Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Exfiltrating data from Android devices

*Quang Do, Ben Martini, Kim-Kwang Raymond Choo**

Information Assurance Research Group, University of South Australia, GPO Box 2471, Adelaide, SA 5001, Australia

ARTICLE INFO

Article history:
Received 24 July 2014
Received in revised form
11 September 2014
Accepted 27 October 2014
Available online 6 November 2014

Keywords:
Android
Code injection
Covert exfiltration
Data exfiltration
Inaudible transmission
Mobile adversary model
Reverse engineering
SMALI
SMS transmission

ABSTRACT

Modern mobile devices have security capabilities built into the native operating system, which are generally designed to ensure the security of personal or corporate data stored on the device, both at rest and in transit. In recent times, there has been interest from researchers and governments in securing as well as exfiltrating data stored on such devices (e.g. the high profile PRISM program involving the US Government). In this paper, we propose an adversary model for Android covert data exfiltration, and demonstrate how it can be used to construct a mobile data exfiltration technique (MDET) to covertly exfiltrate data from Android devices. Two proof-of-concepts were implemented to demonstrate the feasibility of exfiltrating data via SMS and inaudible audio transmission using standard mobile devices.

1. Introduction

Mobile devices and apps are an important tool for accessing information when desktop computers are unavailable. For example, a study of 4125 mobile device users in 2011 found that an average mobile user spent approximately 59.23 min per day on their mobile devices, and the average app session is approximately 71.56 s (Böhmer et al., 2011), and a report by Gartner (2013) forecasts that by 2017, approximately 86% of devices shipped worldwide will be running one of the four major mobile operating systems, namely Android, iOS, Windows Phone and BlackBerry.

Due to the ability of mobile devices and apps to access and store personally identifiable and sensitive information (e.g. geolocation information), they present a genuine security and privacy threat to their users. Gartner (2013), for example, predicts that "[t]hrough 2017, 75% of mobile security breaches will be the result of mobile application misconfigurations. By 2017, the focus of mobile breaches will shift to tablets and smartphones from workstations. Through 2015, more than 75% of mobile applications will fail basic security tests".

To secure the Android operating system (OS) and enhance user privacy, researchers have converged upon several avenues. Firstly and most commonly are systems designed by either modifying the Android source code or as apps to monitor the system externally. Taming Information-Stealing Smartphone Applications (TISSA) is a system developed by Zhou et al. (2011) that gives the user fine-grained control over

* *Corresponding author*.
E-mail addresses: quang.do@mymail.unisa.edu.au (Q. Do), ben.martini@unisa.edu.au (B. Martini), raymond.choo@unisa.edu.au (K.-K.R. Choo).
http://dx.doi.org/10.1016/j.cose.2014.10.016
0167-4048/

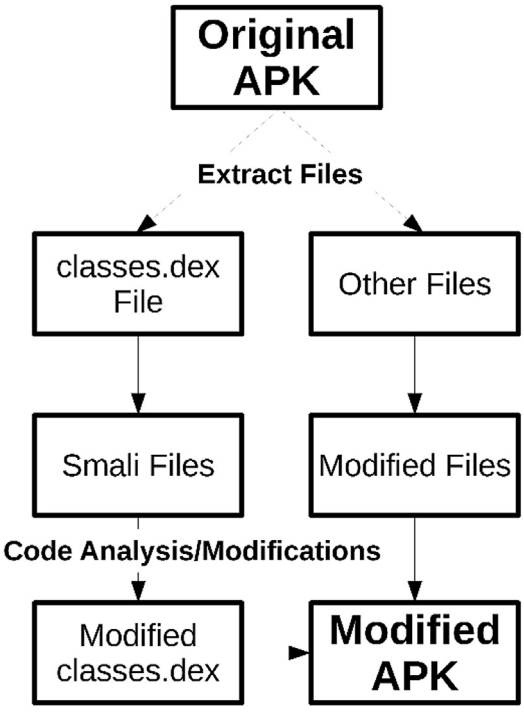


Fig. 1 – Android app reverse engineering.

what information and resources apps on the device can access. Similarly, Hornyack et al. (2011) propose a system called AppFence, which implements a series of Android OS modifications that aim to enhance user privacy. This is achieved through the use of shadow data. Instead of sending sensitive user data when an app requests it, the data is replaced with empty or faked versions. PermissionTracker (Kern and Sametinger, 2012) is a combination of some Android OS code changes and a companion app designed to also allow fine-gained app resource request control. The authors conclude that without performing changes to the Android OS base code, the functionality of these systems would be severely limited. These systems are able to be very deeply integrated into the system due to their nature of being source code additions or modifications but their potential for mainstream usage is extremely low (unless adopted by Google in future versions of Android). Ideally for a privacy enhancing method to be accessible to a wide range of users, it should be implementable on Android devices without modifying (or even affecting) the base OS code.

Another commonly used method for enhancing user privacy, that has a larger potential user base, is Android app analysis wherein an app is decompiled and analyzed and/or modified to determine whether it has malicious intent. The Android apps under analysis are decompiled into a language called SMALI, which is a direct representation of the .dex format used by the Dalvik virtual machine. SMALI is a human readable language in comparison to the language used by the Dalvik virtual machine. This SMALI code can then be analyzed or modified before being assembled into a Dalvik executable ".dex" file. Appropriate changes are made to the rest of the app's files before the file is repackaged into an APK. This process is known as Android app reverse engineering – see Fig. 1.

Examples of Android app analysis using reverse engineering include DroidAnalytics (Zheng et al., 2013), which decompiles an app and uses a combination of information from the manifest file and SMALI code to detect apps that have been repackaged with malware. DroidMOSS (Zhou et al., 2012) is another system that utilizes app disassembly to detect repackaged Android apps on alternative app stores (i.e. app stores other than Google Play Store). DroidMat (Wu et al., 2012) is yet another malware analysis system that uses API calls in its algorithms for malware detection. Aurasium (Xu et al., 2012) is a technology that utilizes reverse engineering to enhance user security and privacy. Rather than just analyzing the APK file for malware, Aurasium injects a monitoring module into the app that sends information to the OS such as when an app requests access to the device's phone number and whether to allow this request. All of these examples are accomplished via repackaging on an external computer, thus reducing accessibility. AppGuard (Backes et al., 2013) is a standalone app that is able to repackage apps on the phone whilst simultaneously monitoring and detecting anomalies or potential threats within.

Reverse engineering of apps has been used by researchers to detect apps containing malware. The most widely used method to detect malicious activities was to determine whether an app was communicating sensitive information back to its servers (or waiting to receive remote commands from certain servers) via the Internet (Apvrille and Strazzere, 2012; Jung et al., 2013; Zhou et al., 2012).

Existing research on covert data exfiltration is generally designed to exfiltrate sensitive data from traditional systems (e.g. desktop computers and laptops). For example, the BadBios malware is rumored to be able to exfiltrate data from desktop computers using high-pitched sounds inaudible to the human ear (Goodin, 2013). This is a controversial topic among security professionals with many believing BadBios is not real. Two more recent articles by Marks (2014) and Sanger and Shanker (2014) alleged that the National Security Agency implants tiny radio transmitters into target computers to exfiltrate data, even when that computer is not connected to the Internet. However, such capability is beyond most actors including the majority of the state actors and it is significantly easier for a user to unwittingly install an application than to install additional hardware. In addition, adding tiny radio transmitters or other physical hardware will leave behind physical evidence.

1.1. *Contributions*

In this paper, we propose an adversary model for Android covert data exfiltration, which we use to build a mobile data exfiltration technique (MDET). We then demonstrate how sensitive data can be obtained from Android devices in a covert manner using communication mediums found on almost all mobile devices. As a case study, we create two proof-of-concept apps which use SMS and audio to exfiltrate data from the test Android device. This is, to the best of our knowledge, the first published work on using inaudible

sounds to exfiltrate data from mobile devices with consumer grade equipment.

1.2. *Roadmap*

The rest of the paper is organized as follows. The next section introduces the Android OS. We present the adversary model, MDET, and the potential data exfiltration mediums in Section 3. We then present our proof-of-concept mobile data exfiltration apps in Section 4, and discuss the findings in Section 5. The last section concludes this paper.

2. Background: Android

The Android OS is an open source OS that relies on a permission-based system, along with a sandboxing structure in order to enforce security. All apps that require access to any resource (such as reading the device's contacts or recording audio via the microphone) must request the appropriate permissions upon installation. These permissions are defined within a manifest file in the app installer package, called the "AndroidManifest.xml" file. A user can either allow an app access to all the resources it has indicated that it requires or decline installing the app. App resource requests (or permissions) can be a useful resource for determining whether an app may have malicious intent (Sato et al., 2013; Zheng et al., 2013).

Android uses the Dalvik virtual machine to execute apps (and application layer and middleware services), which are written in Java. These apps are stored within zipped files called an Android Package File (APK). In order to run an app, the Dalvik virtual machine reads and executes the "classes.dex" file, contained within the app's APK file, which contains the Dalvik executable code (Ehringer, 2010). Each app also runs within its own Dalvik virtual machine in order to enhance security. Other files contained within the APK file include the manifest file – which contains information such as declarations of resources the app requires and the main launching activity within the app, the "resources.arsc" file – containing some of the resources (such as strings in different languages) of the app in a compressed binary format.

3. Adversary model for android covert data exfiltration

In the model, there exists an adversary with the following capabilities to exploit existing vulnerabilities with the aim of obtaining sensitive data from target devices:

1. `Intercept` (Target device) allows the adversary to intercept communications from the target device.
2. `Inject` (Target device, Entry-point, Message) allows the adversary to inject/infiltrate a message (i.e. binary data such as code) onto the target device via an entry-point (e.g. infiltrated app).
3. `Modify` (Target device, Existing message location, Existing message, New message) allows the adversary to modify an existing message (e.g. by replacing existing code or an SMS with a different code or SMS).
4. `Delete` (Target device, Message location) allows the adversary to delete messages stored on the target device.
5. `Encrypt/Decrypt` (Target device, Message location, Key) allows the adversary to either encrypt (e.g. in the case of ransomware such as CryptoLocker) or decrypt a message on the target device.
6. `Transmit` (Target device, Message) allows the adversary to transmit/exfiltrate message (i.e. binary data such as code and SMS) from the target device.
7. `Listen` (Target device) allows the adversary to passively monitor the communication channel on the target device.

