Android anti-forensics at the operating system level

Karl-Johan Karlsson
1100965

Submitted in partial fulfilment of the requirements for the degree of Master of Science in Computer Forensics and E-Discovery in the Humanities Advanced Technology and Information Institute, University of Glasgow

September 3, 2012

Abstract

In forensic analysis of mobile phones, current tools let the phone software run and ask it to provide the data stored on the phone. This means that the correctness and completeness of the analysis depends on the cooperation of the phone software. Many phones running the Android operating system can have their software replaced by the end user. This dissertation presents a set of modifications to the CyanogenMod community distribution of the Android operating system which provide false data to the forensic analysis tools Cellebrite and XRY, while normal use of the phone is unaffected. Modifications are also presented to delay forensic data extractions and to prevent installation of the forensics tools.

Contents

1 Introduction **1**

2 Previous work **4**

2.1 Android anti-forensics 4

2.1.1 Data hiding 5

2.1.2 Artefact wiping 6

2.1.3 Trail obfuscation 7

2.1.4 Attacks against processes and tools 8

2.2 Weaknesses in current approaches 9

2.2.1 Trusting the operating system 9

2.2.2 Unspecific triggering 9

2.2.3 Timing constraints 10

2.2.4 Permanence 10

3 Android **12**

3.1 Content providers 12

3.2 Contact lists 13

3.3 SMS messages 14

3.4 Intents 14

3.5 Package management 15

3.6 Rooting 15

3.7 Community distrubutions of Android 16

4 Methodology **18**

4.1 Scope 19

4.2 Limitations 20
4.2.1 Triggers for anti-forensic behaviour 20
4.2.2 Visibility 22
4.2.3 File system access 22
4.3 Experimental design 24
4.3.1 Examination of current behaviour 24
4.3.2 Implementation of anti-forensic modifications 25

5 Implementation, testing and results 27
5.1 Examination of forensics tools 27
5.1.1 Cellebrite 28
5.1.2 XRY 28
5.2 Triggering 29
5.3 Anti-forensics modules 30
5.3.1 Response delays 30
5.3.2 Rejecting installation of forensics tools 32
5.3.3 Hardcoded false contact list 32
5.3.4 False contact list from alternate databases 33
5.3.5 Delayed restoration 33
5.3.6 Hiding SIM contacts 34
5.3.7 Hiding SMS messages 34
5.4 Testing and results 35
5.4.1 Triggering 35
5.4.2 Response delays 36
5.4.3 Rejecting installation of forensics tools 38
5.4.4 Hardcoded false contact list 38
5.4.5 False contact list from alternate databases 40
5.4.6 Delayed restoration 41
5.4.7 Hiding SIM contacts 43
5.4.8 Hiding SMS messages 44

6 Conclusions 49
6.1 Trust 49

6.2 Modification . . . 50
6.3 Detection . . . 51
6.4 Reverting . . . 51
6.5 Hypothesis . . . 52

7 Future work **54**
7.1 Decompilation of forensic applications . . . 54
7.2 Detection of forensics tools . . . 55
7.3 Encryption . . . 57
7.4 Unrooting . . . 57
7.5 SEAndroid . . . 58
7.6 Data destruction . . . 59

A Software and hardware environments **60**

B Tool behaviour **61**
B.1 Cellebrite . . . 61
B.2 XRY behaviour . . . 64

C Source code **65**
C.1 USBMonitor . . . 65
C.1.1 AndroidManifest.xml . . . 65
C.1.2 res/layout/main.xml . . . 66
C.1.3 res/values/strings.xml . . . 66
C.1.4 USBMonitorActivity.java . . . 66
C.1.5 USBBroadcastReceiver.java . . . 68
C.1.6 USBUeventObserver.java . . . 69
C.2 Instrumentation of contacts provider . . . 69
C.3 Rejecting installation of forensics tools . . . 84
C.4 Delayed responses . . . 91
C.5 Hardcoded false data . . . 94
C.6 False data from alternate databases . . . 109
C.7 Delayed restoration . . . 117
C.8 Hiding SIM contacts . . . 123

C.9 Hiding SMS messages 128

D Turnitin results **138**

E Declaration of originality **139**

List of Figures

5.1 Contacts extracted by Cellebrite from an unmodified phone . . 28
5.2 Contacts extracted by XRY from an unmodified phone 29
5.3 Warning shown when enabling USB debugging 30
5.4 Screenshots of USBMonitor output 36
5.5 Error from Cellebrite with a ten second response delay 37
5.6 Cellebrite extraction logs showing elapsed time 37
5.7 Extraction log from XRY with a six hour response delay . . . 37
5.8 Error from Cellebrite when installation was rejected 38
5.9 Results from XRY when installation was rejected 39
5.10 Contact list application with real data 40
5.11 Cellebrite extraction report showing hardcoded false data . . . 41
5.12 XRY extraction report showing hardcoded false data 41
5.13 Contact list application showing empty contact list when USB debugging is on 42
5.14 Contacts fed to Cellebrite from an alternate database 43
5.15 Contacts fed to XRY from an alternate database 43
5.16 Contact lists with delayed restoration, first extracted with Cellebrite 45
5.17 Contact lists with delayed restoration, first extracted with XRY 46
5.18 Cellebrite extraction reports for SIM contact lists 47
5.19 Cellebrite extraction logs showing SMS messages 47
5.20 Screenshots of the built-in messaging application 48

