Towards an understanding of the impact of advertising on data leaks

Veelasha Moonsamy*, Moutaz Alazab and Lynn Batten

School of Information Technology,
Deakin University,
Australia
Email: v.moonsamy@research.deakin.edu.au
Email: m.alazab@deakin.edu.au
Email: lmbatten@deakin.edu.au
*Corresponding author

Abstract: Recent investigations have determined that many Android applications in both official and non-official online markets expose details of the user's mobile phone without user consent. In this paper, for the first time in the research literature, we provide a full investigation of why such applications leak, how they leak and where the data is leaked to. In order to achieve this, we employ a combination of static and dynamic analysis based on examination of Java classes and application behaviour for a data set of 123 samples, all pre-determined as being free from malicious software. Despite the fact that anti-virus vendor software did not flag any of these samples as malware, approximately 10% of them are shown to leak data about the mobile phone to a third-party; applications from the official market appear to be just as susceptible to such leaks as applications from the non-official markets.

Keywords: android; dynamic; static; data leak; DroidBox; advertising.

Reference to this paper should be made as follows: Moonsamy, V., Alazab, M. and Batten, L. (2012) 'Towards an understanding of the impact of advertising on data leaks', *Int. J. Security and Networks*, Vol. 7, No. 3, pp.181–193.

Biographical notes: Veelasha Moonsamy is currently doing her PhD at Deakin University, Australia. Her research thesis focuses on the security architecture of the Android permission system. She received her Bachelor (Hons) in Information Technology, majoring in IT Security and Mathematical Modelling from Deakin University in 2011. Her research interests include mobile technology, malicious software, machine learning algorithms and security protocols. She is also an associate member of the Australian Computer Society.

Moutaz Alazab is a PhD research student at the Deakin University in the School of Information Technology, Australia, with thesis title 'Prevention and Detection of Mobile Malware'. He received his Bachelor degree (Hons) in Computer Engineering from Al-Balqa Applied University in 2009. He worked as a Systems and Network administrator in Jordan and at the University of Ballarat, Australia. His research interests include mobile malware, computer malware, host and network intrusion detection system, mobile digital forensic, reverse engineer and mobile ad-hoc network. He has published research papers in different well-known international conferences and journals.

Lynn Batten holds the Research Chair in Mathematics and is Director of Information Security Research at Deakin University. She is a Fellow of the Australian Computer Society, a Graduate of the Australian Institute of Company Directors and a Senior Member of the IEEE. Her research interests cover a broad set of area in information security from cryptography to malicious software and digital forensics.

1 Introduction

Android applications can now be downloaded from official and non-official online markets. The sole official market is administrated by Google (Google, 2012a) which regularly tests the applications to make sure they do not contain malicious binaries. In contrast, the applications available on the non-official markets are managed by individuals and businesses and are not checked to determine if they are clean. While downloading applications from the non-official markets therefore represents a known threat to users, the attraction is that they offer unique applications not available elsewhere (Mies, 2010). The numbers of applications available and the numbers downloaded from both markets are increasing at exponential rates (AppBrain, 2010; Panzarino, 2011).

Copyright © 2012 Inderscience Enterprises Ltd.

A major challenge in the development and provision of applications for mobile phones has been the business model (Dhar and Varshney, 2011); however, this now seems to be solved by means of advertising revenue. Google offers an advertisement software development kit (SDK) that allows Android developers to add advertisements into their applications to generate revenue (Apvrille, 2011). Application developers earn revenue from in-application advertisements and are thus encouraged to market their application free of charge; in fact, the more advertising libraries they embed in their applications, the higher the revenue. However, embedded advertisements connect to the operating system level and are difficult to remove once the application is installed. Once an Android user clicks on an advertisement, the system exposes a web browser, and a website might then invoke collection of the international mobile equipment identity (IMEI) code identifying the mobile device; such a website may also invoke the international mobile subscriber identity (IMSI) number found in the SIM card. Researchers (Enck et al., 2010) and (Pearce et al., 2012) have found applications in the applications markets which send the phone identifier and SIM card serial number to developers without the knowledge of the mobile user; the current authors in (Alazab et al., 2012) also discovered applications from the Google market which leak these values.

While we may expect malicious applications to steal or leak identifying device information to third-parties without user permission, it is disturbing that applications classified as non-malicious do so. Thus, the aim of the current paper is to examine a set of applications, all identified as being non-malicious, taken from both the official and non-official markets, and examine them to see if they leak any data identifying the android phone. Financial and games applications are of particular interest to us since they are often used in conjunction with sensitive data and may be a target of attackers trying to steal money or gold. We expect to see attack-resistant development and cautious use of such applications by developers and users respectively; however, we found that approximately 10% of our data set of clean financial and games applications leaked private data about the phone to a third-party without the knowledge of the user.

Throughout the paper, we regularly make use of the phrases ‘malicious application’, ‘clean application’ and ‘leaky application’ and so define them formally here:

- A *Malicious application* is an application specifically designed to harm a computer or the software it is running (Google, 2012b). Such behaviour is usually identified by anti-virus software products.
- A *Clean application* is an application which has not been identified as malicious.
- A *Leaky application* is an application which, unknown to the user, sends data about the mobile hardware to a third-party. A leaky application can be either clean or malicious.

In this paper, we make the following contributions:

- We demonstrate that, without the knowledge of the user, even applications considered to be clean can leak data about the device to a third-party
- For applications in our data set that leak, we determine what data is leaked, how, why, and where it is leaked to.

The paper is organised as follows: in Section 2 we discuss the research literature on Android permissions and the connection between these and advertising libraries and consider the work to date on examination of leaky applications. Section 3 describes DroidBox, an open source dynamic analysis tool designed to analyse Android applications (see Lantz, 2011a)) which we use for dynamic analysis of our data set. Section 4 describes the set of data collected in our experiment. In Section 5, we present the environmental set-up and, in Section 6, we provide a comprehensive analysis of leaky applications. Finally in Section 7, we draw conclusions.

2 Related work

In this section, we present some of the recent work related to Android permissions, to in-application advertising libraries and to data leaks. The reader is referred to Figure 1 in (Android Developers, 2012a) for an excellent illustration of the components of an Android application.

2.1 Android permissions

The Android framework makes use of install-time permissions in order to control access to restricted resources on the smart device. These permissions are generally defined in the AndroidManifest.xml file and require the user to accept them upon application installation.

Felt et al. (2011b) investigate the effectiveness of install-time permission systems in Android applications. They demonstrate that such permissions can be advantageous compared to the traditional user-based permissions system; for example, the install-time permission system ensures that a vulnerable application, present in the host system, will not affect the functionalities of other applications, whilst in the case of a traditional user-based permissions system, all applications are treated equally. However, the authors also found that users are so frequently presented with warning messages about permission requests that they tend to be careless about the use of those permissions once granted. We demonstrate, in our work, that application developers are similarly careless about permissions since they are driven by the revenue earned from in-application advertising.

