Analysis of Malicious and Benign Android Applications

MoutazAlazab, VeelashaMoonsamy Lynn Batten and RonghuaTian
School of Information Technology
Deakin University, Australia
{malazab, v.moonsamy, lynn.batten, rtia}@deakin.edu.au

Patrik Lantz
Ericsson Research
221 83 Lund, Sweden
{patrik.lantz}@ericsson.com

Abstract—**Since its establishment, the Android applications market has been infected by a proliferation of malicious applications. Recent studies show that rogue developers are injecting malware into legitimate market applications which are then installed on open source sites for consumer uptake. Often, applications are infected several times.**
In this paper, we investigate the behavior of malicious Android applications; we present a simple and effective way to safely execute and analyze them. As part of this analysis, we use the Android application sandbox Droidbox to generate behavioral graphs for each sample and these provide the basis of the development of patterns to aid in identifying it. As a result, we are able to determine if family names have been correctly assigned by current anti-virus vendors. Our results indicate that the traditional anti-virus mechanisms are not able to correctly identify malicious Android applications.

Keywords-Android, Dynamic, Mobile malware, Behavior graph, Treemap, Droidbox

I. INTRODUCTION

The Android operating system has been recently deployed by various companies such as Google, Motorola, Samsung, and Dell. The number of Android applications has reached the thousands in 2012 [1]. Attackers are looking for easy and rapid financial gains, and smartphones are easier targets than desktop computers. Malicious applications include a malicious code segment which causes damage and harms the smart device. Malicious applications not only infect user data, but can also steal private information such as passwords, credit card numbers and bank account numbers from smartphones.

Current anti-virus (AV) engines cannot deal with physical limitations in smartphone devices such as CPU, battery and memory cards. Additionally current AV methods were developed on malware attacking desktop computers and laptops[2] [3] and [4] while there are many indications that different kinds of malware are being developed for portable platforms such as smartphones[5].

Android applications can be installed from the Android marketand Amazon (https://market.android.com; http://www.amazon.com) as well as from many open source sites such as 4shared (www.4shared.com), mascobz (www.mascobz.com) and filecrop (www.filecrop.com). Malware authors can thus easily spread their malicious applications by downloading clean applications, injecting their Malicious code and then uploading them to these open source sites for general consumption by unwary users.

In the research literature on malware targeting desktops and laptops, one important approach has been to identify behavioral patterns[6] which then help to classify malware in the hope of better understanding its aims and so more quickly devising counter-measures. This is also the direction of the current paper. Our aim is to identify patterns of behavior in both benign and malicious applications which can distinguish one application from another.

To our knowledge, the work in this paper is the first to examine behavior in malicious applications using DroidBox ([7] and see Section IV). Our dataset comprises samples that were collected from publicly available sources. Each malicious application is executed for 60 seconds in a sandboxed environment and the generated log files are collected at the end of execution. Using Droidbox, we also generate two types of graphs (behavior graphs and treemap graphs) for each sample. Both graphs help us to analyze the activities performed during run-time and also to establish patterns between variants from the same malware family. These graphs also illustrate how some benign applications might leak data connected to short message service (SMS) texting and other features of the applications.

However, we also show that it can be difficult to distinguish between malicious and benign applications based on Droidbox output alone. For instance, not a single malicious application in our (relatively small) sample set invoked a cryptographic activity, while several of our clean applications did so. The use of encryption in malware targeting desktop computers is much more ([7], page 17) than appears to be the case in our Android sample.

The paper is organized as follows: Section II highlights the main contributions from the relevant literature. Section III describes our methodology, including the data collection and the implementation of our system architecture, and describes our tests.Section IV gives an extensive explanation of the workings of DroidBox and in Section V we analyze and discuss the experimental results. Section VI provides a discussion on the limitations of our work and its implications for future research.

II. RELATED WORK

In this section we present some of the recent work related to the propagation of malware on the Android platform.

1545-0678/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDCSW.2012.13

The widespread use of smartphones has provided a new way for malware authors to propagate infectious software. The application market with the highest percentage of malicious software is the Android market. This mobile platform saw a significant increase of five-fold among malicious applications within the first six months of 2011[8]. This is partly due to the loose permission granting structure currently in place for Android applications, as demonstrated in [9], [10] and [11].

Similar to desktop malware, there exist two common methods to investigate the structure and behavior of Android malware: static analysis and dynamic analysis. We discuss recent work in both of these areas.

Static

In[12], the authors focused on Android version 2.2. In their work, they collected 940 applications from both the Android and Amazon market. They proposed a tool, Stowaway, that applies static analysis on the collected sample applications, and then they map the permission with each operation. The aim of Stowaway is to investigate over-privileged permissions and developer errors. They built API mappings inside the Android emulator to determine the permissions required to interact with system APIs. The aim of the experiment was to check application permission and also to add a security exception to the application if it does not exist. They found that 779 APIs invoke normal calls and 428 APIs communicate with system services calls.

A similar study in [13] performed a static analysis on 25 Android applications using an extended version of the Julia system, which is a static analyser tool for Java bytecode. This tool can analyse dead code, nullness and termination. Since all Android applications are shipped in the Dalvik bytecode, they need to be converted to Java bytecode. The static analyzer keeps track of commonly used Java statements to generate an abstract overview of the application. The authors claim that their proposed framework can analyze any application even though its code does not include any explanatory notes from its initial writer.