Chapter 1

Introduction

According to the International Telecommunications Union, at the end of 2011 there were almost 6 billion mobile phone subscriptions for a world population of 7 billion [62]. In the first quarter of 2012 alone, 144 million smartphones were sold [41], of which 56% were running Android [40]. At that time, a total of 331 million Android devices had been activated, representing 59% of the global smartphone population [47].

Smartphones are essentially general-purpose computers with an attached phone. As such, many people use smartphones for their daily computation, storage and communications tasks. This makes smartphones a great source of forensic evidence. However, smartphones present a particular set of forensics difficulties compared to PCs.

In traditional PC forensics, it is usually possible to power down and disassemble a system, whereupon the components can be analysed independently. If a hard disk is connected through a write blocker to an analysis workstation, changes cannot be made to the disk and any anti-forensics programs installed on it do not run automatically. This workflow is supported by the Association of Chief Police Officers guidelines for digital forensics, whose first principle states: "No action taken by law enforcement agencies or their agents should change data held on a computer or storage media which may subsequently be relied upon in court" [53].

For mobile phones, this kind of procedure is often unavailable. Mo-

bile phones are highly integrated and built from non-standard components, running software which is often proprietary, undocumented and frequently changed. To perform a similar component-by-component analysis, the analyst would start by disassembling the phone and removing the surface-mounted memory chips, which is a delicate procedure with a high risk of damage. The memory chips can be read by standardised readers, but the interpretation of the data depends on the software which was running on the phone.

A much easier method is to let the phone run, and access the data through the normal interfaces provided by the software. This presents a high risk of data being modified, both as a normal function of the phone and by specialised anti-forensic applications. However, the savings in time and effort are great enough that this method is endorsed by both ACPO and the American National Institute of Standards and Technology [63].

Because of this, forensic analysts rely on the correct functioning of the phone's software when performing analyses. This means that changing that functionality is a way of thwarting analysis. Smartphones running operating systems such as Android and iOS are designed to allow the installation of third-party applications, so such applications with anti-forensic purposes have been developed [51, 46, 68, 83]. By being regular applications, such anti-forensic systems have to work under the restrictions imposed by the system, such as application isolation and responsiveness demands. If anti-forensic modifications were to be made on a lower level, these restrictions would not apply in the same way, possibly making more advanced methods available. This project investigates the viability of operating system-level anti-forensics for Android smartphones.

The rest of this dissertation is structured as follows. Chapter 2 provides an overview of previous work in the field of Android anti-forensics. Chapter 3 describes some details of the Android system relevant to the implemented anti-forensic system. The methodology is described in chapter 4, and the implementation, testing and results in chapter 5. Conclusions are in chapter 6, with suggestions for future work in chapter 7. Appendix A describes the software and hardware used. Logs of query behaviour exhibited by the

forensics tools Cellebrite and XRY are in appendix B. Finally, appendix C contains the source code for all modifications.

Chapter 2

Previous work

Android is a young system, with the first commercial device, the HTC Dream, also known as the T-Mobile G1, launched in September 2008 [38]. In contrast, one of today's most common PC systems, Windows NT with NTFS, was introduced with Windows NT 3.1 [84] in 1993 and entered mainstream use with the launch of Windows XP in 2001 [37]. As such, it can be expected that the Android forensics and anti-forensics literature will be less mature than the ones for Windows PCs.

Mobile phone forensics in general is a field under rapid development, as are mobile phones themselves. Books written on the subject (e.g. [60]) quickly become obsolete with the release of new devices and new versions of operating systems. There are fairly recent articles on mobile phone forensics which are simple overviews of the field or confirmations that forensics tools perform as advertised [67, 96]. Some more advanced research topics are beginning to emerge, such as live memory acquisition [92] and using data found on phones as proxy evidence for remote data [55].

2.1 Android anti-forensics

Kessler [64] categorises anti-forensics into four groups: data hiding, artefact wiping, trail obfuscation, and attacks against forensics processes or tools.

2.1.1 Data hiding

For PC anti-forensics, this group contains things like steganography, deleted files, and storing data in the cloud or in other users' storage space. These would be substantially similar for an Android phone, with the caveat that recovery of deleted files depends on the file system used, which is usually YAFFS2[1], which may be unsupported in commercial forensics tools.

Specific for Android is the separation between different applications enforced by the operating system. Every application is run as its own Linux user, and standard Linux file system permissions are used to ensure that no other application can read its files. This also applies to the applications uploaded to the phone by forensic analysis tools. This protection can only be bypassed if the phone is first rooted. If that is done, forensics tools such as Cellebrite and XRY can use the elevated privilege to read the entire file system.

