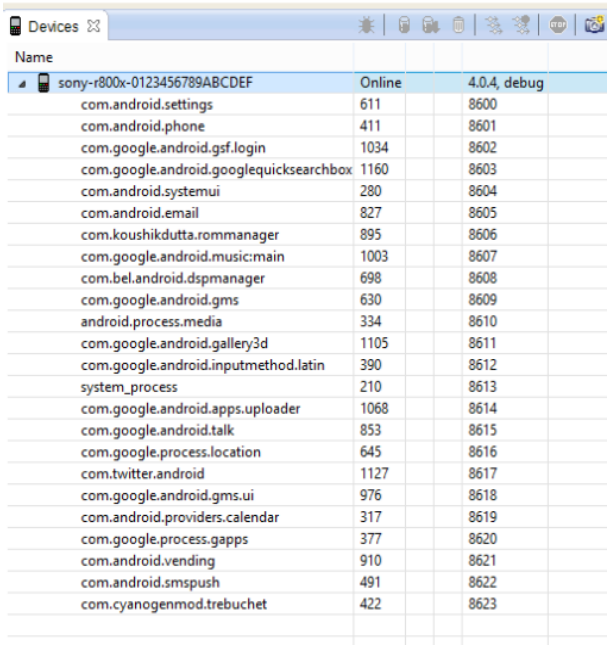

Fig. 8. Multiple processes are created from one App's execution. Different trace files will be created for the processes that were identified to belong to the same application.

After the trace file has been generated and stored on the microSD card, our tracing tool will use "adb pull" to transfer the file from the microSD card in the device to the computer. "adb pull" and "adb push" can be used to transfer any file from the device or emulator to the computer and vice versa.

F. Traceview

Traceview is a graphical viewer for execution logs that you create by using the Debug class to log tracing information in your code. Traceview can also help you debug your application and profile its performance [26]. However, our tool uses the "traceview –r" command to convert the trace file from its raw binary format to a human-readable text file, which can be analyzed and operated on for the purposes of our study. Our tool finishes up by uninstalling the application

from the device and deleting temporary files. Figure 9 shows the Method Stats section in the trace file, and Figure 10 gives an example Call Times section in the trace file.

```
*methods
0x56f553c0  java/lang/Long  equals  (Ljava/lang/Object;)Z   Long.java   197
0x56f55430  java/lang/Long  hashCode    ()I Long.java   290
0x56f55468  java/lang/Long  intValue    ()I Long.java   295
0x56f554a0  java/lang/Long  longValue   ()J Long.java   305
0x56f54c80  java/lang/Long  <init>  (J)V    Long.java   75
0x56f54f58  java/lang/Long  parse   (Ljava/lang/String;IIZ)J    Long.java   357
0x56f54f90  java/lang/Long  parseLong   (Ljava/lang/String;)J   Long.java   319
0x56f54fc8  java/lang/Long  parseLong   (Ljava/lang/String;I)J  Long.java   338
0x56f551c0  java/lang/Long  toString    (J)Ljava/lang/String;   Long.java   439
0x56f55230  java/lang/Long  valueOf (J)Ljava/lang/Long; Long.java   728
```

Fig. 9. Method Stats section in the trace file showing summary of method call related information.

```
Call Times
id  t-start    t-end  g-start    g-end    excl.    incl.  method
 1       13       25        3       15       12       12  android/os/Debug.startMethodTra
 1       48       98       38       88       35       50  android/os/ParcelFileDescriptor
 1       78       93       68       83       15       15  android/os/Parcel.closeFileDesc
 1      127      162      117      152       23       35  android/os/Message.recycle ()V
 1      143      155      133      145       12       12  android/os/Message.clearForRecy
 1      165      313      155     1356       53      148  android/os/MessageQueue.next ()
 1      172      185      162      175       13       13  android/os/SystemClock.uptimeMi
 1      192      195      182      185        3        3  android/os/MessageQueue.pullNex
 1      203      207      193      197        4        4  java/util/ArrayList.size ()I
 1      217      225      207      215        8        8  android/os/Binder.flushPendingC
 1      228      282      218     1325       10       54  java/lang/Object.wait ()V
 1      233      277      223     1320       44       44  java/lang/Object.wait (JI)V
 1      255      255      245     1298        0        0  (context switch)
```

Fig. 10. Call Times section in the trace file showing the method call history made by the process. This data may be utilized in creating API call traces.