Dynamic

In[14], Burguera et al. deployed their methodology in the form of an application, named Crowdroid, on the Android market. Crowdroid collects behavior patterns such as system calls of installed applications on the users' devices. This information is sent to a remote server, where the system calls are clustered using a K-means algorithm into two categories: benign and malicious. The authors initially tested their framework using 4 self-written malware which resulted in 100% detection accuracy. They then collected and executed two types of real-life malicious applications, labeled PJApps and HongToutou Trojans, from VirusTotal and with testing obtained 100% and 85% detection accuracy respectively.

Blasing[15] developed an execution environment named AASandbox (Android Application Sandbox). The authors make use of the Android emulator together with the Monkey tool [16], which mimics the user's behaviour so that the executions of applications closely resemble the ones in real-time. AASandbox logs the time of execution, the name of the system call and the IDs of the processes. The environment was tested on 150 clean Android applications and on written malware. In the future, the authors will apply machine learning algorithms to the information retrieved from the log files in order to detect anomalous applications and also, implement the framework as a cloud service.

Monitoring

In addition to static and dynamic work used in identifying and classifying malware, a different approach to securing a system is to monitor it for security breaches. In this subsection we present some such work.

Bugiel et al. [17] present a security framework named XManDroid which monitors the real-time communication between applications and verifies the inter-process communications against a set of pre-defined security policies. The aim is to prevent malicious applications from exploiting transitive permission properties to enable privilege escalation. In order to test their methodology, they included 7 types of attacks in their dataset which cover several possible scenarios whereby rogue applications request transitive permissions. For future work, the authors plan to integrate the methodology into the existing permission framework currently in use by Android.

In the work carried out by Portokalidis et al. [18], Paranoid Android is a security model implemented on remote servers where identical copies of smartphones are running in a virtual environment. A program, which resides on the device, collects all the necessary information needed to replay the execution and transmits it to the remote server. The information is re-executed on the virtual smartphones. The aim is to run constant security checks on applications while maintaining minimal computational and battery overhead.

In the following section, we introduce our experimental environment and describe our tests.

III. DATASETS AND ENVIRONMENT

In this section we provide details about dataset collection for both malicious and benign applications. Additionally, this section describes how we setup the experimental environment to monitor behavior for these applications.

A. Datasets

- **Clean Applications Collection:**

For the clean application samples, we chose 23 popular downloadable applications from the Android market.

As mentioned earlier, Android applications in the Android market may be infected with malware([14] and [19]). In order to ensure that the clean applications chosen were really clean, we tested them by uploading each sample individually to VirusTotal where five robust AV engines (Sophos, Symantec, F-Secure, TrendMicro and MacAfee) were used to verify cleanness.

The samples chosen were: adobe_flash_player, android_market, beautifulwidgetsbetterlayouts, antivirus, applanet, cut-the-rope, doodle, ebuddy, facebook, fancywidgetpro, fruitninja, gamefly, gingerbreak, googlemaps, imdb, kayak, mob4, mobi, nav4, pes2011, shazamencore, Shirley, youtubedownloader.

- **Malicious Applications Collection:**

Smartphone malicious applications can be collected from several open source sites such as Contagion, Offensive and VXHeavens. Malware samples used in this research were collected from Contagiodump[20] which provides smartphone malware that infects various smartphone platforms such as Android, iPhone, BlackBerry and Windows mobile. Since we were only interested in malware that targeted Android devices, we picked from the Contagiodump folder files only those applications with the following extensions:

a) Application package file (.apk)

b) Java archive (.jar)

This left us with 44 malicious applications targeting Android-based devices. In order to determine if some were variants of others, we again employed VirusTotal and used only Sophos to name them and group them into families. Family naming is a vendor-specific process and so not all AV vendors would adopt the same nomenclature as Sophos, so one aim of our first test was to determine how accurate their naming was.

Since we were interested in examining families, we show in Table 1 those malware families we obtained, their Sophos names, and the number of variants for each family, as long as this was at least 2, along with the time period in which they first appeared as identified by Sophos. This leaves us with a total of 33 malicious applications grouped into families.

TABLE 1.Malicious applicationsSophos family names and number of variants used

Family Name	No. of Variant	Appearance time period
BBridge	5	Oct 2011 – Dec 2011
PJApps	5	Jun 2011 – Dec 2011
RootCage	3	Oct 2011 – Nov 2011
Gone60	5	Dec 2011
SMSRep	4	Aug 2011 – Dec 2011
Kmin	4	Oct 2011 – Nov 2011
KongFu	3	Jun 2011 – Oct 2011
AdmSMS	2	Jul 2011 – Nov 2011
FakPlay	2	Mar 2011 – Apr 2011

B. Experimental Environment

We ran each application, both malicious and benign, in a the Droidbox sandbox environment. We were particularly careful to prevent contamination of our samples. Before starting, we generated the three hashes, MD5, SHA1 and SHA256 [21], and after each test, the hashes were again generated to ensure that the code had not been changed.

We ran both benign and malicious applications in the DroidBox sandbox [7]which monitors API calls and data leaks. DroidBox was designed to enable interaction with a sample, collecting certain information about its activities over a fixed time period, and had to be manually terminated by the user. We modified it to automatically stop monitoring each sample after exactly 60 seconds so that we could compare the output for the samples more easily. (Other research on malware for desktop computers indicates that 60 secs is sufficient to extract the important information [22], however, this is likely to change in the mobile environment as applications can be returned to markets to be redistributed possibly with multiple infections.)

Figure 1 describes the architecture of our experiment. The clean applications were scanned through Symantec antivirus, installed on the host machine, and then uploaded to VirusTotal to confirm cleanness.