In other work, Felt et al. (2011a) developed a tool, *Stowaway* to detect over-privileged applications; these are applications which have more permissions than actually required to execute. They manually generate a permission map for the entire Android system which defines the

relationship between an API call and the permission it requires for execution. *Stowaway* includes a static analysis component which takes as input an application, disassembles it and parses through the Java classes to collect the API calls defined in each method. In the next step, the tool then compares the API call with the permission map to check if the correct permission has been assigned. It should be noted that *Stowaway* is an efficient tool to ensure that application developers do not request unnecessary permissions for their applications. However, the tool cannot differentiate between the permissions that can cause data leaks and those required for the execution of the application.

2.2 Advertising libraries

In-application advertising is a major source of revenue for application developers. The service is provided by advertising companies such as Google Ads (Google Ads, 2012) and Admob (Admob, 2012) which distribute pre-packaged advertising libraries to be embedded in existing applications in order to allow the developers to earn revenues. Fundamentally, advertising libraries inherit the same permissions granted to their host applications. Nevertheless, application developers also have the possibility of including additional permissions which can be used to enhance targeted-advertising, thereby generating more revenue. Unfortunately, unnecessary permissions can inadvertently introduce vulnerabilities within the functionalities of the application as for example, causing data to leak through advertisements.

Shekhar et al. (2012) investigate the idea of splitting the host application and advertising into separate processes. This means that the advertising libraries will no longer be able to inherit the same permissions as the host application. It also prevents malicious applications from being able to generate fraudulent clicks in order to steal revenues from application developers. In their framework, the authors also examine the extent to which applications use their permissions only for advertising purposes. The results show that permissions such as internet and READ_PHONE_STATE are widely used only for advertisement; this observation agrees with the results from our experiment.

The work of Pearce et al. (2012) looks into separating the privilege of the host application from the embedded advertising library. In order to effect this, they introduce a new set of API calls and permissions to be used only within advertising libraries. The authors claim that this change in the Android framework will not require application developers to embed advertising libraries anymore; instead, the new set of API calls and permissions will be used to instruct the applications to fetch the advertisements from certain sites. This new framework would eliminate the need for an application to request unnecessary permissions during install-time in order to ensure that advertising libraries are executing properly.

Grace et al. (2012) focus on the potential privacy and security risks posed by advertising libraries. They collected a dataset of 100,000 applications from the official market and manually extracted the AndroidManifest.xml files in order to separate the applications that used the internet permission. Then they manually investigated the set of approximately 52,000 applications with internet permission to determine if the advertising libraries requested dangerous permissions and to establish their impact on the related API calls. In contrast, in our work, we first execute the applications in our dataset and then by examining the log files and AndroidManifest.xml files, we are able to deduce if an application leaked device-related information via advertising libraries.

2.3 Data leaks in android applications

As mentioned in Section 1, we consider a data leak to be an event where an application reads private information and sends it to a third-party without the primary user's consent.

The work by Gibler et al. (2012), Chan et al. (2012), Mann and Starostin (2011), Kim et al. (2012) applied static analysis to identify data leaks in Android applications. Gibler et al. (2012) proposed a framework to detect leaks of personal information. They started by generating a permission map, which included information about API calls and the related permissions they require to execute; the map also contained information about potential sources of leaks. Next, they used the decompiled version of an application to generate a call graph. They iterated repeatedly to cover all possible execution paths of the application and recorded the instances where external methods invoke restricted information; hence identifying potential leaks and their types. After evaluating their work on a set of 23,000 applications which were collected from two different non-official markets, they found 9631 possible privacy leaks in 3258 applications. As for the work of Chan et al. (2012), the authors parsed the AndroidManifest.xml files to collect the requested permissions and then identified components, such as activity and broadcast receiver, that are potential sources of data leaks. They then applied inter-procedural control flow searching for each component and followed the information flow to confirm if the leaks actually occurred.

The work by Mann and Starostin (2011) and Kim et al. (2012) investigated the occurrence of data leaks within the Dalvik bytecode implementation. They began by generating a reduced set of execution instructions in order to capture the relevant information flow paths. Based on the aforementioned set, the authors then manually traversed each application, written in bytecode, to identify potential sources of leaks.

While Chan et al. (2012), Mann and Starostin (2011), Kim et al. (2012) and Desnos (2011) identify leaks, and what is leaked, in this paper, we go further and determine how the leaks occur and the destinations of the leaked data. In our work, we reverse engineer the application package files and statically analyse the Java version of the

applications in order to determine the occurrence of a leak and its possible cause(s). Moreover, we further support our experiment by dynamically executing the applications from our dataset in a sandbox to monitor and record the behaviours between an application and the operating system during run-time, thus identifying the leaked information as well as its destination.

In the next section, we expand further on the tool that was used to perform the dynamic analysis.

3 DroidBox

In our previous paper (Alazab et al., 2012), we determined that some clean applications in fact leaked private data about the Android phone to a third-party without the knowledge of the user. We suspected that this happened because of a vulnerability in the permissions set-up and so in using DroidBox in the current experiment, we focus on analysis of permissions. In this section we describe the features of DroidBox that were used to assist us with our experiment.

DroidBox is an open source dynamic analysis tool and is the extended version of TaintDroid, which was developed by Enck et al. (2010). A thorough description of DroidBox is available in the thesis of Lantz (2011) and also in Alazab et al. (2012) and so we limit ourselves here to only those details which are important for the current paper.

Figure 1 demonstrates the architecture of DroidBox showing the several phases during the analysis of an Android application. First, the sample is located in the windows host as an .apk or .jar extension. Then DroidBox performs static analysis to unzip each sample by using Androguard (Desnos, 2011); DroidBox relies on Monkeyrunner (Android Developers, 2012c) to install and run the application, without any human interaction, by sending random commands and events from an API to the sample. After the sample is installed, the modified system records the requested APIs and then compares each operation with the corresponding manifest file to check whether the application bypasses permissions. DroidBox continues to run and track the sample until the analyser triggers an outcome.

Subsequently, logcat (a logging mechanism provided by Google) is applied for the duration of running the application in the sandbox; the log displays the following: each operation, level, process identification, time, and date in the Android SDK. The DroidBox system then pipelines the log output from an emulator to Python script in order to allow an analyser to read the log file from the host operating system; a script will parse the logcat output to find relevant operations. In the final stage of using DroidBox, and once a log file is created in the host operating system, two graphs are produced – the behaviour graph and the treemap graph; these graphs, together with the DroidBox log assist in interpreting the behaviour of the application.

3.1 The DroidBox log file

As explained by Lantz (2011a), the log file is generated from the sandbox logs collected by logcat at the end of the application execution period. Each log file begins with the following information: the name of the Android package and three types of hash values – MD5, SHA1 and SHA256. The hashes are computed at the end of the execution and they help to ensure that the application code has not been modified during run-time.