Previous research utilizing adversary models in Android security and data exfiltration techniques discuss these adversary models in a very detached manner. Adversary models are often mentioned but not further detailed by the authors. For example, Davi et al. (2011) proposed a privilege escalation attack for Android devices and assumed the use of a "strong adversary". However, they did not specify all of the capabilities that this adversary had. Without a concrete adversary model and adversary capabilities, it is difficult to simulate an adversary for a particular environment (i.e. not generalizable).

Other work provided adversary models that were specific and difficult to adapt to other areas of research. Wu et al. (2013) considered the effect of vendor customizations on Android's security. The authors suggested an adversary model which allowed the adversary to assume the role of a malicious third-party app on an Android device. The adversary was limited by both the Android sandboxing system and the fact that it could not request any permission that was considered "sensitive". Similarly, Zhou et al. (2013) investigated what information a zero-permission adversary (i.e. a third-party app requesting no permission) could obtain from an Android device. The adversary models of Wu et al. (2013) and Zhou et al. (2013) can be considered similar in nature. Such a specialized adversary model cannot be used in many other areas of Android security. For instance, a different adversary model (with equally different adversary capabilities) must be utilized in order to simulate an attacker seeking to passively listen to messages sent and received on an Android device within a particular network. The third-party app adversary model of Zhou et al. (2013) is inapplicable in this scenario. Ren et al. (2013) proposed a scheme for uniquely identifying users in a healthcare system based on their gait (obtained via the mobile device's accelerometer) and, in doing so, offered an applicable adversary model. The adversary in this model is a user of the healthcare system who seeks to masquerade as another active user of the system. This model is, once again, specific to their area of research.

Bindschaedler et al. (2012) presented a passive adversary model, where the adversary is able to eavesdrop on messages sent by devices in a network. The weaker model was used to evaluate the effectiveness of their proposed privacy protection technique against user tracking attacks. The adversary is only able to eavesdrop on messages in a specific area of the network and has no information on any messages sent outside of this area. The adversary is also unable to capture all messages in this area due to hardware limitations and is

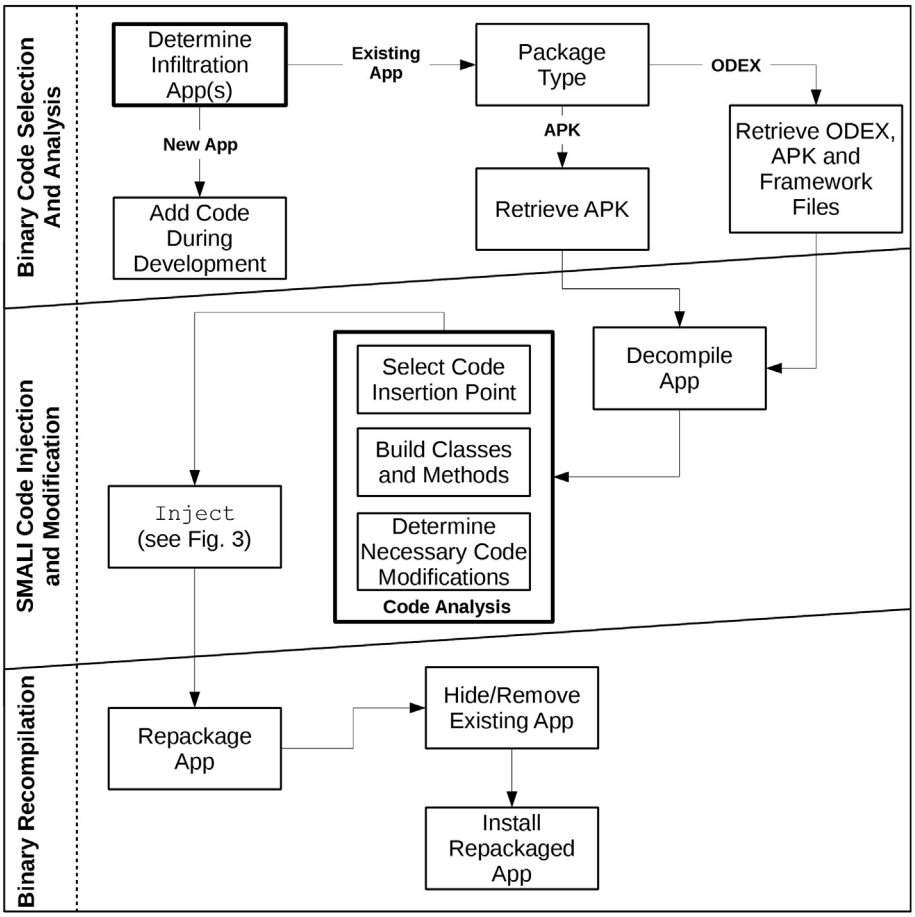


Fig. 2 – Mobile data exfiltration technique (MDET).

unable to inject or create false messages within the network. This significantly weakened and specialized adversary may be useful for the purposes of their research into modeling an attacker attempting to eavesdrop on a particular network and launch user-tracking attacks, but is difficult to apply to other research.

These examples of previous work in the area have highlighted the need for a generalized and well-formalized adversary model. The adversary model proposed in this research captures an adversary's capabilities in a distinct manner and is of a general nature and, as such, it can be applied to many areas of security (and data exfiltration) research. We now construct our mobile data exfiltration technique (MDET) for Android devices using this adversary model.

3.1. *Mobile data exfiltration technique*

MDET is designed to exploit various exfiltration mediums and support numerous methods of code injection ('entry-points') with a view to extracting binary data from Android devices. The process that supports this system consists of three major phases comprising binary code selection and analysis, SMALI code injection and modification, and binary recompilation – see Fig. 2 and Fig. 3.

The processes are discussed in greater detail as follows.

3.1.1. *Binary code selection and analysis*

In the first stage of data exfiltration from a mobile device, the adversary needs to determine the entry point to inject a message (e.g. code) onto the target device. For example, the adversary needs to select the location from which the data can be extracted (i.e. which app should be infiltrated) and then

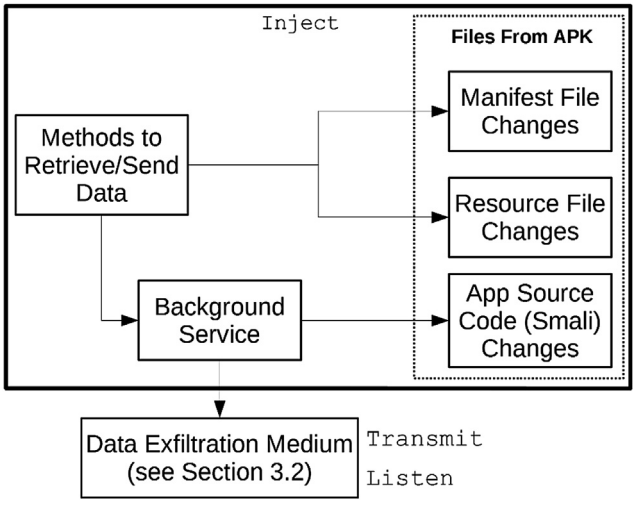


Fig. 3 – Inject (i.e. code injection).

analyze the executable package. In terms of selecting an extraction location, a decision must be made as to whether the code will be injected into the mobile device via the installation of a new app (under control of the adversary) and/or the modification of an existing app (not developed by the adversary).

Installation of a new app would generally require the user to be aware of the app's existence and approve of its installation. The user would need to trust the new app to approve its installation and possibly its access to existing data or be provisioned with new data by the user. In this case, it is likely that the exfiltration function of the app will be hidden behind a legitimately useful function which the app provides. For example a calendaring app could exfiltrate the user's appointments whilst not appearing malicious based solely on the permissions the app requests (which would seem in line with its function).

If a new app is to be installed, then code 'infiltration' is not required since the exfiltration code can be incorporated into the app before it is compiled. This would not be possible if the exfiltration location is an existing app. In many cases the Android sandboxing system makes accessing sensitive data stored by other applications difficult. For example, it would be difficult for an app to access the data stored in the built in email client for exfiltration. Sandboxing makes the insertion of exfiltration code into existing applications a potentially more valuable attack strategy for an adversary. For the remainder of this section, we will assume that the adversary seeks to use an existing app as the interception location.

Injecting exfiltration code into an existing app avoids the issue of the user needing to trust the app as users often implicitly trust apps preinstalled on their mobile device or highly popular apps (such as social networking apps). While the existing app method seems to be the obvious choice for exfiltrating sensitive data, it does present one major limitation. In our experiments, we required (short term) physical access to the unlocked mobile device to 'make the swap' (uninstall the clean app and replace it with the infiltrated version). While this reduces the potential target group size for the attack significantly, it would still be particularly useful in a targeted attack environment (e.g. in an espionage or a national security context).

In our research, we found that there were significant differences in the difficulty and processes required depending on whether the app is a normal APK (generally installed by the end user after ROM development) or an ODEX app (often installed by the vendor during ROM development). Normal APK packages can generally be decompiled to SMALI code with relative ease (as demonstrated in the decompilations undertaken by Do et al. (2014)), but 'ODEX' apps proved much more difficult. However, we also found that ODEX apps were more likely either to be built in or system applications which commonly handle sensitive data (e.g. email, messages, keyboards). While it is possible, in principle, to inject exfiltration code into both types of app, the process differs somewhat in terms of decompilation. This is discussed further in the implementation section (see Section 4).