On a non-rooted phone, then, information can be hidden by having an application store it somewhere secluded and restore it at a later time (such as when the user enters a password). This approach was tested by Distefano et al. [51]. Their program takes data from a number of standard databases on the phone (e.g. contact list, call logs, and SMS[2] messages) and user-specified files, copies this data to files in the program's own directory, and deletes the originals. This approach also allows for quick mass deletion, since the Android package manager deletes all files private to an application if it is uninstalled.

They attempted to use the forensics tool Paraben Device Seizure [50], but found that this was incompatible with the phone they were using. Instead, they used backup programs, which require the phone to be rooted and perform a logical acquisition of the phone memory. As expected, these programs were able to read the private directory where the data had been stored. Had the phone not been rooted, the backup programs would not have worked [57].

Distefano et al. say nothing about how their data hiding program is

[1] Yet Another Flash File System, version 2 [82]

[2] Short Message Service, a system for sending short text messages between mobile phones.

triggered, nor when data is put back. They do, however, include test results for how long it takes for the hiding process to run (on the order of 10 seconds, depending on the amount of data to be hidden). This suggests that the hiding process is time sensitive, which would be the case if it was triggered by the connection or starting of a forensics tool. Presumably, the data would then be manually restored by the user after regaining control over the phone.

2.1.2 Artefact wiping

Artefact wiping is the act of overwriting data so that it is impossible to restore, even with undeletion techniques. While the overwritten data will be irrevocably destroyed, Kessler [64] notes two weaknesses with this class of techniques: they may miss some data, and they may leave traces of the wiping having occurred, most notably the presence of the wiping tool. Most of the existing Android anti-forensic literature is concentrated on artefact wiping.

Albano et al. [43] describes a technique for sanitising a phone, removing deleted files. Their technique works by booting a custom recovery image, copying all files to an external storage device, overwriting the entire internal flash memory and copying the files back. The recovery image is a minimal operating system image which was originally intended for performing a complete reset of the phone but which can be replaced on a rooted phone [59]. This procedure will make sure that any data previously deleted will now be irrevocably lost, but has the disadvantage of being an off-line procedure requiring significant time and manual effort.

Another pattern of design for artifact wiping, adopted by several researchers [46, 68, 83], uses an application on an unmodified (or rooted) Android phone to detect the presence of a forensic analysis tool and start deleting data. To trigger the wipe, two methods have been used: reading the system logs [46, 68] and detecting a USB connection [83]. Reading the system logs has the disadvantage of being slow, since it has to wait for the event to occur, the log message to be generated and written and finally read back in before being able to take action.

Detecting a USB connection suffers from lack of specificity, especially in earlier versions of Android. Rouwendal [83] was using version 8 of the Android API (corresponding to Android 2.2 "Froyo" [3]) and was therefore unable to use the improved USB support introduced in API version 12 (Android 3.1 "Honeycomb MR1") and backported to API 10 (Android 2.3.4 "Gingerbread MR1") [9]. His system was therefore forced to use a signal that the phone has entered USB mass storage mode as a trigger. It is unclear what impact, if any, this surrogate had on the result. Both forensic analysis systems he used (Cellebrite and XRY) require the phone to be switched to USB debug mode before it is connected, which disables USB mass storage mode. From a naïve reading, this should have precluded his system from triggering, but his experimental results show that it does trigger. This apparent discrepancy is not discussed in the report.

Regardless of triggering mechanism, the anti-forensic application then has to delete data before the forensic analysis tool can extract it. All papers using this approach are concerned with this time window, reporting measurements of time taken and how much can be overwritten in that time.

2.1.3 Trail obfuscation

In this section, Kessler [64] concentrates on network forensics. When an attacker does not need a reply to a network communication, they can falsify the sender address (so called *address spoofing*) to make tracing the attack to its source harder. It is also possible to use spoofed sender addresses for attack amplification, by tricking third parties into sending much more traffic to a victim than the attacker could on their own. For example, using DNS, it is possible for an attacker to send a 60-byte query packet to the third party, which generates a 4320-byte reply to the victim [94], thereby overloading the victim's data transfer capability much faster than if the attacker had to do it directly.

Other tools in this category are onion routers [35], web proxies [15] and e-mail anonymisers [27], all of which hide the real sender of traffic behind a server which serves many clients.

Finally, this category includes log file and timestamp alteration. Tools such as TimeStomp from the Metasploit framework [25] have long done this on PCs to hide evidence of attacks.