The main body of the log file includes details of the following format:

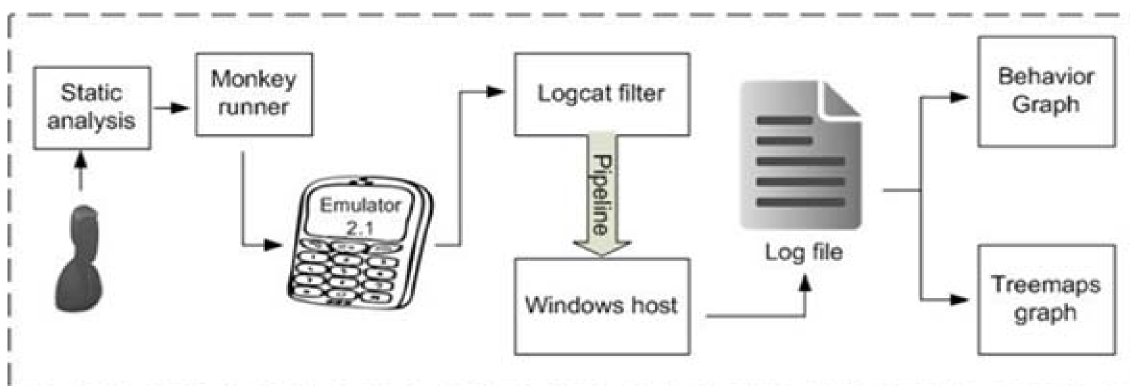
[Section]

[Operation]

[Timestamp] *log data*

where the Section tag captures information related to file operations, cryptographic functions, network-related activities, broadcast receivers, enforced and bypassed permissions, information leaks, sent SMS and phone calls. As for the Operation tag, it relates to the types of operations being performed, such as read from or write to file, encrypt or decrypt data and lastly, open, read or write network activities. The timestamp denotes the time the data was logged and is relative to the starting time of the analysis.

Figure 1 DroidBox architecture



In our current work, we place special focus on the portion of the log file where the information leakage is recorded, as our aim is to better understand how data leaks work. First, we need to introduce the terms taint tag, taint source and taint sink in order to explain how DroidBox can intercept a leak. DroidBox is equipped with a list of pre-determined sources of information (also known as taint sources) such as network and databases that will be monitored during execution in order to intercept information outflow; taint sinks are used to filter and monitor this information. Whenever a piece of information is parsed through the taint source, it is assigned a tag – taint tag – in order to differentiate it from clean data. Any tainted data (that is, those marked with a tag) which pass through a taint sink and are about to leave the device, will trigger an alert, thus informing the emulator about the occurrence of a leak. For a full comprehension of how the taint process operates, and to view the full list of taint tags, we refer the reader to Chapter 3 and Appendix C in Lantz (2011a).

In fact, in our analysis, we do not make use of the graphs provided by DroidBox; however, see Lantz (2011a, Chapter 4) or Alazab et al. (2012) for examples of DroidBox graphs which indicate data leaks.

In the next section, Section 4, we describe the data collection process for our experiment.

4 Data collection

In this section we describe the data collection process, including our rationale for selecting the applications present in our dataset.

In our previous work (Alazab et al., 2012), we found that three popular Games applications, which were pre-classified as clean applications, leaked sensitive device-related information, IMEI and IMSI, to external parties. This led us to raise the question whether there are other clean applications present on the application markets which can expose the data stored on smart devices to outsiders. The Finance and Games categories contain applications which are likely to keep track of personally identifiable information and so, for our experiment, we select applications from these two categories. One might hypothesise that applications from the non-official market would be more likely to leak private data than those from the official market, and in order to test such a hypothesis, we take applications from both markets.

In the subsection below, we elaborate on the dataset collection process.

4.1 The dataset

We collect only the most popular (as defined in (a) below) clean applications which are classified under the Finance and Games category. It should be noted that we do not include (pre-classified) malicious applications in our dataset as one can expect these applications to leak, depending on the extent of the harmful actions they have been instructed to carry out. Hence, they are of no interest to this experiment. Additionally, we only collect those applications which include the internet permission; this decision is made based on the related works (Stevens et al., 2012; Grace et al., 2012; Leontiadis et al., 2012) which demonstrate that the internet permission is highly used in free applications and is also one of the main facilitators of information leaks.

(a) *Official market (known as Google Play)*:

In order to identify the most popular applications available under the Finance and Games categories available in the Google Play market (Google, 2012a), we use a ranking website called Android RunDown (Android RunDown, 2012) which lists by category those applications downloaded more than 250,000 times. Based on those listings, we then search for the top (largest number of downloads) 50 financial and 25 Games applications in the Google Play market.

However, due to the fact that on several occasions malicious applications have been found on the official market (Burguera et al., 2011; Deng, 2011; Kim et al., 2012), we run an additional scan to ensure that the 75 applications are truly clean. This is accomplished by uploading each individual application to an online malware scanner, VirusTotal (VirusTotal, 2011) where 42 anti-virus engines examine it. We found that the following two applications, World Stock Alert Widget (World Stock Alert, 2012) and Super Fishing (Super Fishing, 2012) were classified as malicious. It should be noted that at the time of writing, both applications were still available on the official market for download. The Finance application, World Stock Alert Widget, and Games application, Super Fishing, were detected as malicious by 6 and 7 anti-virus engines respectively. However, the remaining 123 applications were determined to be clean.

The applications chosen under the Finance category are: ApiDemos, aCurrency lite, NYSE STOCKS LIVE WATCH, Stock Quote, Ministocks – Stocks Widget, Google Finance and Bloomberg, MoneySmart Financial Calc, Bloomberg for Smartphone, Money Lover, Daily Money, Droid Wallet, Mortgage Calculator, Loan Calc, Fifth Third Bank, EasyMoney – Expense Manager, Creditscore, Fidelity Investments, Financial Calculators, Cash Droid, Google Finance, ADP HR & Salaris Info, Gesture Tool, Tip N Split Tip Calculator, Higher One Mobile Banking app, TurboTax SnapTax, MyTaxRefund by TurboTax – Free, Tip Calculator by TradeFields, Balance and Budget, ABECU/AECU CU MobileAccess, City National Mobile, 3Rivers Mobile, Pinnacle Financial Partners, Mint.com Personal Finance, My Money Manager, Naver, Portfolio Manager, PNC Mobile, Quote Rocket Insurance, Regions Mobile, Rush Card, Check Book, Mortgage Calculator and Rates, Exchange Rates, Fuel Calculator Mileage Free, GestureBuilder, ING DIRECT, metatrader5 and Office Calculator Free.

The 24 applications chosen under the Games category are: Hidden Ballons, Tiny Station, Whack Your boss, Sudoku, Solitaire, Word search, mobilityware Solitaire, noshufou, Crazy bike Racing Motor, Drage Racing,

Zombies Wave, Amazing Broken Display, copter, GameFly, blackjack, Chess, kayak, Sketchndraw, nyan, coloringbook, fialfree, tourality, plumber and moveitfree.