Next, both clean and malicious samples were run through DroidBox that was installed in the VM machine. We created a virtual device configuration for each sample in order to avoid contamination. The sandbox environment comprised an Android emulator and Android SDK, which provides the tools and APIs necessary to develop applications on the Android platform using Java. For each sample, DroidBox created a log file and two graphs visualizing the behavior; these items were used in the analysis of the data.

- **Setting up the sandbox environment.**

The research laptop had Windows 7 installed with the following specifications: i7 CPU 2.7 GHZ, 8 GB RAM DDR3 and 720 GB hard disk; on the machine, we also installed the virtual machine VMware player version 3.1.1 build-282343. Inside the virtual machine we installed Ubuntu 11.10 32bit and the following:

a) Android SDK (A Software Development Kit that allows users to create and test Android applications. http://developer.android.com/sdk/index.html)

b) PyLab and Matplotlib (Which provide visualization of the analysis results. http://matplotlib.sourceforge.net/and http://www.scipy.org/PyLab

c) HashMyFiles v1.80 (A utility to calculate hash values. http://www.nirsoft.net/utils/hash_my_files.html)

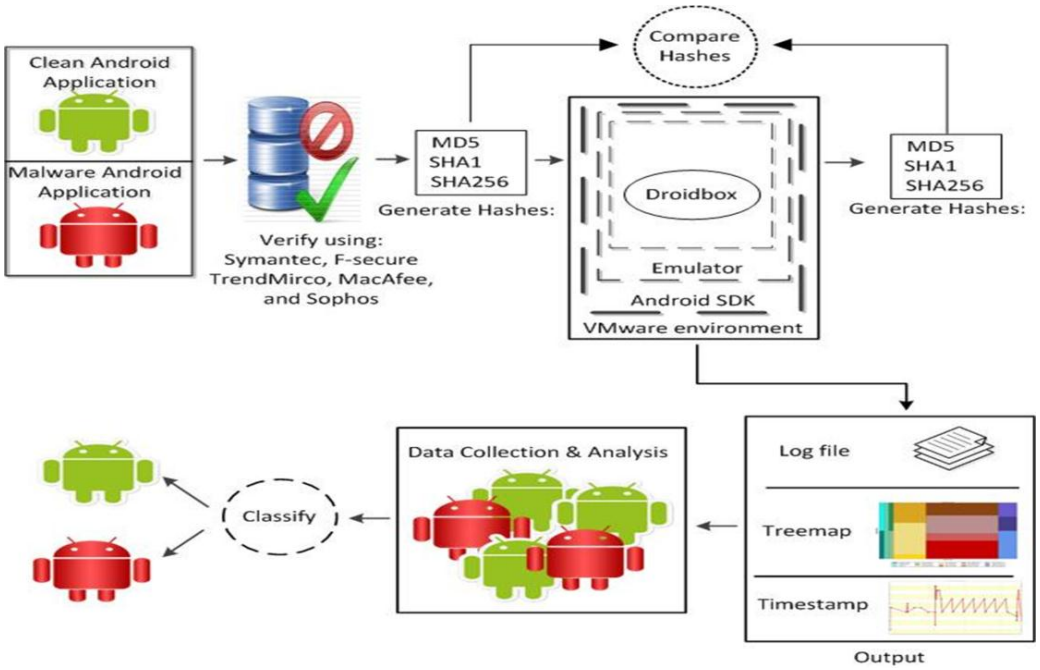
Since mobile malware can be spread from mobile device to PC or vice versa [23], we cancelled all interaction between the virtual machine and the Windows host during analysis.

- **Checking Hash Values.**

In this section we explain how we checked the applications hashes before and after exaction. The main reason for checking the hash value is to check if the application changes during the analysis.

1. We used HashMyFiles installed in the Windows host to generate the hash value.

2. For both malicious and benign applications, we generated the three hash values MD5, SHA1 and SH256 and copied them into a spread sheet with the date and time taken.



F igure 1. Architecture of the Behavior Monitoring

3. After running each sample for 60 secs in Droidbox, we again generated the three hash functions and checked for differences.

It is noteworthy that in all cases, the hash functions before and after execution matched, indicating that there had been no change to the sample.

- **The Tests.**

Our first test is on the families of malicious applications identified by Sophos in Table 1. In Section V, we combine the information from the graphs generated by Droidbox by family to see if the graphs indeed appear to represent the family. We also consider family designations by Sophos, TrendMicro and F-Secure also available in VirusTotal (See Table 2).

In our second test, we examine the benign applications to determine if there is any suspicious behavior. In fact, we find that several of them leak information and so can be considered suspicious or prone to abuse by malware writers. This will lead us, in future work, to try to determine if such applications could be easily compromised.

IV. DROIDBOX

We devote this section to a detailed description of Droidbox and its capabilities. DroidBox[24] is a dynamic analysis tool for Android applications targeting Android version 2.1. The tool is based on TaintDroid[25] for detecting information leaks but has been extended, by modifying the Android framework, to monitor API calls of interest invoked by an application.

Applications are executed within the Android SDK emulator and logs are issued for each monitored behavior and collected in the host operating system. A text-based report is generated after analysis has ended and provides a summary of the execution. Additionally, two graphs are generated to visualize the behavior of a sample, see Figures 2 and 3 for examples.

On mobile phones, malware has been discovered that listens for incoming SMS and forwards this information to the attacker. In [25], TaintDroid was used to track sensitive data originating from the phone's database. DroidBox can extend this approach by adding and modifying output channels throughout the Android framework to detect leaks via outgoing SMS and to disclose full details of the network communication, not only in network leak scenarios.