Since Android anti-forensics is mainly concerned with data legitimately stored and usable on the phone, and not with attacks or traces on other devices on the network, this category is not very relevant. However, one piece of research which stands out is that of Albano et al. [42]. They develop an automation system which can be used to make the phone act as if a user is present, thereby providing false evidence that the user was where the phone was at a particular time. They describe several systems based on generally available software automation and testing tools, and finally build one of their own based on recording and playing back user interactions (e.g. touch screen input). For recording events, they connect the phone to a PC and performed the recording over a USB debug connection. Playback is performed either from the recording PC or on the phone from a script file uploaded from the PC. Tests show that this system can be used to post messages to Facebook and send SMS messages. After sending the messages, forensic analyses were performed, which showed no conclusive traces of the automation system on the phone. For their system to work, the phone had to be rooted, and for running without being connected to a PC, a general-purpose scheduling application had to be installed. The researchers hint at, but do not state outright, that the uploaded script is sufficiently obtuse not to be a significant trace. The controlling PC was run entirely in RAM from a Linux live CD, and thus left no traces whatsoever.

2.1.4 Attacks against processes and tools

In this section, Kessler [64] places attacks which force the forensic analyst to perform non-standard procedures or call into question the data recovered.

The procedures used by computer forensic analysts are supposed to follow the public guidelines set by central bodies (e.g. ACPO [53] and NIST [63]). It is therefore relevant to design anti-forensic tools to attack these procedures. An example would be hiding information where only a single, password-

protected program can access it, as in Distefano et al.'s design [51]. In this case, the analyst can choose between following procedure and not getting the data, or rooting the phone, getting the data and violating ACPO principle 1[3].

Smartphones are integrated devices, often necessitating the use of the entire original system in the analysis. This stands in contrast to a PC, which consists of discrete components connected through standard interfaces which can be examined one by one, thereby bypassing some protection[4]. For this reason, the published standards condone much more invasive examinations for mobile phones than for PCs [63, s. 6.5]. However, many of these take time and effort, are specific for each phone model, and require individual testing and, should they be relied on in court, explanation.

2.2 Weaknesses in current approaches

2.2.1 Trusting the operating system

Cellebrite and XRY trust the Android operating system, since they go through it to get the information they want to extract. For unrooted phones this is required, whereas for rooted phones it greatly simplifies the job for the tool.

Android is released under an open source license, and several community projects exist which take advantage of this to build their own versions.

Taken together, these facts suggest that this trust is misplaced, since the user could modify the operating system and make it lie to the forensic tools.

2.2.2 Unspecific triggering

Previous approaches have been in the form of a separate application running on the phone. Due to the security separation which Android enforces on

[3] "No action taken by law enforcement agencies or their agents should change data held on a computer or storage media which may subsequently be relied upon in court." [53]

[4] For example, an operating system asking for a login password does not help against an investigator plugging the hard disk into another computer and reading the data without going through the original operating system.

applications, there is a limit to the amount and timeliness of information available to the anti-forensic application for determining when to trigger. In a modified operating system, the triggering code can be located anywhere in the system (except in the uploaded forensic application itself), making much more detailed information available without having to wait for and parse log messages.

For triggering on the connection of a USB cable, previous approaches have used older versions of the Android operating system which only make available very general information about the USB connection. Contemporary versions allow the triggering code to single out USB debugging connections, which are commonly used by forensics tools but rarely by end users.

2.2.3 Timing constraints

Anti-forensic applications which run separately from the forensics application are limited in how much they can influence its behaviour, again because of the operating system-enforced security separation. It is therefore vital that any manipulations of data are completed before extraction starts.

If the operating system is modified to insert anti-forensics code into the call path of the forensic tool, this constraint disappears, since the code has full control of what data to return to the tool and when. The only remaining timing constraint is on human timescales, when the analyst gives up waiting or the tool determines that an excessive delay means that an error has occurred.

2.2.4 Permanence

Except for Distefano et al. [51], who offer manual restoration of hidden data, all approaches described change the data stored on the phone permanently. While this may be optimal in some cases, it would be desirable to have the option of making non-permanent modifications, so that the legitimate user can restore all data after regaining control of the phone.

Non-permanent modifications also present the option of having different

modifications at different times, thereby denying the forensic analyst repeatability of the examination.

Chapter 3

Android

Android is an open collection of software intended for running on mobile phones. Developed by the Open Handset Alliance [29], it is a complete software stack, with operating system, middleware and end-user applications. Estimates of its share of the smartphone market in 2012 vary between 50% [39] and 60% [47].

At its core, the Android system has a Linux kernel with some Android-specific modifications. The rest of the system is almost entirely Android-specific, with the majority of the code being written in Java and running in the Dalvik [20] virtual machine.

3.1 Content providers

The Android system includes the SQLite database [32] for storing structured data, e.g. the contact list. To simplify the job of each application which wants to use the data, Android wraps each database in an application called a content provider [1]. The content provider provides a simple CRUD[1] interface to other applications, and regular Linux file system access control makes sure that the content provider is the only application allowed to touch the database file directly. This means that the content provider can be solely

[1] An acronym for the four fundamental operations exposed: Create, Read, Update, and Delete [69, p. 381]

responsible for handling security and access control per data item, and variations in the data format. For security, the content provider builds on top of the Android permissions system [87], which decides which applications can access it at all.