(b) *Non-official market (SlideMe)*:

As for the non-official market, we use SlideMe (SlideMe, 2008) to download the Finance applications because it allows us to download the Android application package (APK) files without the need of having an Android phone account. We proceed by searching for the applications under the Finance category and then start to collect the ones which include (at least) the internet permission. We apply the same rationale as mentioned at the beginning of subsection 4(i) for consistency. Finally, we upload each application to VirusTotal to confirm if it is clean.

Table 1 Total numbers and types of applications we collected in the data set

	Number of applications
Google Play	49 Finance and 24 Games
SlideME	50 Finance
Total	123 applications

The applications installed from SlideMe under the finance category are: Finance Central, Corporate, Finance News, Footprint, Swift Tip Calculator, Symbol Lookup, Budget Manager, StockLite (V.1.4.9), Technical Stock Charting, mDroid, Stock Analyst Free, InOut, Connector, I Spent Too Much Money, Anvestor, Tip Split, Personal Financial Organiser, Tip Calculator, Pocket Books, Xpense Lite, Stock Observer, Stock Watcher, Budget Helper, Media Budget, Stock Buzz, Mobile Forex, Funky Expense, Moneybag, Cash Droid, American Stock Exchange, Simple Budget, Toshl, Till Budget Free, WorkIt Expenses, Debt Droid, CashLog, Dividend Predictor, My Stock Ticker Lite, Best Deal, Wallet Manager Lite, Housing Loan Calculator, Expense Notes, Tipped Off, Budget Droid, honeyCombWalletManager, Pocket Budget, Go Dutch, Track Moneyand and Pulse.

5 Experimental work

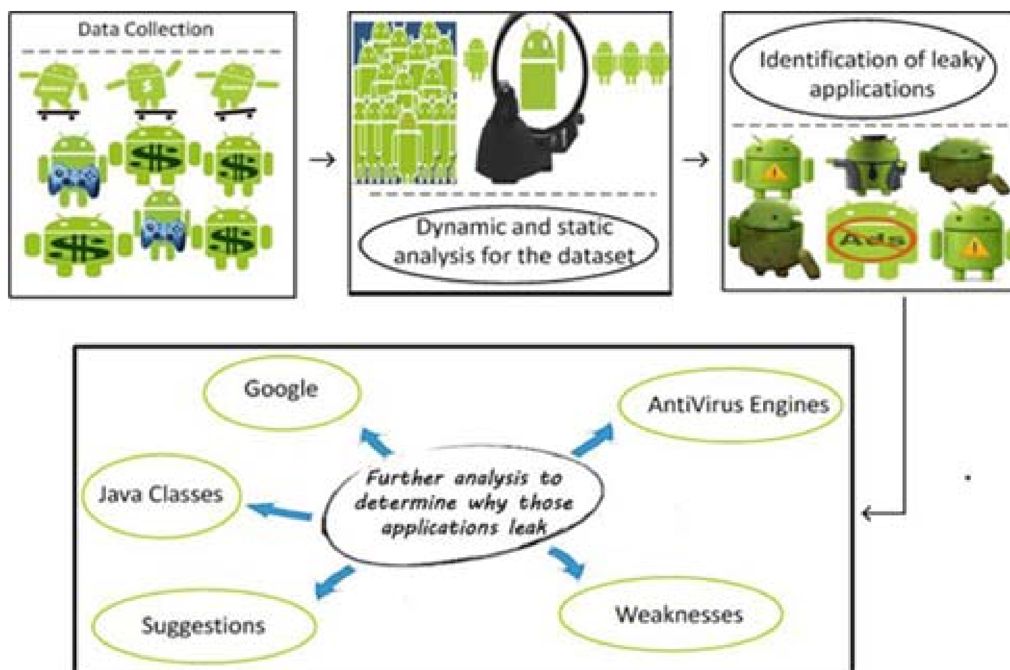
In this section, we describe how we apply the different phases of our methodology on the entire set of 123 applications collected. We begin by performing a dynamic analysis to identify any leaky applications in our dataset.

We then proceed to the static analysis stage where we extract the AndroidManifest.xml files to look at the permissions requested by the applications; the collected permissions are used later to determine their impact on leaky applications. Additionally, we also convert the applications Java (.jar) so that we can examine the advertising libraries embedded in the application code and thus have a better understanding of the effect of those libraries on the general execution of the application. Once the above steps are completed, we conduct a further analysis on the leaky applications to determine the cause of the leak and the destination of the leaked information. The steps are described in Figure 2.

5.1 Set up sandbox environment for dynamic analysis

In this subsection we expand on the set-up of the virtual environment where the dynamic execution of the applications takes place. We also explain the different types of behaviours we record during application run-time.

Figure 2 Overall experimental framework (see online version for colours)



We maintain the same experimental set-up of Alazab et al. (2012) in conducting the dynamic analysis. Only two of the applications chosen for this paper were also examined in Alazab et al. (2012); however, we rerun our entire experiment described in this paper on each application in our current data set. We used a research laptop which is equipped with Intel (i7) CPU 2.7 GHZ, 8 GB of DDR3 RAM and 720 GB hard disk on windows 7. We then install the virtual machine, VMware Workstation build-591240, which runs an Ubuntu 11.10 32bit operating system. Next, we set up the Android Emulator (Andoid Developers, 2012), along with DroidBox (Lantz, 2011b) and disable all interaction between the virtual and local host in order to build a safe environment to run the applications and record the execution process.

Figure 3 illustrates the architecture used to execute the applications. First, we generate the MD5, SHA1 and SHA256 hashes for all applications, using the tool – HashMyFiles (NirSoft, 2007). We choose HashMyFiles since it is able to take more than one application at a time and give the output in a few seconds and thus saves us time. In fact, it also generates the hashes for all of the 123 applications at one time. Matching the hash values with the ones computed by DroidBox after the execution phase is completed is a key step as it ensures that the code has not been modified during execution. The hash values pre- and post-execution were identical; hence no modifications were made to the code during run-time.