The file AndroidManifest.xml, included in the Android package, contains permissions that are needed for the application to interact with the operating system, for example, connecting to the Internet, sending SMS, making calls and receiving incoming SMS. Applications that need to interact with any resources must declare the appropriate permission in the manifest file. It has been demonstrated that malicious Android applications can circumvent the permission policies[26] and [27], thus DroidBox compares each monitored operation that requires any permission with the package's manifest file to check if any permission policy has been bypassed.

Some malicious Android applications can evade anti-virus software by performing obfuscation and changing themselves during run-time [28]. Obfuscation may include cryptographic functions applied to the data. DroidBox is designed to detect applications as they invoke cryptographic keys or perform encryption or decryption on the data

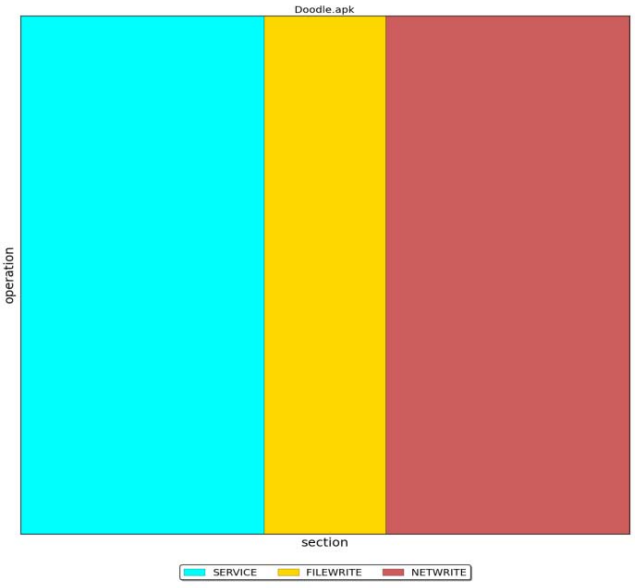
Malicious Android applications can perform phone calls or send SMS to premium rate numbers that are declared by the

attacker. DroidBox can disclose these operations by listening to API calls when SMS and calling methods are invoked by a sample.

A. *Treemaps*

Treemaps[29] display a tree structured graph and its data as nested rectangles. Each branch of the tree is assigned a rectangle that is divided into smaller rectangles representing sub-branches. The area of a rectangle is proportional to the dimension of the data in the leaf node and each leaf node is colored for visual clarity. The main motivation for the use of treemaps is to provide a human visualization of tree-structures that simplify the presentation of multi-dimensional data. The authors of [30] used treemaps (and other graphs) in malware analysis arguing that they "can effectively support a human analyst" in detecting and classifying malicious behavior.

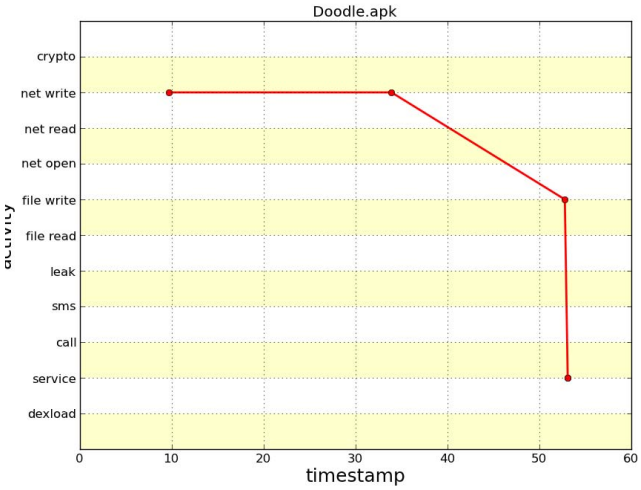
Figure 2 shows an example of a treemap generated for the benign sample called Doodle.apk, that is the popular Doodle Jump game; the cyan colored part represents the section related to services that have been started, yellow part represents file activity, in this case write operations and the red part shows the network write operations. In general, the wider a section becomes (horizontally), the more frequently the corresponding operation is executed; the higher a section rises (vertically), the more frequently the corresponding operation has occurred within the section. The position of the sections and operations are fixed, as well as the color (which represents operation type).



F igure 2. An example of a Treemap Graph

B. *Behavior Graph*

The behavior graph produced by Droidbox describes the temporal order of the operations. Figure 3 shows the output of the sample doodle.apk that was executed for 60 seconds. On the x- axis the time of the monitored operation is shown, while the y-axis describes what kind of operation type was monitored. This graph is generated by picking operations and timestamps for each operation from the sandbox log and plotting them.



F igure 3. An example of a Behavior Graph

V. EXPERIMENTAL RESULTS

In this section, we give the details of the two tests described in Section III.

We observed that the majority of the malicious applications in our dataset are games or applications intended for changing the wallpaper. We also observed that during execution of the two applications classified by Sophos as the AdmSMS family, other operations were performed in the background without the user's knowledge. Figure 4 shows a snapshot from the log file generated by this family; it appears that the application sends SMS after 20 seconds from the start of the execution. The log shows when the application tried to send an SMS, what the message was and to which number it was being sent. The malicious activity has messages sent to a premium-rate number.

```
[Sent SMS]
----------
    [20.3702008724]       Number: 10621900
                  Message: M6307AHD
    [20.3844468594]       Number: 10626213
                  Message: aAHD
    [20.4062190056]       Number: 106691819
                  Message: 95pAHD
    [20.4261310101]       Number: 10665123085
                  Message: 58♯28AHD
```

F igure 4. Snapshot of AdmSMS family log output

Naming of Malicious Families

In order to determine if existing anti-virus engines were classifying Android-based malware accurately, we compared the names assigned to the 33 samples in Table 1 by three vendors: Sophos, TrendMicro and F-Secure and then considered the Droidbox graphs for the samples to see if there was support there for the assigned names. Table 2 gives the raw output of the three AV vendor names along with the decision of the research team based on Droidbox output.