3.1.2. *SMALI code injection and modification*

Using the decompiled SMALI code, the adversary will select a point of insertion ('entry-point') within the app where it is most appropriate to exfiltrate the data. This is the most complex part of the attack procedure. The MDET system uses an Android service to allow the data exfiltration to continue even when the app is not currently running. The use of a separate service and method also increases the feasibility of the attack as the required changes to the app being modified are significantly reduced. In our implementations, as few as three lines of code were inserted to an existing data handler (in this case, a key press handler) to facilitate the transfer of data from the handler for exfiltration. This small footprint is critical in ensuring that the app can be simply and quickly modified with a high probability of recompilation with few errors or none at all.

This is, however, a more difficult process than it may initially appear as the code is only decompiled to SMALI (a low level programming language) level and not the original Java (a high level programming language) code used by the developer. While it may be easier to work with Java code than SMALI, avoiding the additional decompilation step is necessary as each level of decompilation introduces a significant number of additional differences between the compiled code and the derived source code. In our experiments, we have not been able to successfully recompile an app once decompiled to Java but we have had greater success where the app is only decompiled to SMALI.

Code injection into the relevant data handler is only one component of the required modifications; however, it is perhaps the most difficult. This is due in part to the semantics of the SMALI language that makes adding variables difficult due to its variable count within each method which requires tracking and incrementing. Once the code has been injected to pass the data from the app's handler to the exfiltration service, the services code (previously compiled to SMALI) can then be copied into the existing decompiled package.

One final issue in the SMALI code which must be considered is ensuring that the class path within the SMALI files match. If the adversary is attempting to duplicate the function of a system app (assumedly disabling the legitimate system app) on a device without root access, then they will need to ensure that the class path and package ID are changed appropriately throughout the code. An Android device will not permit the simultaneous installation of two packages with the same package ID.

Once the necessary modifications have been made to the SMALI code, the app's manifest file will also need to be updated. These updates will depend on the changes made to the app, and in our experiments, the updates included changing the app ID or class path and inserting code to launch the service. It should be noted that the manifest file is stored as a binary XML file and cannot be modified in a text editor as would be possible with a standard XML file (as is possible with the manifest file before it is initially compiled). The manifest file is also not part of the "classes.dex" file (where the SMALI code is located) but is instead considered a resource file within the APK.

In our experiments, we attempted to use the existing decompilation tools available to decompile the manifest file and the resources. However, we found that they were not able to recompile the resources due to deficiencies in the decompilation process. As such, we found that we needed to modify

the manifest file while it remained in its (compiled) binary XML format. It is possible to accomplish this using a third party tool; however, it is not a straightforward process.

3.1.3. *Binary recompilation*

Once the code modifications have been completed and the relevant resource data has been updated (e.g. the manifest file), the app would need to be recompiled and repackaged. The exact procedure required for this process varies to an extent between apps and is discussed further in Section 4.1 (see Fig. 4). The generic process involved recompiling the SMALI code to create the modified classes.dex file, which is then inserted with the modified manifest file into the APK with the original resources file. This APK then needs to be resigned with a development key.

Once the APK has been repacked and signed, it then needs to be installed on the mobile device that is to be compromised. There are a number of methods of achieving this including directly over USB via Android Debug Bridge (ADB) or by downloading the package from an HTTP server from the mobile device. We were successful with both installation methods. The previous version of the app also needs to be removed or where this is not possible (as in the case of some system or preinstalled apps) hidden. This can be achieved via uninstalling or disabling the app as appropriate using the Android application manager.

3.1.4. *Detection of MDET*

This technique proposed is similar to anti-forensics in that any activities occurring (such as modification of the manifest file) are likely to leave traces. Our process is more difficult to detect than normal malicious apps as it either uses no permissions or uses permissions which malware detection systems would not generally pick up as malicious.

For example, DroidMOSS (Zhou et al., 2012) relies on the apps already being on the Google Play Store so they can obtain the APK and generate a signature from the app. System apps generally are not located on the Google Play Store due to them being device or manufacturer specific. Thus, DroidMOSS would not be able to detect if the Samsung Keyboard onboard the Samsung Galaxy S3 has been modified. DroidAnalytics (Zheng et al., 2013) also works similarly, but analyzes the apps at a deeper level including comparison of SMALI level source code. Once again, if the apps are not on the Google Play Store, their signatures will be unable to be generated.

Other systems that have been developed include those that detect malware in Android apps by analyzing the permissions that an app requests. This eliminates the need for a version of the app to exist on the Google Play Store but relies entirely on the permissions system having perfect security. Di Cerbo et al. (2011) propose such a system, called AppAware, and also comment that this system would not be able to detect apps that exploit Android vulnerabilities.

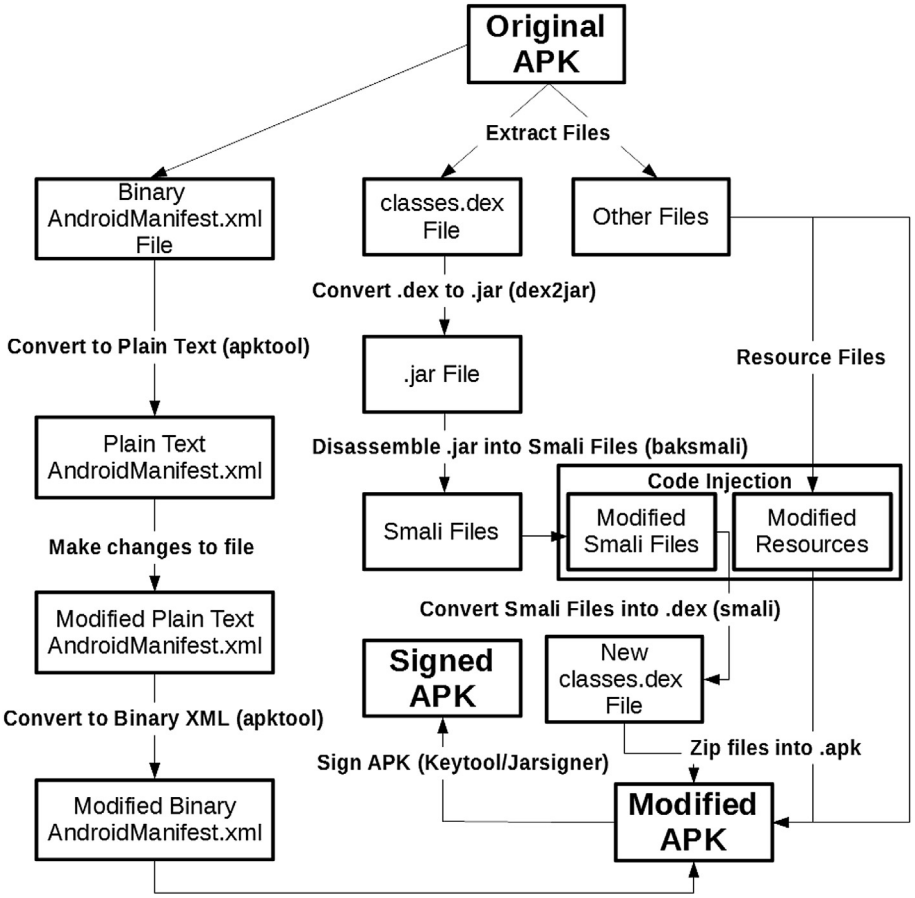


Fig. 4 – In-depth app reverse engineering process.

Table 1 – Existing data exfiltration mediums.

Exfiltration mediums	Limitations	Permissions required	Prevalent usage
Messages (e.g. SMS and MMS)	May be limited based on carrier and plan. Limited by the Android OS in terms of the number of messages in a given timeslot.	SEND_SMS WRITE_SMS	Sending Botnet commands to compromised devices (Zeng et al., 2012), malware propagation (Fleizach et al., 2007) and location tracking (Croft, 2012).
HTTP	Typically monitored by malware detection systems	ACCESS_NETWORK_STATE CHANGE_NETWORK_STATE ACCESS_WIFI_STATE CHANGE_WIFI_STATE INTERNET	Mobile Botnets (Pieterse and Olivier, 2012), key logging attacks (Mohsen and Shehab, 2013). Silent root attacks allowing the adversary to obtain most of the data on a device via the Internet (Egners et al., 2012). Demographical information of the user (mainly by advertising libraries) (Book and Wallach, 2013).
Bluetooth	Transmission range (short)	BLUETOOTH BLUETOOTH_ADMIN	Malware distribution (Castillo, 2011), location tracking and SMS capturing (Cole et al., 2012), remote audio eavesdropping (Bose and Shin, 2006).
NFC	Transmission range (near physical proximity)	NFC	Privilege escalation attacks (Naraine, 2012) that allow the adversary to gain root access and relay attacks (Wang et al., 2012).
Wi-Fi Direct	Transmission range (medium)	ACCESS_WIFI_STATE CHANGE_WIFI_STATE INTERNET	
USB Connections	Transmission range (cable length) Android debugging enabled	None	Computer to phone and phone to computer malware infections (Wang and Stavrou, 2010)
Speaker	Transmission range (varies)	None	
Phone Calls	Visually obvious	CALL_PHONE	
Infrared Emitter	Transmission range (line of sight)	None	

3.2. *Potential data exfiltration mediums*

For the purposes of this research, we define data exfiltration as the retrieval and transfer of data from a device without the user's authorization. We do not include as part of this definition malware that sends messages or calls numbers (Apvrille and Strazzere, 2012; Pieterse and Olivier, 2012; Sarma et al., 2012) as these methods do not generally aim to transmit/exfiltrate a user's personal data, but rather to profit from the infected device or mobile service. Covert data exfiltration can serve a number of purposes; most notably, this includes theft of personal information or credentials. This information is then commonly used by an adversary to impersonate the individual from which the data was stolen or to gain direct access to the information held by the individual (which is generally of a confidential nature).

As shown in Table 1, there are a wide range of possible data exfiltration mediums for an adversary to utilize once they have infiltrated an Android device. As the Android OS matures, more obvious methods such as HTTP via the Internet and Bluetooth become more difficult to exploit. Removing an app's access to the particular resource can prevent the use of these mediums for exploitation outright due to these communication resources requiring explicitly listed Android permissions in the manifest file (which is then enforced by the OS sandboxing).