G. *Conversion of Trace File into CSV Format*

CSV is a common, relatively simple file format that is widely supported by consumer, business, and scientific applications. Among its most common uses is moving tabular data between programs that natively operate on incompatible (often proprietary and/or undocumented) formats. This works because so many programs support some variation of CSV at least as an alternative import/export format [27]. We decided to convert the trace files into CSV format files to enhance the portability and usability of other library routines that support csv format. Our tool utilizes various java classes to implement this conversion process. Figure 11 shows the example CSV file obtained by converting the original trace file.

```
id,t-start,t-end,g-start,g-end,excl.,incl.,method,,args,return,num. args
 1,13,25,3,15,12,12,android/os/Debug,startMethodTracing,,(Ljava/lang/Stri
 1,48,98,38,88,35,50,android/os/ParcelFileDescriptor,close,,(),V,
 1,78,93,68,83,15,15,android/os/Parcel,closeFileDescriptor,,(Ljava/io/Fil
 1,127,162,117,152,23,35,android/os/Message,recycle,,(),V,
 1,143,155,133,145,12,12,android/os/Message,clearForRecycle,,(),V,
 1,165,313,155,1356,53,148,android/os/MessageQueue,next,,(),Landroid/os/M
 1,172,185,162,175,13,13,android/os/SystemClock,uptimeMillis,,(),J,
 1,192,195,182,185,3,3,android/os/MessageQueue,pullNextLocked,,(J),Landro
 1,203,207,193,197,4,4,java/util/ArrayList,size,,(),I,
 1,217,225,207,215,8,8,android/os/Binder,flushPendingCommands,,(),V,
 1,228,282,218,1325,10,54,java/lang/Object,wait,,(),V,
 1,233,277,223,1320,44,44,java/lang/Object,wait,,(JI),V,
 1,255,255,245,1298,0,0,(context,,switch),
 6,6,1145,245,2202,32,1139,android/os/Binder,execTransact,,(IIII),Z,
 6,16,45,255,284,24,29,android/os/Parcel,obtain,,(I),Landroid/os/Parcel;,
 6,33,38,272,277,5,5,android/os/Parcel,init,,(I),V,
 6,50,66,289,305,13,16,android/os/Parcel,obtain,,(I),Landroid/os/Parcel;,
```

Fig. 11. Trace files are filtered and converted into CSV format. Only relevant APIs with proper prefix class paths are extracted to be included in the CSV file.

H. *Automatic Repetition against a Collection of Apps*

To automate the trace file generation process for a given set of multiple apk files, we implemented another component in our tool to iteratively visit each app and apply the entire trace file generation process. It accepts the name of the directory as an argument. It starts by searching the input directory for subdirectories and file directories. It adds these to a file tree created using the Java Collections framework. The end result is a data structure with paths to every directory and file under the input directory. This structure is then recursively searched for files with the extension ".apk", which are Android package files, commonly referred to as Android "apps". And, for each of them it applies the previously introduced steps to create trace files in CSV format.

IV. EXPERIMENTS

We generated a set of trace files from the collection of about 500 malware-infected apps obtained from malware genome project database [28]. These infected apps are classified into 50 different categories as is shown in Figure 12, and are stored in a hierarchically organized directory structure. Figure 12 shows the category of malwares tested with our tracing tool.

```
ADRD                    7/7/2013 09:20:18 AM    7
AnserverBot             7/7/2013 09:20:18 AM    7
Asroot                  7/7/2013 09:22:16 AM    7
BaseBridge              7/7/2013 09:22:17 AM    7
BeanBot                 7/7/2013 09:22:23 AM    7
Bgserv                  7/7/2013 09:22:24 AM    7
CoinPirate              7/7/2013 09:22:24 AM    7
CruseWin                7/7/2013 09:22:24 AM    7
DogWars                 7/7/2013 09:22:24 AM    7
DroidCoupon             7/7/2013 09:22:25 AM    7
DroidDeluxe             7/7/2013 09:22:25 AM    7
DroidDream              7/7/2013 09:22:25 AM    7
DroidDreamLight         7/7/2013 09:22:26 AM    7
DroidKungFu1            7/7/2013 09:22:27 AM    7
DroidKungFu2            7/7/2013 09:22:29 AM    7
DroidKungFu3            7/7/2013 09:22:31 AM    7
DroidKungFu4            7/7/2013 09:23:00 AM    7
DroidKungFuSapp         7/7/2013 09:23:06 AM    7
DroidKungFuUpdate       7/7/2013 09:23:06 AM    7
Endofday                7/7/2013 09:23:06 AM    7
FakeNetflix             7/7/2013 09:23:06 AM    7
FakePlayer              7/7/2013 09:23:06 AM    7
GamblerSMS              7/7/2013 09:23:07 AM    7
Geinimi                 7/7/2013 09:23:07 AM    7
GGTracker               7/7/2013 09:23:13 AM    7
GingerMaster            7/7/2013 09:23:13 AM    7
GoldDream               7/7/2013 09:23:13 AM    7
Gone60                  7/7/2013 09:23:18 AM    7
GPSSMSSpy               7/7/2013 09:23:18 AM    7
HippoSMS                7/7/2013 09:23:18 AM    7
Jifake                  7/7/2013 09:23:19 AM    7
jSMSHider               7/7/2013 09:23:19 AM    7
KMin                    7/7/2013 09:23:20 AM    7
LoveTrap                7/7/2013 09:23:26 AM    7
NickyBot                7/7/2013 09:23:26 AM    7
NickySpy                7/7/2013 09:23:26 AM    7
Pjapps                  7/7/2013 09:23:26 AM    7
Plankton                7/7/2013 09:23:30 AM    7
RogueLemon              7/7/2013 09:23:31 AM    7
RogueSPPush             7/7/2013 09:23:31 AM    7
SMSReplicator           7/7/2013 09:23:32 AM    7
SndApps                 7/7/2013 09:23:32 AM    7
Spitmo                  7/7/2013 09:23:32 AM    7
Tapsnake                7/7/2013 09:23:32 AM    7
Walkinwat               7/7/2013 09:23:32 AM    7
YZHC                    7/7/2013 09:23:32 AM    7
zHash                   7/7/2013 09:23:33 AM    7
Zitmo                   7/7/2013 09:23:33 AM    7
Zsone                   7/7/2013 09:23:33 AM    7
```