As is evident from the data in the table, in many cases, the AV vendors disagreed on the labels. We do not know how long the AV vendors executed samples when trying to identify them and different times may have been used. As mentioned earlier in the paper, in our testing, we used one minute. Differences in time of execution (and in the environment) may account for the differences of opinion in the table.

Regarding the samples numbered 1a,b,c,d,e: the first three all use the service and netwrite operations. Filewrite is also used by 1a and 1c. However, 1d does not use service and because of this, may not belong in the bridge family. 1e does use service, but not netwrite; it does use filewrite, so it may be bridge.

Now samples 3 (not classified by F-Secure), 4,5 and 6 all use service and netwrite. Some, but not all, use netopen. While the vote of the AV vendors F-Secure and TrendMicro identifies them as DroidDream, both Sophos and our analysis identifies them as PJapps. However, why is this not the same family as bridge? (Sample 23 also uses service and netwrite and so could be included in the bridge family.)

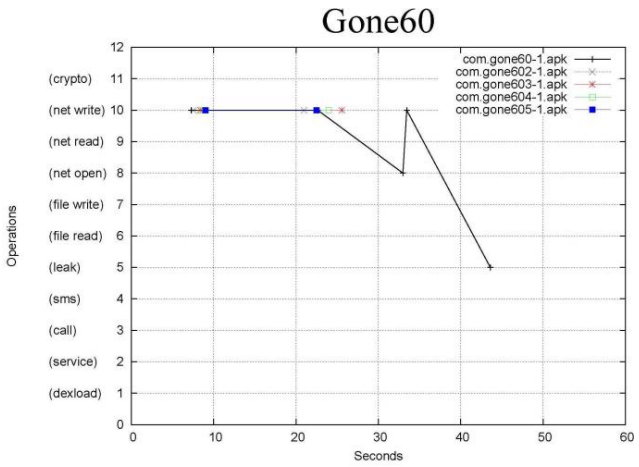
There were also some difficulties with the families Gone and RootCage. In comparing them, samples 7,8 and 9 (designated RootCage by Sophos and given other names by the other vendors) all call netwrite twice and nothing else. Gone 60 (10a) calls netwrite 3 times and open once while 10b,c, d and e all call netwrite exactly twice and have behavior and treemap graphs that look exactly like those of the RootCage samples.

So as far as we can see, based on our behavior graphs, seven of these samples are equivalent. Only one (10a) is different. Thus, there may really be only one family here apart from the single exception. We noticed also that 10b, c ,d and e all appeared on the same date - 29/09/2011 while 10a appeared on 23/09/2011. It may be the case that 10a was a trial version that the malware author released on the market before changing to the others. The five samples 10a, b,c,d and e are shown in Figure 6.

On a final note in the discussion of the Table 2 data, we raise the question: should a sample only belong to a family with SMS in the name if the graph includes SMS as an operation? It may be the case that since we only run the

TABLE 2. Classification decisions

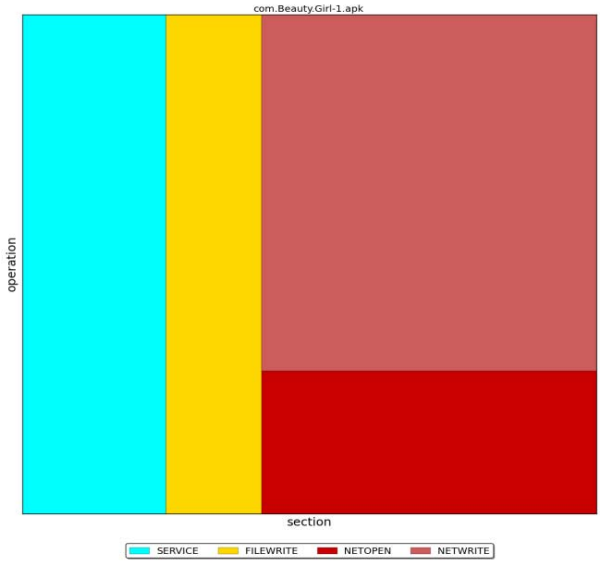
App No.	F-Secure	Trend-Micro	Sophos	Our decision
1a	BaseBridge.A!mfb	BRIDGE.AK	BBridge-F	BBridge
1b	BaseBridge.A	BBRIDGE.M	BBridge-E	BBridge
1c	BaseBridge.A	BBRIDGE.A	BBridge-E	Bbridge
1d	BaseBridge.A	BRIDGE.B	BBridge-A	??
1e	BaseBridge.A	BRIDGE.B	BBridge-A	Bbridge
2	DroidDream.B	DORDRAE.L	PJApps-E	??
3		DORDRAE.N	PJApps-E	PJApps
4	DroidDream.B	DORDRAE.L	PJApps-D	PJApps
5	DroidDream.B	DORDRAE.L	PJApps-D	PJApps
6	DroidDream.B	DORDRAE.L	PJApps-D	PJApps
7	DroidRooter.A	LOTOOR.A	RootCage-A	gone or rootcage
8	DroidRooter.A	LOTOOR.A	RootCage-A	gone or rootcage
9	DroidRooter.A	LOTOOR.A	RootCage-A	gone or rootcage
10a	GoneSixty.A	GONESIXTY.A	Gone60-A	Gone
10b	GoneSixty.A	GONESIXTY.A	Gone60-A	gone or rootcage
10c	GoneSixty.A	GONESIXTY.A	Gone60-A	gone or rootcage
10d	GoneSixty.A	GONESIXTY.A	Gone60-A	gone or rootcage
10e	GoneSixty.A	GONESIXTY.A	Gone60-A	gone or rootcage
11	Jifake.A	JIFAKE.E	SMSRep-M	SMSRep or JIFake
12	HippoSms.A	HIPPOSMS.A	SMSRep-D	SMSRep
13	Lovetrap.A	LUVTRAP.A	MSRep-G	??
14	Kmin.A!mfb	KMIN.SMA	Kmin-C	??
15	Kmin.A!mfb	KMIN.A	Andr/Kmin-D	??
16	Kmin.A!mfb	KMIN.SMA	Kmin-C	??
17	Kmin.A!mfb	KMIN.A	Kmin-D	??
18	DroidKungFu.A	KUNGFU.B	KongFu-A	KongFu
19	DroidKungFu.A	GONFU.A	KongFu-A	KongFu
20	DroidKungFu.A	DROIDKUNGFU.B	KongFu-B	KongFu
21	Zsone.A!mfb	CLICKER.A	AdSMS-B	AdSMS
22	AdSMS.A	ADSMS.A	AdSMS-A	??
23	Fakeplayer.B!mfb	FAKEMOBI.D	FakPlay-C	??
24	Fakeplayer.B!mfb	FAKEP.A	FakPlay-D	??