If malware is seeking to circumvent this and exfiltrate data via one of these controlled permissions, it would need to be included in an installed app or app update. It should be noted the app updates need to be signed using the developer key which signed the original app, and updates that require new permissions will prompt the user to manually accept the update. However, to inject an app that did not originally require the permission, the malware would need to modify the app to request this permission. This would make it obvious that the app had been tampered with to the OS and any anti-malware software on the device. Ideal data exfiltration methods would, therefore, be those that would not require additional permissions in order to operate.

Once the issue of infiltrating the code into an app has been resolved, the technical mediums also have their various limitations which must be addressed. Internet and Bluetooth appear to currently be the most commonly used data exfiltration mediums (see Table 1). This is, in fact, a serious limitation as it makes them a focus for detection by anti-malware and app analysis systems. This is, perhaps, one of the reasons that Internet based data exfiltration via remote servers (Apvrille and Strazzere, 2012; Jung et al., 2013; Zhou et al., 2012) is a commonly researched area with many malware detection systems being capable of detecting it. Bluetooth based transmissions require the Bluetooth to be enabled on the infected device, which the user may notice and simply disable.

NFC and Wi-Fi Direct, like infrared above, are uncommon in many low to medium priced Android smartphones, limiting their availability. For example, upon receiving a request from an app to make a phone call, Android will make it visually

obvious (i.e. full screen dialer or notification center call icons) that an outgoing call is in progress to the user. This significantly limits any practical data exfiltration via this method.

This leaves SMS/MSS as a potential data exfiltration medium. It only requires two permissions, as opposed to the five and two of Internet and Bluetooth respectively. SMS message sending is non obtrusive and is limited only by the mobile network in terms of physical range. It is also relatively simple to implement SMS sending functionality in an app's code, which is an important consideration for an exfiltration medium where code may need to be injected after the development of the app as discussed in Section 3.1.

In contrast to these common (permission controlled) exfiltration mediums, we have also outlined three data exfiltration mediums that require no Android permissions in Table 1. Due to these mediums not requiring a permission request in the manifest file, systems designed to detect malware based on analysis of Android permissions usage (e.g. AppAware (Di Cerbo et al., 2011)) would not be able to detect data exfiltration utilizing these mediums.

One of these mediums is USB connections which are commonly utilized by users to charge and/or transfer data on their phones. One of the major issues in using USB as a data exfiltration medium is that in order for data exfiltration to occur, both devices (i.e. the computer and the connected Android device) must be infected (or in the case of the device at least configured) in order to send data back to the server. The communication between the computer and the adversary's server would also have to be accomplished by other means (such as the Internet).

Another exfiltration medium is the infrared emitter available on some Android devices. At the time of publication, the medium is only available on a select few flagship Android devices. This significantly reduces the potential for this medium to be useful in data exfiltration. Another problem with using infrared signals is the requirement for the sender and receiver to be within (approximate) line of sight.

The final exfiltration medium that is not permission controlled is audio output via the phone loudspeaker. As opposed to the other methods of data exfiltration, this medium has greater potential to be used due to the fact that all phones have a speaker as a basic requirement and it also does not require a physical connection with another device in order to transmit data. There is also currently no research (that we are aware of) that utilizes the phone speaker in order to exfiltrate data from a mobile device in a covert manner (see Section 4.3.5).

In consideration of the various advantages and limitations discussed for each exfiltration medium, we have selected SMS and the audio output (loudspeaker) as our data exfiltration mediums for this research. This is due to their low footprint both in terms of code injection and difficulty of detection, along with the lack of research in these areas into general data exfiltration. Current data exfiltration methods utilizing SMS as the covert channel focus on exfiltrating specific subsets of data. For example, the propagation of malware to other devices and the exfiltration of location data are common aims of SMS-based data exfiltration (see Table 1).

SMS is selected for long-range exfiltration of larger datasets (e.g. images and documents) especially in consideration of the increasing prevalence of unlimited (free) SMS as part of mobile service plans. Audio output is selected for short-range exfiltration of smaller datasets (e.g. passwords, private keys, keystrokes) as no permission is required to use this medium and it can be very difficult for a user to detect its presence.

4. Implementation

To validate the utility of MDET, we attempted to implement two exfiltration mediums, namely SMS and Audio, into a new and existing app respectively. This section discusses the two proof-of-concept apps created as part of this process in detail. We conducted these proof-of-concept experiments using a range of devices and tools. In terms of devices (and Android versions), we used an HTC One X (4.0.1), Samsung Galaxy S3 (4.0.4) and a Nexus 4 (4.4.2) along with various emulated Android versions. The tools used are listed and discussed in the next subsection.

4.1. *SMALI code injection and modification*

Before the exfiltration mediums can be implemented in an existing app, a process for SMALI code injection and modification must be undertaken. This subsection outlines the technical details underlying the process discussed in Section 3 for our implementation in the proof-of-concept apps.

An application called dex2jar (http://code.google.com/p/dex2jar/) is first used to convert the "classes.dex" file contained within an app's installation package (APK) into a Java bytecode ".jar" (Java ARchive) format. This JAR file can be used with Java decompilation tools such as JD-GUI (http://jd.benow.ca/) to partially view the Java source code of the app. However, it is not possible to fully decompile an app into error-free Java source code using such tools. Therefore, we used the bakSMALI (https://code.google.com/p/smali/) application to disassemble the classes.dex file into SMALI assembly code (as the latter is generally error-free due to it being a more direct representation of the DEX language in which the code is stored). At this stage, the app can be fully read and modified in its SMALI code state.

As discussed, other files of importance contained within the APK file are the AndroidManifest.xml file and resources.arsc file along with the resources contained within the "res" folder. These files can be decompiled using the apktool program (https://code.google.com/p/android-apktool/). If, for example, additional activities or services were added to the source code, they must be defined in the AndroidManifest.xml file along with any additional permissions these new services or activities require. The resources.arsc contains compressed resources of the app (e.g. all the strings used throughout the app – titles and app name).

After all modifications are performed, the modified files are written over the original files in the original APK file. As APK contents have changed, the APK must be re-signed before it can be installed. Signing of an app is a major issue that arises if we do not have access to the key that signed the original APK file. Consequently, the new APK will not install over the old one on an Android device. Before this APK can be installed on an Android device, it must first go through the signing process

using the Keytool and Jarsigner tools included with the Java Development Kit (JDK).

4.2. *Proof-of-concept: SMS exfiltration*

4.2.1. *Overview*

The goal of this proof-of-concept is to exfiltrate the target user's stored camera photos via SMS. This process can be easily applied to transmit other files such as videos or documents. This example requires a service or app to run constantly to detect files in the user's DCIM folder (their camera photos) and convert them individually into a Base64 encoded string. In versions of Android up to 4.4, no permissions are required to read files from the SD card (emulated or physical) storage. From version 4.4 onwards, a READ_-INTERNAL_STORAGE permission is required in addition to the SEND_SMS permission which is required for all versions of Android.

Base64 is a commonly used encoding for converting binary data into a printable string. We use this encoding to send data via SMS as it is only designed to send printable symbols. While there are more efficient encoding schemes than Base64 (such as "yEnc" and "Base91"), we found Base64 to be the most reliable encoding scheme as the other schemes used characters which not all SMS networks support.

Algorithms 1 and 2 describe SMS exfiltration sending and reception respectively.

Algorithm 1: SMS Transmission

Input: binary representation of file(s)
Output: sent SMS messages containing encoded file parts

```
files = all designated files;
file_parts = encoded file fragmented (FIFO)
BS = background service for exfiltration code;
TM = SMS transmission method Android API code;
T1 = number of seconds until message limit expires;
M = messages per minute limit;
N = number of sent messages in this minute;
A = max message length;
B = total number of messages for this transmission;

while files ≠ ∅ do
        //encode the first file
        tmp_file = files.pull();
        tmp_encoded = BS.encode(tmp_file);
        tmp_length = tmp_encoded.length();
        B = ceiling(tmp_length / A);

        //split the encoded file string into parts
        i := 0;
        j := 0;
        while i =< tmp_length do
                if j == 0 then //first message (includes total messages + index)
                        file_parts.add(B + '!' + j + tmp_encoded.substring(i, A));
                else if i + A < tmp_length + 1 then
                        file_parts.add(j + '!' + tmp_encoded.substring(i, A));
                else
                        file_parts.add(j + '!' + tmp_encoded.substring(i, tmp_length - 1));
                i = i + A;
                j++;

        //send parts as messages
        foreach file_part ∈ file_parts ≠ ∅ do
                while N < M do
                        TM.send(file_part);
                        N++;
                sleep (T1);
                N = 0;

        //empty the file_parts set
        file_parts = ∅;
```

Once the image has been encoded, it is split into a number of parts with a character size less than the maximum supported message size of an SMS (i.e. 140 characters). Some characters must be reserved for use as index numbers to ensure that the total file is reassembled correctly on the receiving end.

On the receiving end, the parts are reconstructed using their index numbers and the Base64 encoding is decoded to retrieve the binary image.

Algorithm 2: SMS Reception

Input: received SMS messages containing encoded file parts
Output: binary representation of the file

```
message_queue = messages received from compromised device(s);
file = assembled binary file;
SA = set containing encoded file parts;
SR = accumulated string based on messages received;
T = total number of messages expected in this transmission;
MN = source mobile number for transmission;

//process first message
tmp_msg = message_queue.pull();
tmp_array = tmp_msg.split('!');
T = tmp_array.pull();

|SA| = T; //initialise SA to length T

tmp_index = tmp_array.pull();
tmp_encoded = tmp_array.pull();
SA.add(M);

//process remaining messages
while message_queue ≠ ∅ do
        tmp_msg = message_queue.pull();
        tmp_array = tmp_msg.split('!');
        tmp_index = tmp_array.pull();
        tmp_encoded = tmp_array.pull();
        SA.add(I, M); //add encoded message part at index

//reassemble initial encoded string
foreach tmp_string ∈ SA do
        SR.append(tmp_string);

//decode string and return binary file
file = decode(SR);
return file;
```

In our experiments, we found that SMS could not be relied upon to reassemble messages in the order in which they were sent, especially when multiple messages may be sent within the same second.