Fig. 12. Malware categories

As an example application of our tracing tool, normalized API feature vectors were obtained by analyzing the trace files created from the collection of malware infected apps that belong to the same category. Normalized API feature vector contains the most frequently invoked resource-critical APIs along with their normalized frequencies obtained from the aggregate frequencies. For example, we obtained a normalized feature vector set for DroidKungfu3 malware category. Data structure appropriate for dictionary style of data needs to be employed in obtaining these, and we adopted hash table in our implementation.

V. CONCLUSION

Our experiments' goal was to develop an efficient API call

tracing tool for dynamic app behavior analysis. This tool may be applied to a wide range of applications including designing effective malware detection techniques on mobile devices. Project modules developed from the different components in this tool will be incorporated into the department's Senior Capstone course at Bowie State University. In the future, visualization components will be added to the experiment to make the data easier to interpret. We will design and implement a malware detection technique based upon the finite state machine-based pattern recognition.

REFERENCES

[1] A. Desnos and G. Gueguen. Android: From reversing to decompilation. Blackhat, 2011.

[2] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[3] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical Report CSTR- 4991, Department of Computer Science, University of Maryland, College Park, November 2009.

[4] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 239–252, New York, NY, USA, 2008. ACM.

[5] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semanticallyrich application-centric security in android. In *Proceedings of the 2009Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[6] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, Feb. 2012.

[7] L. Liu, G. Yan, X. Zhang, and S. Chen. Virusmeter: Preventing your cellphone from spies. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 244–264, Berlin, Heidelberg, 2009. Springer-Verlag.-2792-2 50.

[8] A.-D. Schmidt, J. H. Clausen, A. Camtepe, and S. Albayrak. Detecting symbian os malware through static function call analysis. Number March2006, pages 15–22. IEEE, 2009.

[9] A. Shabtai and Y. Elovici. Applying behavioral detection on androidbased devices. In *MOBILWARE*, pages 235–249, 2010.

[10] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. "andromaly": A behavioral malware detection framework for android devices. *J. Intell. Inf. Syst.*, 38(1):161–190, 2012.

[11] P. Teufl, S. Kraxberger, C. Orthacker, G. Lackner, M. Gissing, A. Marsalek, J. Leibetseder, and O. Prevenhueber. Android market analysis with activation patterns. In *MOBISEC*, 2011.

[12] M. Zhao, F. Ge, T. Zhang, and Z. Yuan. Antimaldroid: An efficient svmbased malware detection framework for android. In C. Liu, J. Chang, and A. Yang, editors, *ICICA (1)*, volume 243 of *Communications in Computer and Information Science*, pages 158–166. Springer, 2011.

[13] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[14] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing interapplication communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[15] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova. Detection of malicious applications on android os. In *Proceedings of the 4th international conference on Computational forensics*, IWCF'10, pages 138–149, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, Feb. 2012.