F igure 6. Combined Behavior Graphs for the Gone family (10)

application for 60 secs., a true 'SMS' type does not indicate that it used the SMS operation because it was not run for sufficiently long. However, we may need to run an application for several hours or to have it run in a specific environment in order to force it to reveal that it may eventually use SMS. This affects numbers 11, 12, 13, 21, 22, 24.

The malicious application named com.Beauty.Girl-1 was found by VirusTotal but was not identified by any of the three AV vendors above as being in any family of Table 1. It is described by Figure 7.



F igure 7. Treemap for com.Beauty.Girl-1

Based on the Droidbox graphs, we would place this sample in the PJAppsfamily as it, along with samples 4, 5 and 6, uses all of service, filewrite, netopen and netwrite. It may well be possible to use our graph approach to identify many other Android applications not yet identified by the AV vendors.

Broadcast Receivers

While executing malicious applications, we observed that almost all of them listen to broadcast receivers such as BOOT_COMPLETE, SEND_MESSAGE, SMS_RECEIVED and OUTGOING_CALL whereas the benign set of applications did not register any such receivers. The Android system sends broadcasts to receivers as announcement for events such as loss of battery power. Both benign and malicious applications can also send broadcast messages as 'intent messages' to the system, for example, indicating that applications are waiting for an event. Broadcast receiver attackers can design their malicious application to listen for incoming messages and forward them to predetermined or premium numbers.

Zeus[31] is an example of such a malicious application; it appeared in late 2011 and aims to steal online banking data by spying on incoming messages. Once the device receives an SMS, the application hiding inside it forwards it to a predetermined number. Zitmo, a variant of Zeus, opens an Internet connection upon capturing an SMS and forwards the message via the network. Figure 8 shows a snapshot of the Zitmo log.

```
[Outgoing traffic]
------------------
[10.0010280609]       Destination: null Port: 0
                      Data: OK

[22.3963680267]       Destination: null Port: 0
                      Data: OK

[Broadcast receivers]
---------------------
.SmsReceiver          Action: android.provider.Telephony.SMS_RECEIVED
```

F igure 8. Snapshot ofZitmo log output

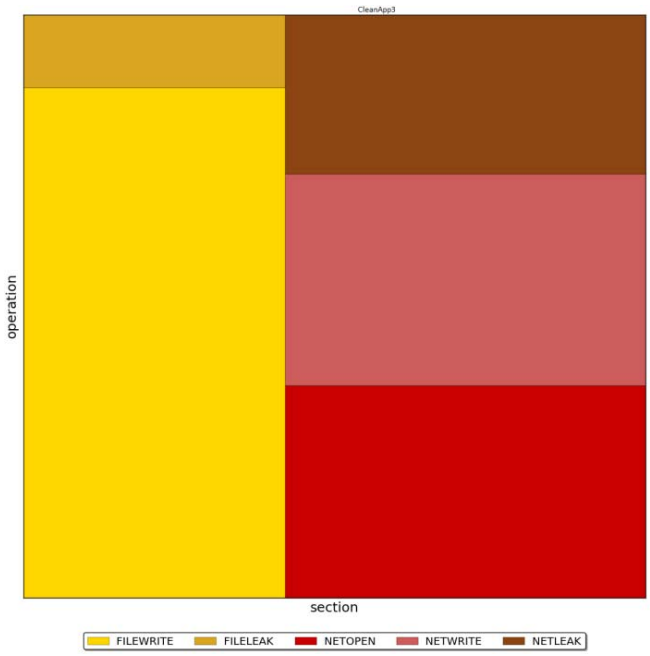
Clean Applications

In Section IIIA, we listed the clean applications that were examined in our experiments. Even though these applications were verified as being benign by VirusTotal, some of them we deemed to be suspicious by virtue of the fact that they were shown to leak information. A summary of the samples with suspicious leaks follows:

CleanApp1: Writes IMEI to config file and leaks two times the IMEI value via the network sink.
CleanApp2: Leaks IMEI and IMSI value in one HTTP GET request.
CleanApp3: Writes IMEI to config file. Leaks IMEI twice in an HTTP GET request to data.mobclix.com and leaks IMEL together with other database values (OTHERDB) to www.umeng.com.
CleanApp3 is the only application in our set that leaks with tag OTHERDB, designating data originating from database URIs that are not common; this data could be anything, even its own application data. However, a clean file should not be calling any logs or stored SMS. **CleanApp1** and **CleanApp3** have very similar treemaps, however they connect to completely

different servers; while they appear to be benign, and they violate privacy rules and misuse permissions. In future work, we will look for and test such clean files more extensively to determine why and how they leak data.

We were able to incorporate the 'leak' operation into the generation of graphs and Figure 9 is a picture of the treemap graph for **CleanApp3** where the netleak is indicated in the top right-hand bar.



F igure 9. Treemap of CleanApp3 showing a net leak.

Comparison with Other Work

To our knowledge, the current paper is the first paper to examine malicious applications behavior using Droidbox. In making the decision to work with Droidbox rather than CWSandbox[32], for example, which was used in [30], we relied on the work of Lantz in [7]. In that thesis, the sandbox approach was transformed from the desktop platform to mobile devices, and while many operations, such as file and network activity, are common, SMS and phone activity as well as use of the DexClassLoader are peculiar to the mobile operating system and the library link operations are slightly different. In addition, Droidbox has the advantage of being an open source project, while we did not have access to the source code for any other suitable sandbox environment.

In [30], Trinius et al. consider a situation similar to ours, but based in a wired computer environment rather than that of mobile devices. Like us, they study techniques to visualize the actions of malware with the aim of both detecting and classifying it. Also like us, they use two means of visualization, one with treemaps and the other with thread graphs; we, on the other hand, use treemaps and behavior graphs. As described in the previous section, treemaps use a combination of colour and area in a rectangle to give a fast visual representation of the type and number of API calls they represent. Thread graphs present the chronological behavior of a sample showing what actions were performed and when. The authors of [30] use a combination of the treemap and thread graph to provide them with a 'behavioural fingerprint' of a sample.

We developed our behavior graph for the purpose of replacing the thread graph, but using non-proprietary software. Rather than visualize each individual thread, we gathered an overview of the behavior. Operations and times were extracted from the Droidboxsandbox logs and plotted using Python Matlab libraries[33]. This gave us access to the same type of visualisation information available to Trinius et al. in[30]. In experimentation, they use a set of 2,000 computer malware samples pre-classified unanimously by six major anti-virus softwares into 13 families. They executed the samples in a sandbox environment for two minutes and used the two types of visualization methods to consider the similarities and differences. In many cases they were able to distinguish between families based on these graphs, and also show that members of the same pre-classified family had similar graphs. They state (Section 5): "The images of different malware samples are visually different, but samples of the same malware are almost identical. Nevertheless, this does not apply for all classes. Since the results of behavior-based analysis depends on several variables which are beyond the analyst's control (e.g., in case the command and control server of a bot is unreachable, the bot will show an entirely different behavior) a perfect clustering seems infeasible." Thus, the authors of [30] come to a conclusion similar to ours, however, based on a much larger dataset.

In considering the differences in data used, we point out that the malware used by Trinius et al. [30] were executables captured by a honeypot in 2009. Thus, they are single files several years in age, and classical anti-virus technology has had the time to agree on correct family labels for them. In the case of the data used in the current paper, we are using twenty-three very recent (all dated 2011) malicious applications (containing several files and folders rather than single executables) which target the mobile phone environment.

VI. Discussion and Future Work

We compared family identification by three major anti-virus vendors with results of Droidbox and were able to find support for the argument that these vendors were in some cases incorrect. We demonstrated that while applications may be classed as benign by these same vendors, some could nevertheless be leaking data and so be prone to easy misuse by malware writers.

We have demonstrated that Droidbox can be a very useful tool both in classifying malicious Android applications and in determining weaknesses in benign Android applications. Some of the limitations of DroidBox are that it only monitors operations performed within the Android framework. Thus, any native code could potentially leak data that goes unnoticed.

Another obstacle is that when running the analysis, some of the malicious behavior is hidden and only triggered on certain events, such as on incoming SMS. Since we do not interact with the sample during analysis, such hidden behavior is not viewed in the results. Additionally, all samples were executed for the same length of time, while malicious activity can be triggered at varying times in different samples. However, these are common problems with all experiments run in a standardized environment.

In our future work, we will examine Android application family classification more extensively and also investigate the implications of data leakages in benign applications. We will also extend DroidBox to provide more detailed API monitoring. Broadcast receivers and their role in identifying malicious applications will also be part of our future investigations.

REFERENCES

[1] Braden. (2011, Dec 1). Google's Android Market Reaches 400,000 Applications. Available: http://www.ijailbreak.com/applications/googles-android-market-reaches-400000-apps/

[2] M. Alazab, S. Venkataraman, and P. Watters, "Towards Understanding Malware Behaviour by the Extraction of API Calls," in *The second Cybercrime and Trustworthy Computing Workshop*, Ballarat, Vic, 2010, pp. 52-59

[3] Ammar Alazab, Moutaz Alazab, Jemal Abawajy, and M. Hobbs, "Web Application Protection against SQL Injection Attack," in *7th conference on Information Technology and Applications (ICITA 2011)*, Sydney, Australia, 2011.