Versions of Android below 4.4 do not store the sent messages by default. However, in Android 4.4 and above, messages are stored and the sent message needs to be immediately deleted from the Sent folder. To achieve this, the OS requires that the app is the default SMS handler or has SMS deletion enabled via the App Ops interface. If the messages are stored and cannot be deleted, this attack becomes much more difficult to conduct as a user would be bound to notice a large number of encoded text messages in their Sent folder. In the circumstance where the user was running a newer version of Android and cannot be convinced via social engineering to enable the relevant permission via App Ops, then embedding this attack in an SMS app would be the most feasible implementation of this technique.

4.2.2. *Process*

The following outlines the technical process that we implemented in the proof-of-concept. The process commences with the background service or app (either implementation is possible) detecting image files in the "/sdcard/DCIM" folder and subfolders. When image files are found, they are converted into a Base64 string.

An SMS is then instantiated with the first 130 characters of the Base64 encoded image, and a unique SMS index number (to ensure that the total encoded string is reconstructed in the same order that it was sent) is prepended, which is then incremented for each subsequent message. If the message is the first message sent, the total number of messages being sent is also prepended in order for the receiver server (see Fig. 5) to know when an entire image has been received. The background service or app then sends the SMS message and the process loops.

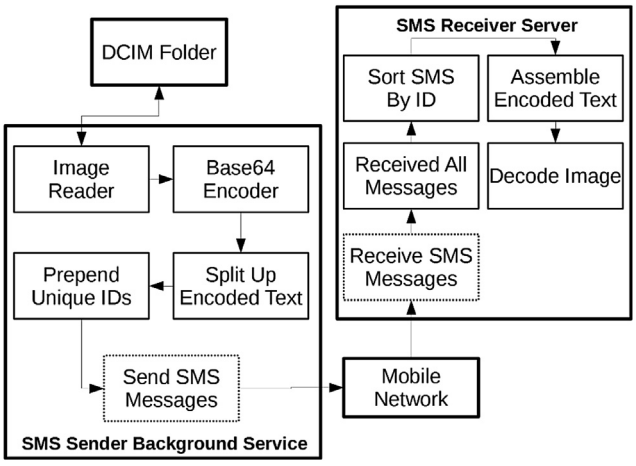


Fig. 5 – SMS exfiltration.

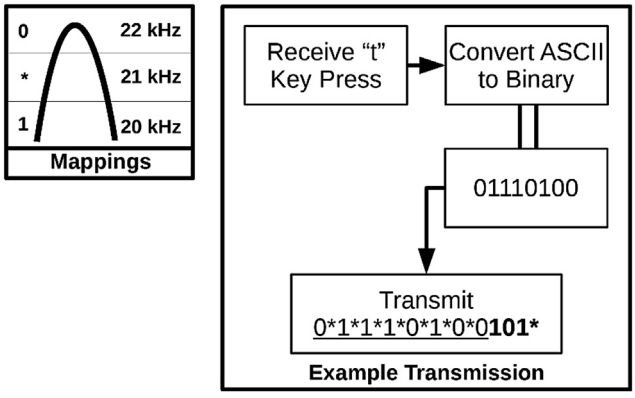


Fig. 6 – Example transmission of the "t" character.

The service or app continues to take the next encoded part of the image, prepend the unique identifier and send it until the entire image has been sent. The messages are all sent to a predefined phone number, which is the server where the messages will be ordered and rebuilt into the original binary format. This will then be decoded to obtain the binary image.

4.3. *Proof-of-concept: inaudible exfiltration*

4.3.1. *Overview*

Although inaudible exfiltration techniques have been implemented successfully on personal computers (see Section 4.3.5), there has been little to no research literature as to whether such a technique is possible on a smartphone. Smartphones have far inferior speakers than those of a personal computer or even laptop device.

Many smartphone speakers are capable of outputting sounds at frequencies outside the 20 Hz to 20 kHz range of human hearing. Similarly, smartphone microphones are capable of picking up sounds outside the human hearing frequencies. This allows an Android device to encode and transmit data inaudibly via the device speaker and allows another Android device to receive and decode the inaudible audio transmission.

To demonstrate this, we designed and successfully implemented a proof-of-concept, which involves injecting exfiltration code into the default keyboard on the Nexus 4 that produces coded inaudible sounds based on which keys the user presses. These audible key presses are then detected and decoded by a microphone on an external device. The default keyboard on the Nexus 4 was chosen due to the fact that (by default) it handled all data input by a user (including communications, usernames, passwords, etc.). Another advantage is that it is always running, allowing services it instantiates to always run as well.

As many users will be able to type on the onscreen keyboard faster than our transmission method is able to transmit, the inaudible sounds are queued as key presses are made. As key presses enter the queue, they are transmitted inaudibly (as discussed below) in a serial manner.

4.3.2. *Transmission method*

An encoding method must be devised to transmit the textual data inaudibly. We attempted to locate an existing transmission scheme for inaudible transmission on an Android device; however, we were unable to locate a method suitable for our experiments. As such, we devised the following transmission method which shares similarities with the Audio Frequency Shift Keying (AFSK) modulation scheme.

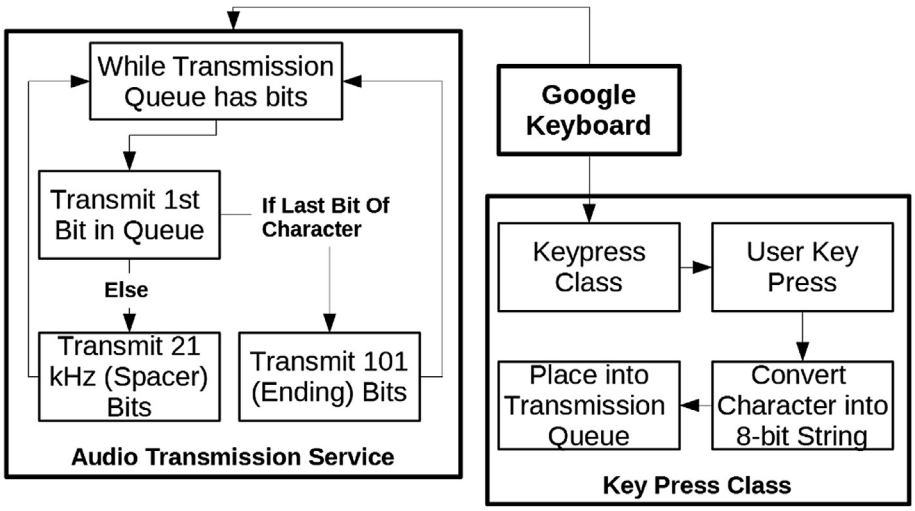


Fig. 7 – Audio exfiltration.

Each character that the user types is converted to an 8-bit binary string and broadcast bit-by-bit. Three frequencies have been selected to transmit data, namely 20 kHz, 21 kHz and 22 kHz. The 20 kHz and 22 kHz ranges are used to represent the binary numbers 1 and 0 respectively. The 21 kHz frequency is used as a spacer frequency transmitted between each bit (0 or 1) transmission (apart from the last bit of a character). After the final bit of a character is sent, a transmission of 101 (20 kHz, 22 kHz then 20 kHz again) without any spacer frequencies is sent. This ending series of bits is useful for error detection. For example, if the receiver has not received 8 bits by the time it receives the 101 ending bits, it can move onto the next character (and mark the current character as corrupt) without the errors affecting the remainder of the characters to be transmitted – see Fig. 6.

Algorithms 3 and 4 describe inaudible exfiltration sending and reception respectively.

Algorithm 3: Inaudible transmission

Input: virtual key-presses as integer ASCII codes
Output: inaudible transmission of binary encoded key-presses

```
keypress_queue = key-presses intercepted from the virtual keyboard;
encoded_queue = key-press ASCII codes represented as binary;
freq_one = frequency for binary 1 – 22 kHz;
freq_zero = frequency for binary 0 – 20 kHz;
freq_spacer = frequency for spacer – 21 kHz;

//queue key-presses
while input ≠ ∅ do
        keypress_queue.add(input)

        foreach keypress ∈ keypress_queue do
                //encode next key-press in queue
                tmp_binary = toBinary(keypress);
                encoded_queue.add(tmp_binary);

        //transmit each encoded key-press
        foreach encoded_key ∈ encoded_queue do
                //iterate then transmit each character
                foreach binary_char ∈ encoded_key do
                        transmit(freq_spacer);
                        if binary_char == '1' then
                                transmit(freq_one);
                        else
                                transmit(freq_zero);
                transmit(freq_one);
                transmit(freq_zero);
                transmit(freq_one);
                transmit(freq_spacer);
```

Algorithm 3 describes the inaudible data transmission scheme. By attaching to the device's keyboard input classes, the inaudible transmission service continuously checks to see whether the keyboard input queue has any characters. If there are any characters, then the algorithm takes the first character, removes it from the queue and adds it in its ASCII binary form to the transmission queue. For each binary digit, a spacer frequency is first sent, then depending on whether the digit is a "1" or a "0", a 22 kHz or 20 kHz signal is transmitted respectively. After the final binary digit for a particular ASCII character has been transmitted, a spacer string is transmitted, consisting of: 22 kHz, 20 kHz, 22 kHz and 21 kHz. This string cannot ordinarily be transmitted by the base transmission scheme due to the need for a spacer (21 kHz) transmission between each character. This ensures that the receiver knows when a character should have been received, and to mark a received string as incorrect if the wrong number of received bits has been received.