A regular Android phone has no means for an application to bypass the Linux file system permission mechanism, so forensics tools must be able to use the content provider interfaces to extract information from the databases on the phone. Therefore, the tools place complete trust in the content providers to provide them with correct data. This is valid on a standard phone, since the same permissions which stop the forensic tool from reading the data directly also stops the user from replacing operating system components, including the content providers. However, by a process known as rooting, the user can bypass the permissions mechanism and replace the operating system on the phone. The content providers can then be replaced with versions containing anti-forensic modifications.

3.2 Contact lists

One of the content providers in a standard Android system is the contact list provider. This allows access to the phone contact list, implemented in CyanogenMod 7[2] as an SQLite database. The Android documentation specifies how to query the content provider for contact list entries, and the format of the returned data.

In a GSM phone, contacts can also be stored on the SIM[3]. These are not directly accessible through the contacts content provider interface. Instead, they are supposed to be imported to the phone's database when the SIM is first inserted. There are also functions for exporting contacts from the phone to the SIM. All this functionality is currently broken in CyanogenMod 7 [19]. For accessing the SIM contact list directly, another content provider[4] exists.

[2] In the file `packages/providers/ContactsProvider/src/com/android/providers/contacts/ContactsProvider2.java`.

[3] Subscriber Identity Module, a smart card containing user information

[4] Implemented in the file `frameworks/base/telephony/java/com/android/internal/telephony/IccProvider.java`.

This is available to programs, but no standard user interface is provided.

3.3 SMS messages

Android has three standard content providers for accessing messages, depending on which type of messages is desired. The SMS provider gives access to SMS messages, the MMS[5] provider gives access to MMS messages, and the MMS+SMS provider handles conversations consisting of both types of messages[6].

SMS messages can be stored on the SIM card as well as on the phone, but unlike SIM contacts there is no interface in Android for directly accessing SMS messages on the SIM card. Instead, the SMS provider itself is responsible for copying messages from the SIM to the phone's database.

3.4 Intents

To communicate between different processes, Android uses a framework called Binder [85] to send messages called *intents* [2]. An intent can be directed to a specific receiver or broadcast, and carries information describing an action to be performed or an event that has occurred. For example, whenever something happens to the phone's USB connection, an intent called `ACTION_USB_STATE` is broadcast, containing e.g. whether the USB cable was connected or disconnected and which mode the phone is in, such as USB mass storage or USB debugging. Any application can register with the Binder system to receive such broadcast messages.

Non-broadcast intents can be either explicit (directed to a named application and component) or implicit (directed to a function). To receive an implicit intent, an application first registers with the Binder system that it wishes to implement a given function. For example, the content provider for

[5] Multimedia Messaging Service, a system for sending messages containing multiple media formats such as text, images, and sound.

[6] These are implemented in files in the directory `packages/providers/TelephonyProvider/src/com/android/providers/telephony/`, named `SmsProvider.java`, `MmsProvider.java`, and `MmsSmsProvider.java`, respectively.

contact lists registers that it provides contacts, and any application wishing to read contact lists will send intents addressed to whichever content provider is currently registered, regardless of its actual name and implementation.

All inter-process communication in Android uses Binder, and Binder provides the name of the calling application to the callee. This means that any application receiving an intent asking for information can check which application is asking.

3.5 Package management

Applications are delivered to Android devices in a format called APK[7] [16]. In addition to the program code, the APK file contains metadata such as a digital signature, requests for permission to access restricted data, and the minimum version of the Android API required for the application to run [8]. The package manager, called “pm”, is responsible for taking this file, processing the metadata and, if no problems occur, install the program. It is also responsible for uninstalling programs, which includes deleting the contents of their private directories.

3.6 Rooting

Android uses the Linux kernel, which includes the standard UNIX discretionary access control system. In this system, the superuser, called *root*, has complete access and can bypass all privilege checks. Normally, Android does not allow a physical user access to the root account. However, using security vulnerabilities in the standard Android system, it is possible to make the system run code as root for the user. This code can then set up more convenient methods for the user to gain root access later. This process is called *rooting* the phone. Some phones aimed at developers, such as the Google/Samsung Nexus S [71], have a built-in switch for allowing root access, removing the need for exploiting security vulnerabilities.

[7] Application Package

Once root access has been obtained, the user is in complete control over the phone. In particular, they can replace the operating system with a custom version of Android. This is accomplished by replacing the recovery image [59], an alternate firmware present in all Android devices which the standard Android bootloader can boot instead of the main Android system. The standard recovery image allows very few operations, such as resetting the device to factory defaults and installing approved operating system updates. Alternate recovery images, such as ClockworkMod [17], allow additional operations such as taking complete system backups and replacing the entire operating system from a ZIP file on the SD card. Installing a modified version of Android is accomplished by storing the new version in a ZIP file on the SD card, booting the recovery image, choosing to install a new operating system, and selecting the ZIP file.