[4] M.Alazab, P.Watters, S.Venkatraman, M.Alazab, and A. Alazab, "Cybercrime: Current Trends of Malware Threats," presented at the International Conference on Democracy, CCIS series from Springer, Thessaloniki, Greece., 2011.

[5] C. Hsiu-Sen and T. Woei-Jiunn, "Mobile Malware Behavioral Analysis and Preventive Strategy Using Ontology," presented at the Conference on Social Computing (SocialCom), 2010.

[6] M. Alazab, S. Venkatraman, P. Watters, and M. Alazab, "Zero-day Malware Detection based on Supervised Learning Algorithms of API call Signatures," in *Ninth Australasian Data Mining Conference: AusDM 2011*, Ballarat, Vic, 2011, pp. 171 - 181.

[7] P. Lantz, "An Android Application Sandbox for Dynamic Analysis," Master, lectrical and Information Technology, Lund university, . Lund, Sweden, 2011.

[8] L. M. Security. (2011, Dec 27). *'Lookout Mobile Threat Report'*. Available: https://www.mylookout.com/_downloads/lookout-mobile-threat-report-2011.pdf

[9] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, Chicago, Illinois, USA, 2011, pp. 3-14.

[10] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh, "Taming Information-Stealing Smartphone Applications (on Android), In Trust and Trustworthy Computing." vol. 6740, J. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, Eds., ed: Springer Berlin / Heidelberg, 2011, pp. 93-107.

[11] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova, "Detection of Malicious Applications on Android OS." vol. 6540, H. Sako, K. Franke, and S. Saitoh, Eds., ed: 'Computational Forensics' Springer Berlin / Heidelberg, 2011, pp. 138-149.

[12] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *proceedings the 18th ACM conference on Computer and communications security*, Chicago, Illinois, USA, 2011, pp. 627-638.

[13] É. Payet and F. Spoto, "Static Analysis of Android Programs Automated Deduction." vol. 6803, N. Bjørner and V. Sofronie-Stokkermans, Eds., ed: Springer Berlin / Heidelberg, 2011, pp. 439-445.

[14] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for Android," in *proceeding the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, Chicago, USA, 2011, pp. 15-26.

[15] Bla, x, T. sing, L. Batyuk, A. D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android Application Sandbox system for suspicious software detection," presented at the 5th International Conference on Malicious and Unwanted Software (MALWARE), Nancy, Lorraine 2010.

[16] Android. (2011, Dec 26). *Android Monkey tool,* . Available: http://developer.android.com/guide/developing/tools/monkey.html,

[17] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi, "XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks," Technical Report, Technische Universit□Darmstadt2011.

[18] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: versatile protection for smartphones," in *'Proceedings of the 26th Annual Computer Security Applications Conference'*, Austin, Texas, 2010, pp. 347-356.

[19] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google Android: A Comprehensive Security Assessment," *Security & Privacy, IEEE,* vol. 8, pp. 35-44, 2010.

[20] Mila. (2011, Nov 1). *Mobile Malware Sample*. Available: http://contagiominidump.blogspot.com/

[21] A. Satoh and T. Inoue, "ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS," *Integration, the VLSI Journal,* vol. 40, pp. 3-10, 2007.

[22] Veelasha Moonsamy, Ronghua Tian, and L. Batten, "Feature Reduction to Speed up Malware Classification," in *the 16th Nordic Conference in Secure IT Systems (NordSec)*, Springer Berlin / Heidelberg, 2012, pp. 176-188.

[23] M. Alazab, A. Alazab, and L. Batten, "Smartphone Malware Based On Synchronization Vulnerabilities," in *proceeding of the 7th conference on Information Technology and Applications (ICITA)*, Sydney, Australia, 2011.

[24] P. lantz. (2011). *Project 5 - DroidBox: An Android Application Sandbox for Dynamic Analysis*. Available: http://www.honeynet.org/gsoc/slot5

[25] William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *proceeding of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada 2010.

[26] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege Escalation Attacks on Android Information Security." vol. 6531, M. Burmester, G. Tsudik, S. Magliveras, and I. Ilic, Eds., ed: Springer Berlin / Heidelberg, 2011, pp. 346-360.

[27] A. Lineberry, D. L. Richardson, and T. Wyatt. (2010, Dec 29). *THESE AREN'T THE PERMISSIONS YOU'RE LOOKING FOR*. Available: http://dtors.files.wordpress.com/2010/09/blackhat-2010-final.pdf

[28] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""Andromaly": a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems,* pp. 1-30, 2011.

[29] B. Shneiderman. (2009, Oct 25). *Treemaps for space-constrained visualization of hierarchies*. Available: http://www.cs.umd.edu/hcil/treemap-history/

[30] P. Trinius, T. Holz, J. Gobel, and F. C. Freiling, "Visual analysis of malware behavior using treemaps and thread graphs," in *prcoeeding of the 6th International Workshop on Visualization for Cyber Security* Atlantic City, NJ 2009, pp. 33-38.

[31] Kaspersky. (2011, Dec 15). *ZeuS-in-the-Mobile for Android.* Available: http://www.securelist.com/en/blog/208193029/ZeuS_in_the_Mobile_for_Android

[32] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security and Privacy,* vol. 5, pp. 32-39, 2007.

[33] John Hunter and D. Dale. (2007). *Matplotlib*. Available: http://matplotlib.sourceforge.net/