Algorithm 4, inaudible data reception, is more difficult to implement than inaudible transmission. This is due to the fact that there may be significant noise in the ambient environment that may be of a high frequency nature. Such noise would interfere with the receiving of data from the transmitting device. As such, the algorithm starts with the averaging of the magnitude of the environment noise levels for each of the considered frequencies: 20 kHz, 21 kHz and 22 kHz. Naturally, a downside of this method is that it cannot be performed whilst the transmitting device is performing inaudible transmissions. The use of an audio spectrum meter would aid in confirming whether this was the case. In order to determine if a bit has been sent, the three frequencies of interest are monitored. If the current magnitude of a frequency exceeds the measured average by a certain value, then the algorithm considers that bit to be received. The value of change chosen (i.e. the increase in magnitude of the signal compared with the average of the frequency in the present environment) will vary depending on the type of sampling data received by the microphone. For example, if a 22 kHz signal is detected above the average magnitude and the currently stored binary string is less than 8 bits long, then a "1" is appended. Similarly, if a 20 kHz signal is detected, the stored

binary string is appended with a "0" (if the stored string contains less than 8 bits). In both of these cases, the algorithm is then set to wait for a spacer signal (21 kHz) to be received. Receiving a spacer then allows the algorithm to await the next "1" or "0" transmission. If at any time, a sequence of 22 kHz, 20 kHz and 22 kHz is received in that order (i.e. "1", "0" and "1" with no spacers) then the full binary string has been received. This string is decoded into an ASCII character and appended to an array containing all received characters. Once there are no more transmissions, the "*keypresses*" array in the algorithm now contains the data that has been received from the inaudible transmission.

Algorithm 4: Inaudible reception

Input: inaudible reception of audio
Output: key-presses as ASCII strings

```
audio_sample = sample of current microphone input;
keypress = contains each binary character of a key-press;
keypresses = set of all received key-presses;
freq_one = frequency for binary 1 – 22 kHz;
freq_zero = frequency for binary 0 – 20 kHz;
freq_space = frequency for spacer – 21 kHz;
freq_one_avg = average background decibels for 22 kHz frequency;
freq_zero_avg = average background decibels for 20 kHz frequency;
freq_space_avg = average background decibels for 21 kHz frequency;
S = number of samples to average;
T = percentage change in frequency to register as a peak;
A = FALSE - Boolean to check if awaiting a spacer frequency;
B1 = FALSE - Boolean to check if a 1 has been received;
B10 = FALSE - Boolean to check if 10 has been received consecutively;
B101 = FALSE - Boolean to check if 101 has been received consecutively;

//calculate average of each frequency
freq_one_avg = calc_avg(freq_one, S);
freq_zero_avg = calc_avg(freq_zero, S);
freq_space_avg = calc_avg(freq_space, S);

while audio_sample ≠ ∅ do
        //get current decibel level for each frequency
        tmp_freq_one = audio_sample.measure_freq(freq_one);
        tmp_freq_zero = audio_sample.measure_freq(freq_zero);
        tmp_freq_space = audio_sample.measure_freq(freq_space);
        //calculate percentage changes in frequency
        tmp_C1= tmp_freq_one/freq_one_avg;
        tmp_C0 = tmp_freq_zero/freq_zero_avg;
        tmp_CS = tmp_freq_space/freq_space_avg;
        //check for peaks
        if tmp_C1 > T and tmp_C1 > tmp_C0 and tmp_C1 > tmp_CS then
                if A ≠ TRUE and keypress.size() < 8 then
                        keypress.add('1');
                        A = TRUE;
                if B1 ≠ TRUE then
                        B1 = TRUE;
                else if B10 == TRUE then
                        B101 = TRUE;
        else if tmp_C0 > T and tmp_C0 > tmp_C1 and tmp_C0 > tmp_CS then
                if A ≠ TRUE and keypress.size() < 8 then
                        keypress.add('0');
                        A = TRUE;
                if B1 == TRUE then
                        B10 = TRUE;
                else
                        B1 = FALSE;
        else if tmp_CS > T and tmp_CS > tmp_C1 and tmp_CS > tmp_C1 then
                A, B1, B10, B101 = FALSE;

        if B101 == TRUE then //if B101 is true then full binary received
                tmp_binary = to_string(keypress);
                tmp_char = binary_to_ascii(tmp_binary);
                keypresses.add(tmp_char);
return keypresses;
```

This approach was chosen solely for its simplicity and speed in demonstrating this proof-of-concept. We acknowledge that there are likely more advanced inaudible encoding schemes; however, signal processing was outside the scope of our research.

4.3.3. *Process*

Similar to the SMS proof-of-concept, the appropriate SMALI files are added or modified in the default keyboards set of SMALI code files. In this case, some analysis is required in order to determine where the keyboard receives key press inputs from the user, and modify this code such that the character ASCII code is sent to the newly added (injected) service. The service constantly runs in the background waiting for the transmission queue to have data in it to transmit.

In order to inject this exfiltration code into a keyboard, several steps must be followed – see Fig. 7.

Firstly, we require SMALI versions of the background service as we are injecting the code in the SMALI form of the app. This can either be done manually, which is time consuming (as the SMALI form of an average Java class can be several times as long and the code is esoteric in nature), or by compiling the background service in a separate app and then decompiling it to the SMALI form.

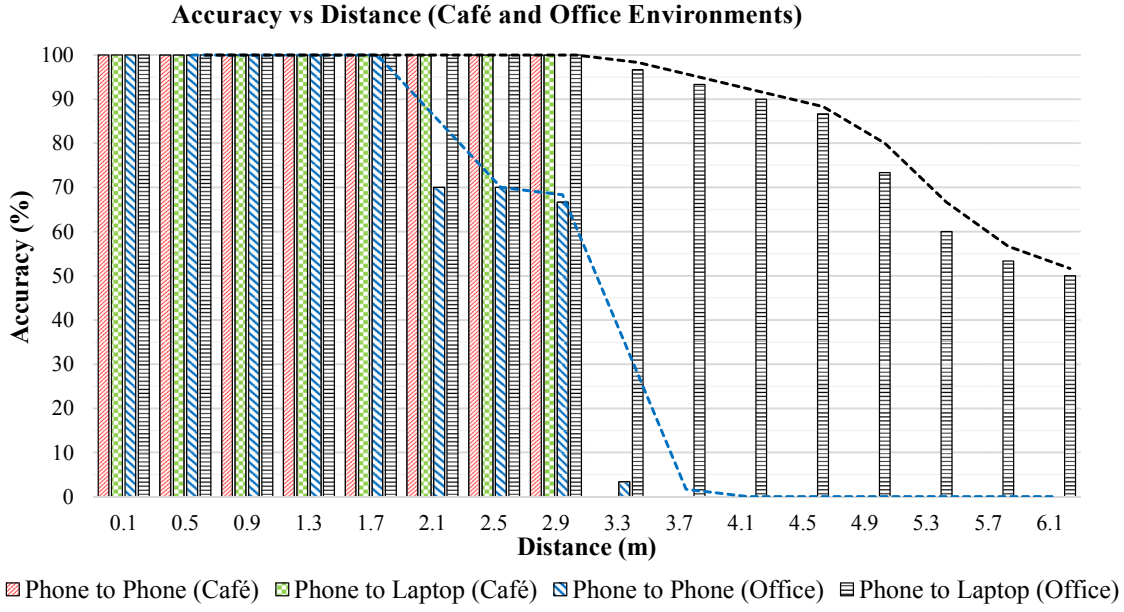
Secondly, the decompiled SMALI files from the default keyboard need to be modified to have updated package paths and then the code created or generated in the first step is integrated with the app's code. The injected code (the code which needs to be inserted within the app's code, as opposed to classes which can be included during the recompile) should be kept minimal (in our proof-of-concept, this comprised approximately four lines of SMALI code). As this exfiltration medium requires a service, our additional service must be added to the AndroidManifest.xml file in order to accommodate this.

In practice, we found that the 22 kHz frequency was too weak for reliable reception on our test devices. Therefore, we move each of the frequency ranges down by 1 kHz (i.e. 19 kHz for '1', 20 kHz for 'spacer' and 21 kHz for '0') in our proof-of-concept app. We found that we were still unable to hear the transmissions (other than the limitation already outlined in Section 5.1). We propose that as mobile devices with improved speakers (e.g. front-facing stereo speakers) become more prevalent, the 22 kHz band will be practical.

4.3.4. *Results*

In order to test the accuracy of our proof-of-concept transmission scheme, we performed a series of experiments in two different environments. We assumed the role of a user entering a password using their keyboard app and used two different devices to receive the inaudible frequencies transmitted by our proof-of-concept app. The experiments were performed in a café of typical ambient noise level and an occupied office with standard ambient noise. We used a Samsung Galaxy SIII (I9300) smartphone and Lenovo ThinkPad Edge E530 laptop as the two receiving devices, which were placed at a stationary position whilst the transmitting device was moved increasing distances away. We recorded the received strings on each of the devices as the distances were increased, noting how many correct characters were received. Fig. 8 details the results of our experiments.

We found that the Samsung Galaxy SIII smartphone microphone was able to accurately (i.e. receive 100% of all data transmitted) receive data in real-time up to a distance of around 1.7 m. Inaudible frequencies transmitted further from this distance (in the office environment) resulted in a drastic decrease in the accuracy of the received data with eventually no data being received at distances over 3.7 m, in the case of the office environment. Interestingly, in the café environment we were able to transmit accurately at a distance of up to 2.9 m



In the café experiments, we only tested up to 2.9 m due to the practicality of conducting experiments in a café environment during working hours.

Fig. 8 – Accuracy of communications between phone (sender) and phone and laptop (receivers) over increasing distances.

(where we concluded testing due to practicality). This is an increase of distance of over 70% when compared with the Samsung Galaxy SIII in the office environment. This could be explained by the fact that the café environment has a constant ambient noise (and, therefore, a stable average ambience in which to detect high frequencies) whilst the office environment is often quiet with moments of loud sounds (e.g. typing on a keyboard, squeaking of a chair and answering of phone calls). These random peaks in sound may introduce noise into the transmitted data.