A few Android devices allow the user to replace the recovery image without first rooting the phone, using a method called *fastboot* [58]. When in fastboot mode, instead of booting from the built-in memory, the phone waits for commands from a PC attached via USB. The accepted commands include one for writing a new recovery image to the phone. However, only a few old phone models and models targeted towards developers enable fastboot by default [65]. For the majority of Android phones, using the fastboot method would require first rooting the phone and using the elevated privilege to install a new bootloader which allows fastboot.

3.7 Community distrubutions of Android

In order to make Android a successful platform for diverse devices, Google made it open source [74]. In particular, they chose to license the source code for the system under the Apache License 2.0 [44] as far as possible [73]. This license is approved as an open source license by the Open Source Initiative [89] and as a free software license by the Free Software Foundation [88]. The Android Open Source Project [11] distributes this source code and provides compliance test suites for ensuring that any modifications still meet the Android specifications.

Several projects have built upon this source code to create their own distributions, which users can install on their own Android devices. Examples of such projects include Replicant [81], a community effort to remove the few modules with non-free licenses still existing in Android; MIUI [26], developed by Chinese phone manufacturer Xiaomi and installed as default on one of their phones [66]; and CyanogenMod [18], which is "designed to increase performance and reliability over Android-based ROMs released by vendors and carriers" and "[offer] a variety of features & enhancements that are not currently found in these versions of Android" [77]. Since these distributions replace operating system components, the phone must first be rooted in order for the user to be able to install them.

As of June 2012, CyanogenMod claims to have been installed and run at least once on over 2.4 million different devices [78] where the user has actively opted in to being counted [90]. For comparison, 331 million Android devices had been activated as of the first quarter of 2012 [47]. However, neither figure considers deactivated devices. A possibly more relevant figure is the rate of new activations. Google claims they activate more than 900 000 devices per day as of June 2012, while CyanogenMod claims almost 15 000 new installations per day as of July 2012 [79].

CyanogenMod claims official support for 63 different devices, with an unspecified number of others being unstable or having incomplete support [80]. There are therefore large both actual and potential user bases for modifications built on top of CyanogenMod.

Chapter 4

Methodology

This dissertation aims to examine the potential for operating system-level anti-forensics on Android. The hypothesis is that *it is possible to modify the Android operating system to present false information to the forensics tools Cellebrite and XRY*. Several subsidiary research questions need to be answered in order to explore the hypothesis:

- Which components of the Android operating system do the forensics tools trust?
- Is it possible to modify these components to present false information?
- Can the presence of a forensic analysis tool be detected?
- Is it possible to make the presentation of false information reversible, such that the phone will revert to presenting the real information after the forensic analysis?

The equipment used will be an HTC Desire phone running the Cyanogen-Mod distribution of Android, analysed by the two forensics tools Cellebrite and XRY. See appendix A for details.

An unrooted phone requires the use of content providers in order to access application-private data. Assuming that the forensics tools also use the same method, the content providers will be instrumented to find if they are used by each tool, and if so how. If they are, the content providers will be modified

to return false data to the tools. To test the modifications, the forensics tools will be used to extract data from the phone while running the modified software, and the results compared to the ones from an unmodified phone.

For analysing the behaviour of the forensics tools, and for providing real data to be hidden during experiments, a dataset will be entered into the phone. The phone contact list will be entered using the standard on-board tools, and will have two entries, each with a name and a phone number. It will not be synchronised with any other service. The SIM contact list will contain two entries, which are entered by default by the service provider. SMS messages will be sent from another phone to the experiment phone.

4.1 Scope

The work produced for this dissertation is intended to be a proof of concept. As such, only a single category of forensically interesting data will be considered in depth, with others touched upon lightly in order to ascertain whether the method can be generalised.

There are several content providers in a standard Android phone. Of these, the contact list is of high value to a forensic analysis, and thus a prime target for anti-forensics. The contact list is stored only on the phone and other devices the phone's owner has chosen to synchronise it with, unlike call logs and SMS messages which can be retrieved from the SIM card or the network provider. It is therefore a good target for anti-forensics locally on the phone. The contact list also has a history of being used as a target for proof of concept implementations. Previous work [46, 83] has used erasing or overwriting contact list entries as a measure of the performance of anti-forensic tools. Due to these factors, the phone contact list will be selected as the primary target for anti-forensics.

The other targets will be SIM contacts and SMS messages. They are both potentially available external to the phone, in the SIM. The SIM can be removed by the investigator and read separately, thus negating any anti-forensic measures taken on the phone. However, this may be undesirable, e.g. if the SIM is PIN-locked and there are exigencies preventing the acquisition

of the PUK[1] from the service provider. Also, the SIM only has limited storage for a small, fixed number of messages, so other messages would have to be retrieved from the phone. There is therefore a point in performing anti-forensics on these items on the phone. Furthermore, the difference in back-end data storage between the contact list and these SIM data items should provide a test of the generality of the method.