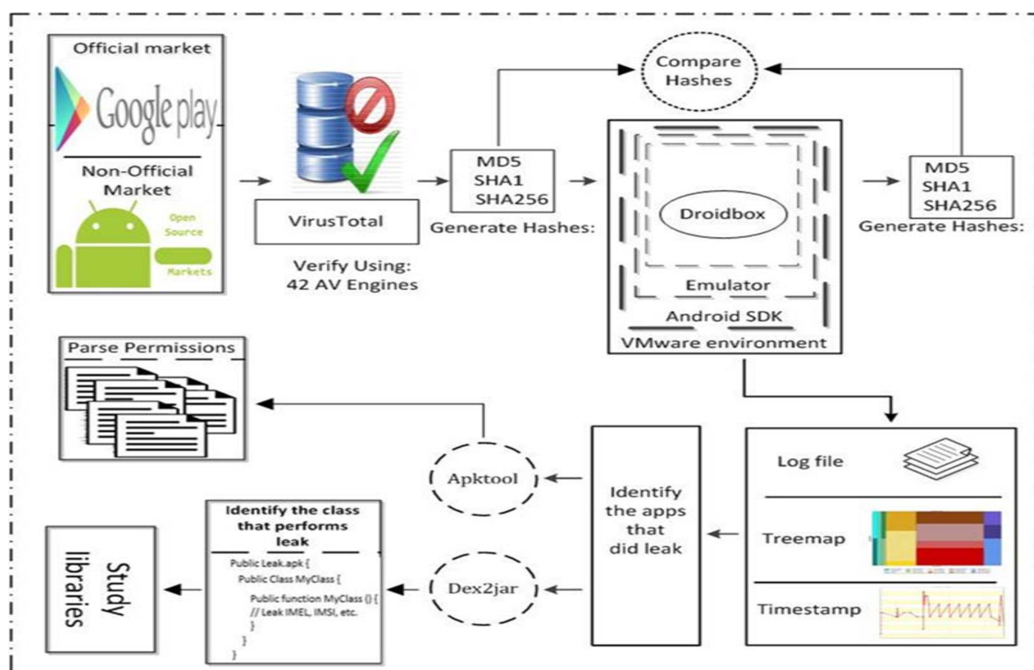
Next, we move to the dynamic analysis and execute our set of 123 applications in the sandbox environment. We had to decide on a cut-off time at which to end the execution process. Other research work showed that 5 minutes (Shih-Yao and Sy-Yen, 2007) or even 1 minute (Bayer et al., 2010) are enough to execute each application and be able to collect sufficient behaviours to conduct further analysis. We therefore opt to run each application for 3 minutes as we believe that, based on the above work, this cut-off time would generate an adequate amount of behaviours for analysis. Nevertheless, restricting the execution time to a fixed period is always problematic in this type of experiment as some applications may need hours or even days to expose all features of the execution; malicious applications are particularly good at avoiding detection by waiting for long periods of time before exhibiting their behaviour. However, all of our applications are clean and so unlikely to be deliberately avoiding detection by suppressing behaviour.

During the execution, DroidBox records several types of system operations in the log files. We are interested in the following as these have been shown to be related to leaks (Isohara et al., 2011): read and write operations, open network connections, enforced and bypassed permissions, information leaks, sending sms and making phone call. We refer the reader to Chapter 4 in Lantz (2011a) for an extensive description of the generic log file generated. Finally, after executing the entire dataset, we import the log files to our local machine and start the static analysis, as explained in the next subsection.

5.2 Perform static analysis on the dataset

In this subsection, we describe our static analysis of the set of 123 applications in order to view the list of permissions requested by extracting the Manifest files. We also decompile the applications to identify whether advertising libraries were embedded in them.

Figure 3 Architecture for detecting leaky applications (see online version for colours)



We begin by extracting the AndroidManifest.xml files using the publicly available tool – Apktool – to identify the permissions requested by each application. We then parse the files, look for the <uses-permission> tag and record the declared permissions. This process helps us to: firstly, ensure that the application includes the internet permission – as this was a pre-requisite defined in the data collection phase (Section 4); and secondly, find out if the application requested additional permissions by analysing the individual AndroidManifest.xml file. It should be noted that Android developers can define their own permissions to protect their applications from being exploited; however, we focus only on the set of permissions defined in the official documentation (Android Developers, 2012d).Moreover, in order to have a better understanding of the in-application advertising libraries, we use the tool – dex2jar – to decompile the application package files into JAR format, which is easily readable by a Java decompiler. We then search for the advertising libraries by browsing for the namespaces. For example: com.google.ads defines the classes required to implement advertisements from Google Ads. Similarly, we identify the various advertising libraries used by the other applications in our dataset, as illustrated in Figure 3. We elaborate on our experimental results below.

Permissions:

While the internet permission was legitimately requested for most applications in our dataset, we observed that 6 applications which were collected from the official market bypassed it. Moreover, although these 6 applications did not explicitly request the internet permission during install-time, all of them had the READ_PHONE_STATE permission defined in their AndroidManifest.xml file. This leads us to suspect that the aforementioned permission together with the TelephoneManager class (provided in the Android official documentation (Android Developers, 2012b)) could eventually result in leaking device-related information such as the IMEI and IMSI.

Advertising Libraries:

During the static analysis phase, we examined the decompiled version of the APKs. We found that, as illustrated in Figure 4, Google Ads was the most popular advertising library implemented by most Android developers. Eighty percent of the applications in our data set included one or more advertising libraries and all the advertising libraries in our dataset used the internet permission to communicate with the advertisers.

5.3 *Identify leaky applications*

In order to determine if an application leaked during the dynamic analysis, we parse the log files and search for the section where the leak is recorded, as shown in Figure 5. DroidBox keeps track of five types of information when documenting a leak in the log file. These include the source of the leak (also referred to as 'sink'), the destination of the leak, the port number through which the information is sent to an external party, the name of the taint tag and the html-encoded data which is leaked through the GET command.

Therefore, it can be seen from Figure 5 that the application leaked the IMSI to an external server with address 'intuitandroidtaxstatprod.122.2o7.net' via the network sink. Similarly, we browsed through the 123 log files which were collected from the dynamic analysis (Section 5.1) and searched for the information leakage section to identify the presence of leaky applications. In the end, we found a total of 13 applications which leaked device-related information, such as IMEI and IMSI, to external networks.

Figure 4 The distribution of advertising libraries for 123 applications (see online version for colours)

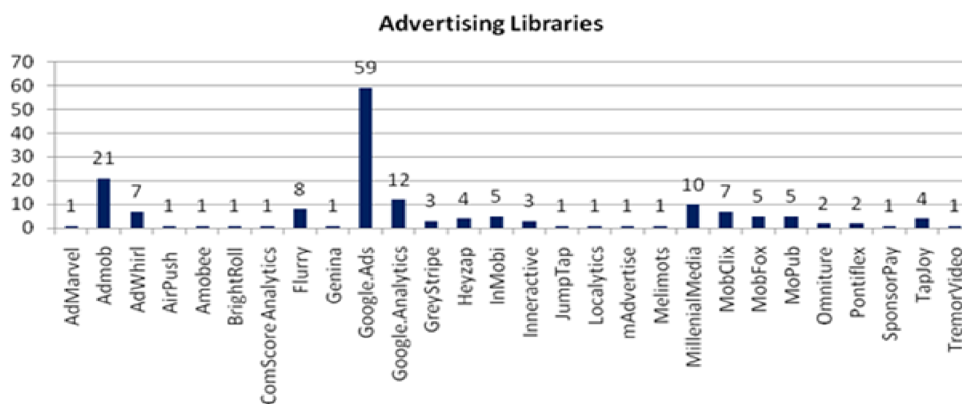


Figure 5 Extract of a log file, showing the section about information leakage

```
[1;48m[Information leakage]
---------------------
      [1;36m31.3967950344]          Sink: Network
                                    Destination: intuitandroidtaxstatprod.122.2o7.net
                                    Port: 80
                                    Tag: TAINT_IMSI
                                    Data: GET /b/ss/intuitandroidtaxstatprod/0/JAN-1.0/s901348?AQB=1&ndh
```

5.4 Determine the cause(s) of leaks in applications

After confirming that 13 out of the 123 applications from our dataset leak information to external parties, we proceed to conduct further analysis of the AndroidManifest.xml files and decompile APKs to have a better understanding of the events that triggered the leaks.