The inbuilt microphone on the Lenovo ThinkPad Edge E530 was able to accurately receive inaudible frequencies in real-time up to a distance of approximately 2.9 m in both the café and office environments. This is an increase of over 70% compared with the Samsung Galaxy SIII in the office environment. Positioning the transmitting device further away than 2.9 m results in a loss of accuracy, but the laptop microphone was still able to receive more than 50% of the transmitted inaudible frequencies up to a distance of 6.1 m (in the office environment).

4.3.5. *Related work (inaudible exfiltration)*

Existing research on inaudible data exfiltration is generally performed with desktop devices. Research by O'Malley and Choo (2014) shows that inaudible sound is a viable channel for data exfiltration on traditional computing systems (e.g. desktop computers and laptops). The authors demonstrated this by successfully and accurately transmitting RSA secret keys between devices. Hanspach and Goetz (2014) utilized two laptops in order to demonstrate audio-based data transmission. Although the authors used frequencies (18 kHz–18.5 kHz) that are still considered within the range of human hearing (20 Hz–20 kHz), they noted that ultrasonic communication is possible with laptop devices. The authors also noted that using frequencies above 20 kHz would decrease the transmission range. In addition to communicating with two laptops via audio frequencies, the authors also proposed an audio mesh network in order to increase the distance of audio communications.

Security researchers found that harvesting ambient sound, keystrokes and on-screen images from a computer's monitor was possible by attaching a hardware device to the monitor's cable (Marks, 2014). Davis et al. (2014) proposed a technique for extracting audio data from visual sources, such as video. They were able to accurately obtain audio from videos of items such as aluminum foil, glass and potato-chip bags by detecting and analyzing the vibrations on these objects.

A comparison of these existing approaches and their limitations is provided in Table 2.

5. Discussion

5.1. *Limitations*

5.1.1. *Android SMS*

Android versions 4.3 and lower do not add the sent messages into the Sent folder of the messaging app for SMS exfiltration. This means that no further actions are required once the message is sent in order to exfiltrate images from the device. However, in Android 4.4, the API has been changed so that all SMS communication requested from a third party app via code is also broadcasted to the default messaging app on the device. This means another service is required to run simultaneously that will delete these messages as they are sent in order to hide the data exfiltration from the user. App Ops (an esoteric interface for toggling app permissions) is used to grant an app permission to delete messages when it is not the default messaging app.

In addition, Android 4.4 has a base limit of sending 30 SMS messages per minute that is defined in the "com/android/internal/telephony/SmsUsageMonitor.java" file of the Android source code. Every additional message will generate a blocking confirmation dialog box that the user will see and must either allow or deny. This means the SMS sending service must keep under this limit in order to be silent. This limit can only be bypassed by modifying the source code of the Android OS itself and recompiling it.

5.1.2. *APK signature*

As we do not have access to the key used to sign the original APK, we are unable to "update" the APK for the keyboard already installed on the device. This means we must change the package path (unique identifier) of the app in order for the Android device to allow installation.

5.1.3. *Error handling*

To avoid raising suspicion, we did not want the device to request additional permissions (such as RECORD_AUDIO) and therefore, we have opted to make the system forward only transmission. This means we are not able to request retransmission due to interference and dropped bits. We attempted to implement several forward error-detection and error-correcting codes, such as Hamming Code. However, we found that these forward error correction systems did not significantly improve the error free decode rate on the receiving device. For example, the issue with the Hamming Code was that it increased the size of the 8-bit string to a 12-bit string. In other words, an increase of 50% (with a corresponding decrease in speed) only allows for correction of one incorrect bit.

5.2. *Recommendations*

Currently (as of Android 4.4), no permission is required to access the phone's speakers and output audio at any frequency. We have shown this to be a security risk. The ideal solution to prevent apps from leaking data via inaudible sounds would be to ensure that Android requires developers to request for the audio output resource by creating a new permission for it. The challenge, however, would be that this proposed countermeasure would break compatibility with almost all existing apps and outdated apps that developers no longer actively update. An alternative method may be to limit the frequencies in which an app may output sound, and require permission when an app wants to produce inaudible sounds out of the speaker.

The SMS exfiltration attack is significantly limited in the most recent versions of Android by sending speed limits but more significantly by the introduction of only one primary

Table 2 – Inaudible exfiltration solutions comparison.

Approaches	Features	Limitations
Davis et al. (2014)	Uses video (potentially from a distance and through transparent objects) to determine the sound being emitted in the vicinity of the object.	Unlikely to be suitable for covert data exfiltration. The solution focuses on human voice which is significantly different from an audio perspective in comparison to AFSK. Multiple audio sources in the vicinity of the monitored object may significantly reduce the utility of the suggested approach.
Hanspach and Goetz (2014)	Uses laptop computers to communicate via audible frequencies. To increase the distance between audio emitting nodes, they implemented an 'audio mesh network'.	The proposed solution uses audible frequencies limiting its utility for use in covert operations. The audio output and input capabilities of laptop computers in comparison to mobile devices are generally vastly improved due to the speaker/microphone size and placement.
Marks (2014)	Describes a range of physical 'retro-reflector' bugs used to replicate signals being sent across physical connections (such as monitor and keyboard cables) to a remote attacker.	Physical bugs are much harder to install and it may be difficult to remove the bugs at a later stage once the interception operation has concluded.
O'Malley and Choo (2014)	Uses laptop computers to communicate via inaudible frequencies.	Similarly to Hanspach and Goetz (2014), the audio output and input capabilities of laptop computers in comparison to mobile devices are generally vastly improved due to the speaker/microphone size and placement.
Our approach	Uses speakers on mobile devices to transmit data via inaudible frequencies to listening devices including laptops and other mobile devices.	See Section 5.1

SMS sending app preventing all others from hiding sent messages.

While (as discussed) there are means to bypass these restrictions, they are quite difficult to execute in the latest version of Android. However, if the App Ops interface is made available in the system settings app in future versions of Android (as would appear by its design), it may be more feasible to socially engineer users to grant the necessary permission manually. Social engineering will be a significant issue to be resolved if the controls for allowing or denying permissions on an Android device is to be handed to the user.

5.2.1. *Audio hardware*

The Android speaker is not designed for rapid changes in audio frequencies that occur when we transmit data using our transmission method. These fast changes cause the speaker to become faintly audible to the user in the form of crackling sounds. In order to prevent this from occurring, a fourth inaudible frequency (that is not used by the transmission method already – in our proof-of-concept, we used 22 kHz) must be constantly broadcast when no other frequencies are being transmitted.

5.3. *Adversary strategies*

The attacks discussed in this paper have the potential to be used both as part of illegal cybercrime and state-sponsored activities and also as part of a legal interception and surveillance operation. For the purposes of discussing their practical implementation aspects, we will use the latter scenario. In practice, the exfiltration mediums selected will operate best when matched with the type of data being exfiltrated and the circumstances under which exfiltration can be undertaken.

The SMS exfiltration technique is designed to collect larger files such as images and documents and exfiltrate them using relative stealth even on devices where internet access is not available, restricted via VPN or software protections have been installed (e.g. software firewalls). However this technique relies on the exfiltration target having unlimited free SMS messages as part of their mobile service plan (a growing practice in many countries).

The inaudible exfiltration technique is designed to collect smaller data items such as passwords and encryption keys and in effect broadcast them into the inaudible frequencies surrounding the device user (although these broadcasts could be coded/encrypted to prevent third party eavesdropping if necessary). These inaudible frequencies can be detected using a standard microphone (such as those on another smartphone) as demonstrated in our proof-of-concept app. However, it is not likely that this would be a common use case in practice. In a practical application, a higher gain microphone (likely directional and tuned for the relevant frequencies) could be used to detect the inaudible sounds from a greater distance. While we were not able to verify this application in this experiment, we intend to do so as part of our future work.

6. Conclusion

The cyber threat landscape is an extremely fast-moving environment. Only a decade ago, several criminologists warned that 'those who fail to anticipate the future are in for a rude shock when it arrives' (Smith et al., 2004, p. 156). A number of recent high profiles incidents involving mobile devices (see Prevelakis and Spinellis, 2007; Steen, 2013; Whinnett, 2014) highlighted the potential for confidential information or communications to be leaked from the use of mobile devices. A recent report by Alcatel-Lucent (2014), for example, noted that '[c]urrently, most mobile malware is distributed as Trojanized apps, and Android offers the easiest target for this approach, because of Android's lenient security measures on the handling of apps'.

In this paper, we proposed a Dolev-Yao type adversary model that captures the capabilities of an adversary in covertly

exfiltrating data from Android devices (one of the most popular mobile platforms). Using this adversary model, we constructed a mobile data exfiltration technique (MDET) that allows an adversary to exploit various exfiltration mediums to covertly extract data from Android devices. We demonstrated that it is possible to inject malicious code into trusted apps on Android devices due to the fact that app signatures are not verified against the original developer's key by the OS based on the package's name and key assets. For example, a package 'com.google.android.inputmethod.latin' presenting the name Google Keyboard with the Google logo should be signed with a key registered to Google. Our proof-of-concept apps demonstrated that SMS and inaudible audio transmission are both viable mediums for covertly exfiltrating confidential data from trusted apps on current Android devices. The attacks (based on MDET) outlined in this paper have the potential to affect a range of different applications and user communities.

An extension of this work includes exploring the use of more advanced inaudible encoding schemes that would allow data exfiltration in a broader range of situations. One potential path is using two-way communications (feedback) with inaudible data exfiltration by requesting the RECORD_AUDIO permission. This would allow for much greater speeds along with an improvement in data quality and accuracy with the downside of requiring a permission. This could be further improved by implementing an inaudible mesh network (Hanspach and Goetz, 2014).

Future work also includes constructing other attack techniques using the adversary model for different situations and devices (e.g. iOS devices, Windows devices and Internet-connected devices).

REFERENCES

Alcatel-Lucent. Kindsight security labs malware report – H1 2014. 2014. viewed 10 September 2014, http://resources.alcatel-lucent.com/?cid=180437.

Apvrille A, Strazzere T. Reducing the window of opportunity for Android malware gotta catch'em all. J Comput Virol 2012;8(1–2):61–71.