4.2 Limitations

4.2.1 Triggers for anti-forensic behaviour

The anti-forensics system needs to be able to distinguish between a request for data made by the legitimate user and one made for the purposes of forensic examination, in order to determine which set of data to return. This is, of course, impossible in the general case. However, we can make assumptions about how the forensic examination is carried out, and base the trigger on these assumptions. Should this prove inadequate, new trigger mechanisms can be designed from the observed behaviour of forensic investigators.

Based on the recommendations in the ACPO Good Practice Guide for Computer-Based Electronic Evidence [53], an examination can be assumed to be carried out in the following steps:

1. Seizure.
2. (Optional) Connection to a portable power supply for transport to the forensics lab.
3. Network cut-off, using e.g. a Faraday cage.
4. Transport to a forensics lab.
5. Enabling of debug mode (required for the analysis system to be able to upload its application to the phone).

[1]PIN Unlock Key or Personal Unlock Key, a code which can be used to reset the SIM PIN code

6. Connection to a forensic analysis system (e.g. Cellebrite or XRY).
7. Data extraction using the forensic analysis system.
8. Disconnection from the forensic analysis system.
9. Disabling of debug mode.
10. Manual check of data on the phone, for verification of data extracted by the forensic analysis system.

Since we assume that the first time the information on the phone is accessed is by the forensic analysis system, detecting it presents a good opportunity to trigger the serving of false data. To accommodate the manual verification after extraction, there should be a delay between the forensic analysis system disconnecting and the real data becoming available.

While a USB device (such as a phone) must identify itself to its host (such as a forensic analysis system), the device does not learn anything about the host [54, part 5]. This means that the phone cannot distinguish the connection of a forensic analysis system from the connection of a PC. However, Cellebrite and XRY (and, according to Azadegan et al. [46], also Paraben Device Seizure [50] and Susteen Secure View [61]) work by uploading an application to the phone over Android's standard USB debug interface [10] and having this application export the data, also over the USB debug interface.

Accordingly, the first trigger used is USB debugging itself. It serves as a fallback option, in case the specific triggers for each forensic analysis system miss, perhaps due to changes in the systems since their signatures were recorded. The end of USB debugging is used to determine when examination has completed, and thus when to start the timer for stopping the serving of fake data.

More specific triggers will be developed by instrumenting the contacts provider and studying the system logs to determine exactly how the different forensic analysis systems query the contact list.

4.2.2 Visibility

The code of the anti-forensics system will be implemented as changes to the default CyanogenMod content provider for contact lists. This means that CyanogenMod has to be installed on the device before the anti-forensics system can be used. CyanogenMod is visibly different from a standard Android distribution, using different branding, standard applications, etc. However, CyanogenMod is installed on a significant number of devices, so its presence is not necessarily suspicious in and of itself.

As the system is a modification of a standard component of the operating system, no indication of anti-forensics will exist in the form of extra applications. For the versions of the system where the decoy data is hardcoded, no extra data will be present in the file system either. To determine the nature of the anti-forensics system, a forensics analyst would have to look at all operating system components, determine that the contacts provider is non-standard, and reverse engineer the changes. The reverse engineering step would be helped by the free availability of Java decompilers, such as JD [52].

For the versions of the system which use external databases to provide decoy data, this decoy data will be present in the file system. However, it resides in files which do not exist in a standard Android system, in a directory which is unreadable for forensics tools on a standard Android system. The analyst would thus have to actively search for them to find them.

4.2.3 File system access

Android content providers use Linux file system permissions to limit direct file access to the database to only the content provider itself. This means that, in a standard Android system, it would be impossible for a forensics tool to read the database file directly. Therefore, Cellebrite and XRY in standard modes do not attempt to do so when extracting the contact list, but use the content provider interface.

The anti-forensics system relies on being able to replace a system application, which is normally protected from the user by the phone manufacturer.

To install it, then, requires the circumvention of this protection, a process known as rooting the phone. This process gives the user the ability to run applications as root, the Linux superuser, which is able to bypass all restrictions.

A forensics tool can take advantage of this elevated privilege to read data directly from the phone's file system, without going through the content providers. Both Cellebrite and XRY offer this functionality. In the case where the decoy data is hardcoded in the application, this would enable the tool to read the database file and extract the real data. When the decoy data resides in databases on the file system, the tool can read all databases and present them to the analyst. However, in order to use this functionality the analyst would first have to suspect that the standard analysis was faulty, and then manually examine the extracted data files.

There is no inherent technical reason for keeping the capability to run applications as root after the system application has been replaced and the anti-forensics system is operational. For a stronger defence against forensic examination the user could either revoke root access permanently (thereby denying even themselves the possibility of further changes), or make it require a customised series of operations (thereby making automated tools unable to use it). This is discussed further as future work.