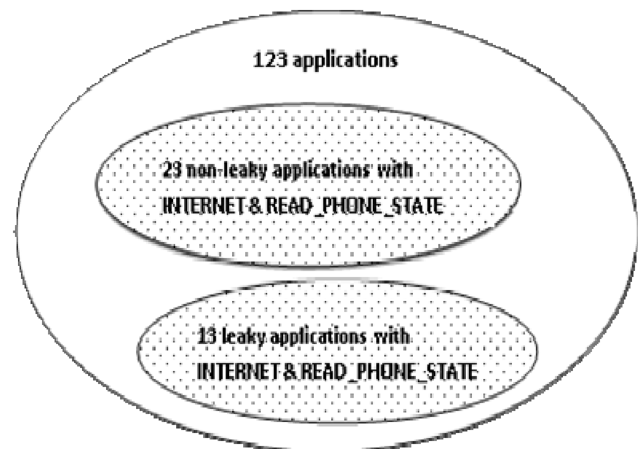
We begin by examining the individual Android Manifest.xml files for each leaky application to find out the list of permissions that were requested. Table 2 shows the distribution of permissions across the set of 13 applications under consideration; 'X' denotes a request that was made for a particular permission. It should be noted, at firsthand, that all 13 leaky applications include both the internet and READ_PHONE_STATE permissions. A combination of these two permissions is sufficient to render an application vulnerable, as noted in (Stevens et al., 2012).

Table 2 Permissions requested by the 13 leaky applications

Permission / *Application Name*	ACCESS_COARSE_LOCATION	ACCESS_FINE_STATE	ACCESS_NETWORK_STATE	ACCESS_WIFI_STATE	INTERNET	GLOBAL_SEARCH	READ_PHONE_STATE	SYSTEM_ALERT_WINDOW	VIBRATE	WAKE_LOCK	WRITE_EXTERNAL_STORAGE
y Tax Refund			X		X		X				
iStock Manager			X		X		X				
Sudoku					X		X			X	X
Solitaire			X		X		X		X		X
Crazy Bike Racing Motor			X		X		X	X			X
Drage Racing			X		X		X		X		
Sketch n Draw					X		X				X
Connector	X		X		X		X				
Finance Central			X	X	X		X				X
Mint.com Personal Finance			X		X	X	X				
Stock Analyst Free	X	X	X	X	X		X				
Symbol Lookup	X	X	X	X	X		X				
Xpense Lite			X		X		X				X

Additionally, it should be noted that out of the 123 applications, only 36 applications (with advertising libraries) included both the internet and READ_PHONE_STATE permissions. However, only 13 applications leaked either the IMSI or IMEI information to advertising networks. We believe that this is the case because the 3 minute cut- off that we used as execution time did not permit the remaining 23 applications to execute the code that could potentially lead to leaks; thus DroidBox was unable to record any information leaks (see Figure 6 for a break-down of the numbers).

Figure 6 Applications with INTERNET and READ_PHONE_STATE permissions



Next, we analyse the classes defined within the decompiled version of each application, generated in the static analysis stage (Section 5.2). We use the destination addresses, recorded under the information leakage section within the log files, to search through the Java classes and compare the namespace to confirm if the application developer has made use of any external in-application libraries. This led to us to observe that all the 13 applications had third-party advertising libraries embedded in their respective Java packages.

Furthermore, the design of the Android security framework does not permit the operating system to differentiate between the permissions required by an application and those needed by an advertising library (Shekhar et al., 2012; Pearce et al., 2012). Consequently, any in-application advertising library will, by default, inherit the same permissions requested for an application and eventually accelerate the occurrence of information leaks.

6 Analysis of leaky applications

In this section, we present an in-depth analysis of the 13 leaky applications.

6.1 Description of leaky applications

We give in Tables 3 and 4 the full description of the leak, including the name and size of the application, the leaked information and its destination. We found 7 and 6 leaky applications from Google Play and SlideME respectively. We observed in Section 5.4 that the cause of information leakage can be attributed to the use of in-application advertising libraries and the inheritance of permissions requested by host applications.

Table 3 Leaky applications found in the official market (7)

	Application name	Detected leak	Leak destination
Finance	My Tax Refund	IMSI (3 times)	• intuitandroidtaxstatprod. 122.2o7.net (hosted by *Omniture*)
Finance	iStock Manager	IMEI (4 times)	• my.mobfox.com (hosted by *MobFox*)
Game	Sudoku	IMEI (5 times)	• ads2.greystripe.com (hosted by *GreyStripe*) • ads.mp.mydas.mobi (hosted by *Millennial Media*) • androidsdk.ads.mp.mydas. mobi (leaks 3 times) (hosted by *Millennial Media*)
Game	Solitaire	IMEI (3 times)	• ads2.greystripe.com (hosted by *GreyStripe*) • service.sponsorpay.com (leaks twice) (hosted by *SponsorPay*)
Game	Crazy Bike Racing Motor	IMEI (2 times)	• www.umeng.com (hosted by *Umeng*)
Game	Drage Racing	IMEI (once)	• data.flurry.com (hosted by *Flurry*)
Game	Sketch n Draw	IMEI (3 times)	• ads.mp.mydas.mobi (hosted by *Millennial Media*) • androidsdk.ads.mp.mydas. mobi (leaks twice) (hosted by *Millennial Media*)

6.2 *Investigation of leaky applications*

Upon further analysis, we noticed that 9 out of those 13 applications included on average two additional third-party advertising libraries, excluding the one through which they leaked. Moreover, we also found a total of 9 advertising libraries embedded in one leaky application which is classified under the Games category and at most 3 advertising libraries included, at one time, in a leaky application from the Finance category.

Generally, it is well-known that application developers earn revenues from in-application advertisements and hence, providing them with the incentive to market their application free of charge. In fact, the more advertising libraries they embed in their applications, the higher the revenue. It is also worth mentioning that each advertiser has their own set of advertising libraries which can be obtained after signing up with the advertising company. Application developers do not have to fully comprehend the advertising code as they only need to follow the instructions given by the advertising companies to successfully include advertisements in their applications; therefore unknowingly leaking sensitive information through the advertising libraries.