Backes M, Gerling S, Hammer C, Maffei M, von Styp-Rekowsky P. AppGuard–Enforcing user requirements on Android apps. In: Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer; 2013. p. 543–8.

Bindschaedler L, Jadliwala M, Bilogrevic I, Aad I, Ginzboorg P, Niemi V, et al. Track me if you can: on the effectiveness of context-based identifier changes in deployed mobile networks. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium; 2012.

Böhmer M, Hecht B, Schöning J, Krüger A, Bauer G. Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage. In: Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services. ACM; 2011. p. 47–56.

Book T, Wallach DS. A case of collusion: a study of the interface between ad libraries and their apps. In: Proceedings of the 3rd ACM workshop on Security and privacy in smartphones & mobile devices; 2013. p. 79–86.

Bose A, Shin KG. On mobile viruses exploiting messaging and bluetooth services. Securecomm and Workshops; 2006. p. 1–10.

Castillo CA. Android malware past, present, and future. 2011. viewed 26th February 2014, http://www.mcafee.com/us/resources/white-papers/wp-android-malware-past-present-future.pdf.

Cole KA, Silva RL, Mislan RP. All bot net: a need for smartphone P2P awareness. Digital forensics and cyber crime. Springer; 2012. p. 36–46.

Croft N. On forensics: a silent SMS attack. Information Security for South Africa (ISSA); 2012. p. 1–4.

Davi L, Dmitrienko A, Sadeghi A-R, Winandy M. Privilege escalation attacks on Android. In: Burmester M, et al., editors. Information Security, vol. 6531. Berlin Heidelberg: Springer; 2011. p. 346–60.

Davis A, Rubinstein M, Wadhwa N, Mysore GJ, Durand F, Freeman WT. The visual microphone: passive recovery of sound from video. ACM Trans Graph 2014;33(4):79:1–79:10.

Di Cerbo F, Girardello A, Michahelles F, Voronkova S. Detection of malicious applications on android os. Computational forensics. Springer; 2011. p. 138–49.

Do Q, Martini B, Choo K-KR. Enhancing user privacy on Android mobile devices via permissions removal. In: Proceedings of the 47th Hawaii International Conference on System Sciences; 2014. p. 5070–9.

Egners A, Meyer U, Marschollek B. Messing with Android's permission model. In: Proceedings of the 11th International Conference on Trust, Security and Privacy in Computing and Communications; 2012. p. 505–14.

Ehringer D. The Dalvik virtual machine architecture. 2010. viewed 3rd May 2014, http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf.

Fleizach C, Liljenstam M, Johansson P, Voelker GM, Mehes A. Can you infect me now?: malware propagation in mobile phone networks. In: Proceedings of the 5th ACM Workshop on Recurring Malcode; 2007. p. 61–8.

Gartner. Gartner says worldwide pc, tablet and mobile phone combined shipments to reach 2.4 billion units in 2013. 2013. viewed 13th May 2014, http://www.gartner.com/newsroom/id/2408515.

Goodin D. Meet "badBIOS," the mysterious Mac and PC malware that jumps airgaps. 2013. viewed 1st May 2014, http://arstechnica.com/security/2013/10/meet-badbios-the-mysterious-mac-and-pc-malware-that-jumps-airgaps.

Hanspach M, Goetz M. On covert acoustical mesh networks in air. J Commun 2014;8(11):758–67.

Hornyack P, Han S, Jung J, Schechter S, Wetherall D. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In: Proceedings of the 18th ACM conference on Computer and communications security. ACM; 2011. p. 639–52.

Jung J-H, Kim JY, Lee H-C, Yi JH. Repackaging attack on Android banking applications and its countermeasures. Wirel Personal Commun 2013;73(4):1421–37.

Kern M, Sametinger J. Permission tracking in Android. In: Proceedings of the 6th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies; 2012. p. 148–55.

Marks P. Hackers reverse-engineer NSA's leaked bugging devices. NewScientist; 2014. viewed 5th September 2014, http://www.newscientist.com/article/mg22229744.000-hackers-reverseengineer-nsas-leaked-bugging-devices.html#.U7Y7B60cy9I.

Mohsen F, Shehab M. Android keylogging threat. In: Proceedings of the 9th International Conference Conference on Collaborative Computing: Networking, Applications and Worksharing; 2013. p. 545–52.

Naraine R. Exploit beamed via NFC to hack Samsung Galaxy S3 (Android 4.0.4). 2012. viewed 26th February 2014, http://www.zdnet.com/exploit-beamed-via-nfc-to-hack-samsung-galaxy-s3-android-4-0-4-7000004510/.

O'Malley SJ, Choo K-KR. Bridging the Air Gap: Inaudible data exfiltration by Insiders. In: Proceedings of the 20th Americas Conference on Information Systems; 2014. p. 1–12.
Pieterse H, Olivier MS. Android botnets on the rise: trends and characteristics. In: Proceedings of Information Security for South Africa; 2012. p. 1–5.
Prevelakis V, Spinellis D. The Athens Affair. IEEE Spectr 2007;44(7):26–33.
Ren Y, Chen Y, Chuah MC, Yang J. Smartphone based user verification leveraging gait recognition for mobile healthcare systems. In: Proceedings of the 10th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks. IEEE; 2013. p. 149–57.
Sanger DE, Shanker T. N.S.A. Devises radio pathway into computers. 2014. viewed 1st May 2014, http://www.nytimes.com/2014/01/15/us/nsa-effort-pries-open-computers-not-connected-to-internet.html?_r=1.
Sarma BP, Li N, Gates C, Potharaju R, Nita-Rotaru C, Molloy I. Android permissions: a perspective combining risks and benefits. In: Proceedings of the 17th ACM symposium on Access Control Models and Technologies; 2012. p. 13–22.
Sato R, Chiba D, Goto S. Detecting Android malware by analyzing manifest files. In: Proceedings of the Asia-Pacific Advanced Network; 2013. p. 23–31.
Smith RG, Grabosky P, Urbas G. Cyber criminals on trial. Cambridge: Cambridge University Press; 2004.
Steen M. Merkel's phone tapped by US since 2002, leaked documents claim. 2013. viewed 10 September 2014, http://www.ft.com/cms/s/0/65044af4-3f15-11e3-b665-00144feabdc0.html.
Wang Z, Stavrou A. Exploiting smart-phone usb connectivity for fun and profit. In: Proceedings of the 26th Annual Computer Security Applications Conference; 2010. p. 357–66.
Wang Z, Xu Z, Xin W, Chen Z. Implementation and analysis of a practical NFC relay attack example. In: Proceedings of the 2nd International Conference on Instrumentation, Measurement, Computer, Communication and Control; 2012. p. 143–6.
Whinnett E. Foreign Minister Julie Bishop's phone was hacked at the height of the MH17 crisis. 2014. viewed 9 September 2014, http://www.heraldsun.com.au/news/foreign-minister-julie-bishops-phone-was-hacked-at-the-height-of-the-mh17-crisis/story-fni0fiyv-1227026241325?nk=f8bcef3316054a909f3bcf4dc5402c40.
Wu D-J, Mao C-H, Wei T-E, Lee H-M, Wu K-P. Droidmat: Android malware detection through manifest and API calls tracing. In: Proceedings of the 7th Asia Joint Conference on Information Security. IEEE; 2012. p. 62–9.
Wu L, Grace M, Zhou Y, Wu C, Jiang X. The Impact of vendor customizations on Android security. In: Proceedings of the 20th Conference on Computer and Communications Security. ACM; 2013. p. 623–34.
Xu R, Saïdi H, Anderson R. Aurasium: practical policy enforcement for Android applications. In: Proceedings of the 21st USENIX Conference on Security Symposium; 2012. p. 539–52.
Zeng Y, Shin KG, Hu X. Design of SMS commanded-and-controlled and P2P-structured mobile botnets. In: Proceedings of the 5th ACM conference on Security and Privacy in Wireless and Mobile Networks; 2012. p. 137–48.
Zheng M, Sun M, Lui J. Droid Analytics: a signature based analytic system to collect, extract, analyze and associate Android malware. In: Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications IEEE; 2013. p. 163–71.
Zhou W, Zhou Y, Jiang X, Ning P. Detecting repackaged smartphone applications in third-party Android marketplaces. In: Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy. ACM; 2012. p. 317–26.
Zhou X, Demetriou S, He D, Naveed M, Pan X, Wang X, et al. Identity, location, disease and More: inferring your secrets from Android public resources. In: Proceedings of the 20th Conference on Computer and Communications Security. ACM; 2013. p. 1017–28.
Zhou Y, Zhang X, Jiang X, Freeh VW. Taming information-stealing smartphone applications (on Android). In: Proceedings of the 4th International Conference on Trust and Trustworthy Computing. Springer; 2011. p. 93–107.

Quang Do is a PhD Scholar at the University of South Australia, and holds a Bachelor of Computer Science (First Class Honours). He is the recipient of the prestigious University of South Australia Vice Chancellor and President's Scholarship, and is an active Android security researcher.

Ben Martini is a Research Associate at the University of South Australia. His research focus is digital forensics and information security, particularly relating to the cloud and mobile nexus. He has published a book entitled "Cloud Storage Forensics", and a number of refereed conference and journal articles. Ben has worked actively in the South Australian IT industry sectors including government departments, education and electronics across various organizations and continues to deliver occasional invited presentations to industry organizations in his area of expertise.

Dr Kim-Kwang Raymond Choo is a Fulbright Scholar and Senior Lecturer at the University of South Australia. His publications include a book in Springer's "Advances in Information Security" series and a book published by Elsevier (Forewords written by Australia's Chief Defence Scientist and Chair of the Electronic Evidence Specialist Advisory Group). His awards include the British Computer Society's Wilkes Award for the best paper published in the 2007 volume of Computer Journal. He is the editor of IEEE Cloud Computing Magazine's "Cloud and the Law" column and the Book Series Editor of Syngress/Elsevier's "Advanced Topics in Security, Privacy and Forensics".