[17] S. Kim, J. I. Cho, H. W. Myeong, and D. H. Lee. A study on static analysis model of mobile application for privacy protection. In J. J. (Jong Hyuk) Park, H.-C. Chao, M. S. Obaidat, and J. Kim, editors, *Computer Science and Convergence*, volume 114 of *Lecture Notes in Electrical Engineering*, pages 529–540. Springer Netherlands, 2012.

[18] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

[19] B. Dixon, Y. Jiang, A. Jaiantilal, and S. Mishra. Location based power analysis to detect malicious code in smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 27–32, New York, NY, USA, 2011. ACM.

[20] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[21] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: Automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, MCS '11, pages 21–26, New York, NY, USA, 2011. ACM.

[22] Urbashi Mitra, Antonio Ortega, John Heidemann, and Christos Papadopoulos, "Detecting and Identifying Malware: A New Signal Processing Goal", IEEE SIGNAL PROCESSING MAGAZINE [**110**], pp.107-111, SEPTEMBER 2006.

[23] Android Deveopers. "Signing Your Applications." Web. 15 April 2014 article was accessed. http://developer.android.com/tools/publishing/app-signing.html.

[24] ORACLE. "JARSIGNER - JAR SIGNING AND VERIFICATION TOOL." WEB. 15 APRIL 2014 ARTICLE WAS ACCESSED. HTTP://DOCS.ORACLE.COM/JAVASE/6/DOCS/TECHNOTES/TOOLS/WINDOWS/JARSIGNER.HTML.

[25] ANDROID DEVEOPERS. "ANDROID DEBUG BRIDGE." WEB. 15 APRIL 2014 ARTICLE WAS ACCESSED. HTTP://DEVELOPER.ANDROID.COM/TOOLS/HELP/ADB.HTML.

[26] ANDROID DEVEOPERS. "PROFILING WITH TRACEVIEW AND DMTRACEDUMP." WEB. 15 APRIL 2014 ARTICLE WAS ACCESSED. HTTP://DEVELOPER.ANDROID.COM/TOOLS/DEBUGGING/DEBUGGING-TRACING.HTML.

[27] Wikipedia. "Comma-separated values." Web. 15 April 2014 article was accessed. http://en.wikipedia.org/wiki/Comma-separated_values

[28] Yajin Zhou, Xuxian Jiang, "Dissecting Android Malware: Characterization and Evolution," Proceedings of the 33rd

IEEE Symposium on Security and Privacy (Oakland 2012), San Francisco, CA, May 2012.

Seonho Choi is currently a Professor in the Computer Science Department at Bowie State University, USA. His primary research interests lie in the computer and network security. In 1991, he received a B.S. in Computer Science and Statistics from Seoul National University in Korea and a Ph.D. in Computer Science in 1997 from the University of Maryland, College Park, USA.

Michael A. Bijou Jr. was born in Arlington, Virginia in June 1990. He is pursuing an undergraduate degree in Computer Science at Bowie State University, in Bowie, Maryland. Mr. Bijou is focusing on security research while studying at Bowie State.

He is currently a participant in the U.S. Army's Undergraduate Research Apprenticeship Program, while concurrently acting as a research assistant to Dr. Seonho Choi at Bowie State. Mr. Bijou has published four papers for the National BDPA IT Showcase, and has submitted a paper for the 2014 International Conference on Appropriate Technology. He is currently interested in mobile security and communications research.

Mr. Bijou is a member of the National BDPA's Washington, D.C. chapter. He is a two time Johnson and Johnson Scholar.

Kun Sun is a Research Professor in Center for Secure Information Systems (CSIS) at George Mason University, and will join the Computer Science department at College of William and Mary in August 2014. He received his Ph.D. in Computer Science from North Carolina State University in 2006. Dr. Sun was a Research Scientist in Intelligent Automation Inc. between 2006 and 2010. His current research focuses on trustworthy computing environment, moving target defense, smart phone security, cloud security, and wireless security. Dr. Sun is a member of IEEE and ACM.

Edward Jung received both a B.S. degree (1987) and a Ph.D. degree (1994) in computer science from the University of Minnesota, Minneapolis, USA. From 1994 to 2007, he worked at R&D labs (Bell Labs, Samsung Research) where he was a Director of a security research group at Samsung prior to moving to academia. During 2008-2009, he was a Murray visting professor of computer science at Rutgers University, New Brunswick, New Jersey, USA. Since 2009, he has been an Assistanct Professor with the Computer Science and Software Engineering Department at Southern Polytechnic State University, Marietta/Atlanta, GA, USA. His primary research interestes lie in the foundations of computer security, nework security, IP and design protection, and optimization of sequential systems. Mr. Jung is a senior member of the IEEE and a member of the ACM.