Table 4 Leaky applications found in the non-official market (6)

	Application name	Detected leak	Leak destination
Finance	Connector	IMEI (5 times)	• wv.inner-active.mobi (hosted by *Inneractive*)
Finance	Finance Central	IMEI (7 times)	• ads.mp.mydas.mobi (hosted by *Millennial Media*) • androidsdk.ads.mp.mydas.m obi (leaks 6 times) (hosted by *Millennial Media*)
Finance	Mint.com Personal Finance	IMSI (once)	• ci.intuit.com (hosted by *Omniture*)
Finance	Stock Analyst Free	IMEI (twice)	• my.mobfox.com (hosted by *MobFox*)
Finance	Symbol Lookup	IMEI (twice)	• my.mobfox.com (hosted by *MobFox*)
Finance	XpenseLite	IMEI (6 times)	• ads.mp.mydas.mobi (hosted by *Millennial Media*) • androidsdk.ads.mp.mydas.m obi (leaks 5 times) (hosted by *Millennial Media*)

The following steps demonstrate, through an examination of the application Mint.com Personal Finance how an embedded advertising library extracts the IMSI information and sends it to the advertising server:

Step 1:

We look for the main entry point of the application in its AndroidManifest.xml file. This information is stored within the Activity component, with the description *<android. intent.action.MAIN>*. In this case, the *.activity.MintLogin Activity* is the main activity that starts the application (see Appendix A).

Step 2:

Next, we search for the Java class named '*MintLogin Activity*' in the decompiled version of the application package file. We notice that after the user has successfully logged in, the application invokes the class '*MintOmniture TrackingUtility*' (see Appendix A).

Step 3:

The '*MintOmnitureTrackingUtility'* class subsequently invokes the '*AppMeasurement'* class which contains a method, '*getDefaultVisitorID()*' whose purpose is to extract the IMSI using the '*getSubscriberId()*' method (see Appendix A).

Step 4:

The advertising code also included the method 'send' which allowed the application to leak the IMSI via the network (see Appendix A).

Step 5:

Below is an excerpt from the DroidBox log file generated during the dynamic execution of the application. It can be seen that the application did indeed leak the IMSI via Port 80 – which is used for HTTP communications (see Appendix A).

Advertising libraries, by default, require the application developer to include the internet permission in the AndroidManifest.xml file so that the advertising company can track the number of clicks which will then be used to determine the revenue to be paid to the application developer. This leads us to conclude that the application developers follow do not follow the least-privilege method when requesting permissions for their applications.

Furthermore, we also noted that 10 leaky applications made use of the WebView class and APIs to embed the in-application advertisements. WebView can be regarded as an in-application browser and enhances the advertising display to allow the application to present web content in the advertisements. The downfall of this type of advertisement implementation is that it offers a gateway to external parties through the WebView browser to initiate an attack and eventually take control of the device, as explained in (Luo et al., 2011). Additionally, in order for the WebView components to function correctly, the application must request the internet permission during the installation thus, opening a pathway for possible web attacks that can be routed through WebView browser to the application.

In Section 7, we conclude our work and present some ideas for future research.

7 Conclusion and future work

We have demonstrated that, without the knowledge of the user, applications considered to be clean can leak data about the device to a third-party. Our analysis of those applications which leak, described how, why and where information is leaked to.

Clean applications, irrespective of whether they are collected from official or non-official markets, are capable of leaking phone-related information without the users' knowledge. We also observed that third-party advertising libraries were the principal cause of all the leaks that were recorded for our dataset. In order to obtain revenue, embedded advertising libraries often require certain permissions not always available in the application, and so such applications must be made over-privileged in order for the advertiser to obtain revenue; this usage of unnecessary permissions facilitates the occurrence of leaks. Advertisers use the device information to track sites viewed by the user in order to get a picture of what goods and services may be of interest to the user.

7.1 Recommendations

Some device users may indeed be happy to have advertisers track their location in order to be provided with a customised advertising profile and targeted advertising. On the other hand, many users feel that leakage of device ID data is an invasion of privacy (and in some cases may even be illegal) and do not want their location tracked. We believe that the user should have the option. We recommend that every application be equipped with the functionality of permitting the device user to use a downloaded application for a fixed trial period (as determined from the date of the download) and at the completion of this period, be asked to opt in to location tracking, with the option of not doing so. We also recommend that anti-virus software developers include in their products identification of applications containing advertising libraries, or, since this is likely to be by far the majority of applications, those which do not; this flags the presence of advertising libraries to the user who may later make a decision to opt out of location tracking. In making these recommendations, we note that the authors of (Pearce et al., 2012) have also made recommendations for changes in the Android framework to address the issue of private data leaks. We believe that our solution would be easier and cheaper to implement than theirs.

7.2 Limitations of our work

Our restricted data set was of course a limitation, both in the number of applications it contained and in the fact that only Financial and Games applications were chosen. However, as argued in Section 4, these applications were chosen for specific reasons. In further work, we plan to investigate larger and more varied data sets.

In dynamic analysis, restricting the execution time to a fixed three minute period is problematic as some applications may need much more time to expose all features of the execution. We noted in Section 5 that malicious applications are particularly good at avoiding detection by waiting for long periods of time before exhibiting their behaviour. In our case, all applications executed were clean and it is in the interest of advertisers to make an internet connection quickly. We circumvented this limitation by conducting a thorough static analysis on each potentially leaky application, as identified in Figure 6.

References

Admob (2012) *Monetize and promote your apps with ads*. Available online at: http://www.google.com/ads/admob/ (accessed on May 2012).

Alazab, M., Lantz, P., Moonsamy, V., Batten, L. and Tian, R. (2012) 'Analysis of malicious and benign Android applications', *Proceedings of the International Conference Distributed Computing Systems (ICDCS 2012)*, Macau, China.

Android Developers (2012a) '*Building and Running*. Available online at: http://developer.android.com/guide/developing/building/index.html (accessed on May 2012).

Android Developers (2012b) *Download Android Software Development Kit*. Available online at: http://developer.android.com/sdk/index.html (accessed on May 2012).

Android Developers (2012c) *MonkeyRunner*. Available online at: http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html (accessed on May 2012).

Android Developers (2012d) *Reference – Manifest.permission*. Available online at: http://developer.android.com/reference/android/Manifest.permission.html (accessed on May 2012).

Android Rundown (2012) *Android Application and hardware news and reviews*. Available online at: http://www.androidrundown.com/top-apps/all-android-finance/ (accessed on May 2012).

AppBrain (2010) *Number of available Android applications*. Available online at: http://www.appbrain.com/stats/number-of-android-apps (accessed on May 2012).

Apvrille, A. (2011) 'Airpush...pushes the envelope', *FortiBlog: Reports from the Threat Landscape*, retrieved May 2012.

Bayer, H., Kirda, E. and Kruegel, C. (2010) 'Improving the efficiency of dynamic malware analysis', *Proceedings of the 2010 ACM Symposium on Applied Computing*, Sierre, Switzerland, pp.1871–1878.

Burguera, I., Zurutuza, U. and Nadjm-Tehrani, S. (2011) 'Crowdroid: Behavior-Based Malware Detection System for Android', *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM).*

Chan, P., Hui, L. and Yiu, S. (2012) 'DroidChecker: analyzing android applications for capability leak', *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, Tucson, Arizona, USA, pp.125–136.

Deng, Z. (2011) *Design and implementation of an advanced events logging framework for Android*, Masters Thesis, Australia National University.

Desnos, A. (2011) *Androguard -Reverse Engineering Tool*. Available online at: http://code.google.com/p/androguard/ (accessed on May 2012).

Dhar, S. and Varshney, U. (2011) 'Challenges and business models for mobile location-based services and advertising', *Communications of the ACM*, Vol. 54, No. 5, pp.121–128.

Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P. and Sheth, A.N. (2010) 'TaintDroid: an information-flow tracking system for real-time privacy monitoring on smartphones', *Proceedings of the 9th USENIX conference on Operating systems design and implementation*.

Felt, A.P., Chin, E., Hanna, S., Song, D. and Wagner, D. (2011a) 'Android Permissions Demystified', *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ACM.

Felt, A.P., Greenwood, K. and Wagner, D. (2011b) 'The effectiveness of install-time permission systems for third-party applications', *2nd USENIX Conference on Web Application Development (WebApps11)*, Portalnd, OR.

Gibler, C., Crusell, J., Erickson, J. and Chen, H. (2012) 'AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale', *5th International Conference on Trust & Trustworthy Computing (TRUST)*, Vienna, Austria.

Google (2012a) *Google Play – Official Android Application Market*. Available online at: https://play.google.com/ (accessed on May 2012).

Google (2012b) *Safety and Security – Malware*. Available online at: http://support.google.com/adwordspolicy/bin/answer.py?hl=en&answer=1308246&topic=1310876&ctx=topic&path=1308243-1308243-2585946 (accessed on May 2012).

Google Ads (2012) *Advertise on Google*. Available online at: http://www.google.com/intl/en/ads/ (accessed on May 2012).

Grace, M.C., Zhou, W., Jiang, X. and Sadeghi, A. (2012) 'Unsafe exposure analysis of mobile in-app advertisements', *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, ACM.

Isohara, T., Akemori, K. and Kubota, A. (2011) 'Kernel-based Behavior Analysis for Android Malware Detection', *7th International Conference on Computational Intelligence and Security (CIS)*.

Kim, J., Yoon, Y. and Yi, K. (2012) 'ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications', *Mobile Security Technologies (Most), IEEE Symposium on Security and Privacy*, San Francisco, CA.

Lantz, P. (2011a) *An Android Application Sandbox for Dynamic Analysis*, Masters Thesis, Department of Electrical and Information Technology, Sweden, Lund University.

Lantz, P. (2011b) *The Honeynet Project*. Available online at: http://www.honeynet.org/gsoc/slot5

Leontiadis, I., Efstratiou, C., Picone, M. and Mascolo, C. (2012). 'Don't kill my ads! Balancing Privacy in an Ad-Supported Mobile Application Market', *The 13th International Workshop on Mobile Computing Systems and Applications*, San Diego, California.

Luo, T., Hao, H., Du, W., Wang, Y. and Yin, H. (2011) 'Attacks on WebView in the Android system', *Proceedings of the 27th Annual Computer Security Applications Conference*, Orlando, Florida, pp.343–352.

Mann, C. and Starostin, A. (2011) 'A Framework for Static Detection of Privacy Leaks in Android Applications', *Proceedings of 27th Symposium on Applied Computing (SAC)*, ACM.

Mies, G. (2010) 'Third-Party App Stores: Worth the Trouble?', *PCWorld*. Available online at: http://www.pcworld.com/article/210624/thirdparty_app_stores_worth_the_trouble.html (accessed on May 2012).

NirSoft (2007) *HashMyFiles: Calculate hash of files*. Available online at: http://www.nirsoft.net/utils/hash_my_files.html (accessed on May 2012).

Panzarino, M. (2011) 'Android Market hits 10B apps download, now at 53 apps per device, 10c app sale to celebrate', *The Next Web*. Available online at: http://thenextweb.com/google/2011/12/06/android-market-hits-10b-apps-downloaded-now-at-1b-a-month-10c-app-sale-to-celebrate/ (accessed on May 2012).

Pearce, P., Felt, A.P., Nunez, G. and Wagner, D. (2012) 'AdDroid: Privilege Separation for Applications and Advertisers in Android', *Proceedings of AsiaCCS.*

Shekhar, S., Dietz, M. and Wallach, D.S. (2012) *AdSplit: Separating smartphone advertising from applications*, Arxiv preprint arXiv:1202.4030.

Shih-Yao, D. and Sy-Yen, K. (2007) 'MAPMon: A Host-Based Malware Detection Tool', *13th Pacific Rim International Symposium on Dependable Computing, PRDC 2007.*

SlideME (2008) *SlideME – Your MarketPlace for Android Applications*. Available online at: http://www.slideme.org (accessed on May 2012).

Stevens, R., Gibler, C., Crussell, J., Erickson, J. and Chen, H. (2012) 'Investigating User Privacy in Android Ad Libraries', *IEEE Mobile Security Technologies (MoST).*

Super Fishing (2012) *Google Play - Official Android Application Market*. Available online at: https://play.google.com/store/apps/details?id=com.AFTDMedia.superfishing1&hl=en (accessed on May 2012).

VirusTotal (2011) *Free Online Virus, Malware and URL Scanner*. Available online at: https://www.virustotal.com/ (accessed on May 2012).

World Stock Alert Widget (2012) *Google Play – Official Android Application Market*. Available online at: https://play.google.com/store/apps/details?id=fr.aperto.android.worldstockalert&hl=en (accessed on May 2012).

Appendix A

Step 1

```
<activity android:label="@string/app_name" android:name=".activity.MintLoginActivity" android:launchMode="singleTask">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Step 2

```
MintLoginActivity.class

public void onCreate(Bundle paramBundle)
{
  super.onCreate(paramBundle);
  this.measureView = MintOmnitureTrackingUtility.initializeAppMeasurement(this, "login");
  requestWindowFeature(5);
  this. binderDelegate.setNoTitle();
```

Step 3

```
AppMeasurement.class

private String getDefaultVisitorID()
{
  try
  {
    String str2 = ((TelephonyManager)this.application.getSystemService("phone")).getSubscriberId();
    str1 = str2;
    return str1;
  }
  catch (Exception localException)
  {
    while (true)
      String str1 = "000000000000000";
  }
}
```

Step 4

```
public void send()
{
  try
  {
    HttpURLConnection localHttpURLConnection = openConnection(this.requestString);
    localHttpURLConnection.setConnectTimeout(5000);
    localHttpURLConnection.setReadTimeout(5000);
    localHttpURLConnection.setRequestProperty("User-Agent", this.userAgent);
    localHttpURLConnection.setRequestProperty("Accept-Language", this.localeID);
    localHttpURLConnection.getResponseCode();
    label48: return;
  }
```

Step 5

```
[1;48m[Information leakage]
---------------------
      [1;36m34.7022740841          Sink: Network
                                   Destination: ci.intuit.com
                                   Port: 80
                                   Tag: TAINT_IMSI
                                   Data: GET /b/ss/intuitmintandroidprod/0/JAN-1.0/s45991983?AQB=1&ndh
```