If a forensic analysis tool can completely bypass the Android operating system, the anti-forensics discussed in this dissertation will have no effect. The forensic recovery image developed by Vidas et al. [95] is such a tool. It installs and boots a minimalistic alternate operating system on the phone, bypassing the phone's main operating system. This minimalistic operating system can then read the phone's built-in memory without interference. However, neither Cellebrite nor XRY use the technique of recovery image replacement over fastboot developed for that image. Drawbacks of this technique include requiring a specific recovery image for each phone model, modern consumer phones requiring rooting before fastboot will work, and the lack of help from the Android operating system in interpreting the extracted data.

4.3 Experimental design

The experiments will be divided into two stages, examination of current behaviour and implementation of anti-forensic modifications. The implementation is described in chapter 5, and the source code is in appendix C.

4.3.1 Examination of current behaviour

Before any anti-forensics can be implemented, it is necessary to determine if and how the operating system can be modified, and how the operating system and the forensics tools currently behave. This will be accomplished through the following steps:

Step 1 To investigate the possibility of operating system modifications, the source code for the CyanogenMod community distribution of Android will be downloaded, built and installed according to the CyanogenMod project's instructions [23, 24].

Step 2 Once the phone is running this version of CyanogenMod, modifications will be introduced to trace the behaviour of the forensics tools. Content providers are applications that wrap databases on the phone, performing security checks and format conversions as required by the Android specifications. On the assumption that both Cellebrite and XRY use content providers to access data on the phone, these modifications will take the form of altering the content providers to provide logs of how they are called.

Initial investigation will be limited to the content provider responsible for the phone's contact list. Once data is available from that content provider, the techniques developed will be extended to cover SIM contacts and SMS messages, to prove that they can be generalised.

Step 3 From knowing how the calls are made, the Android documentation will be consulted for information on the format of the returned data for each call.

Step 4 According to the documentation for Cellebrite and XRY, they both use a USB debugging connection to connect to and extract data from Android phones. To determine what information is available for triggering anti-forensic behaviour, a separate application will be developed that produces logs of the events seen by the operating system when this connection is established and severed.

4.3.2 Implementation of anti-forensic modifications

The data acquired from the previous stage will help create an understanding of how Cellebrite and XRY call the content providers. Using this information, anti-forensic modifications will be implemented in the following steps:

Step 1 The content providers will be modified to recognise when they are being called by forensics tools. This will take into account the information from the separate program on how to recognise USB debugging, and behaviour specific to the tools.

Step 2 Modifications will be developed to make the content providers exhibit anti-forensic behaviour when they detect the presence of the forensics tools, but still be sufficiently close to the original behaviour for the forensics tools to believe in the data they receive.

The anti-forensic behaviours will be delaying before returning any data, returning no data, returning data hardcoded in the content provider, and returning data from an alternate database. The full range of anti-forensics will be implemented for the phone's contact list. To prove that the techniques can be generalised, modifications will be introduced to return no data for queries for the SIM contact list and SMS messages.

When anti-forensics are used, the real data should not be extracted, the intended false data should be extracted and no errors should be reported.

Step 3 The package manager will be modified to detect attempts by the forensics tools to install their applications on the phone, and reject the

installation. In this case, the tools will be permitted to report errors to the forensic analyst, but no data should be returned.

Step 4 For testing, the phone will be prepared with real data in the form of contact list entries and SMS messages. Forensic extractions will be performed using both Cellebrite and XRY, first with the phone running an unmodified CyanogenMod operating system and then with each anti-forensic modification in turn. The results of the extractions will be inspected for the real data, the false data for that anti-forensic case and any signs of the tools suspecting that something is wrong, according to the test criteria in steps 2 and 3.

Chapter 5

Implementation, testing and results

The experimental work for this dissertation consists of the three stages of examination, implementation and testing. Examination finds where and how the forensics tools find their information. Implementation uses the information gathered to modify the parts of the operating system that the tools rely on, in order to invalidate their results. Testing checks whether the tools recover the real or the false information when the modifications are active.

5.1 Examination of forensics tools

The content provider interface is the only way to gain access to information such as the contact list on an unrooted Android phone. As such, forensics tools need to use this interface at least when dealing with unrooted phones. On a rooted phone, it would be possible for a forensics tool to bypass the content provider and read directly from the database, but this would require the tool to first find and interpret the database.

Under the assumption that the content provider is always used, instrumentation code was inserted into it to write information about its behaviour to the system log. For each call to the content provider's main query function, the code output the name of the calling program, the query arguments

and which part of the existing program logic handled the query. The source code is in appendix C.2.

5.1.1 Cellebrite

Cellebrite uses the content provider in all observed cases. It starts by making seven queries to the `raw_contacts` and `settings` modules, collecting general information such as the number of contacts and whether contacts are marked as deleted. It then goes through the `raw_contacts` module, querying for information on each contact. For each contact, nine queries are made for different kinds of information associated with it (name, phone number, e-mail address, etc.). The full log is in appendix B.1. One of the kinds of information, "dispatch_v2", is not documented in the Android developers' manual and no such query has been seen to return information. Its purpose is therefore currently unknown. The extracted contact list, seen in figure 5.1, matched that entered and seen in the phone's built-in contact list application.

Phone Contacts

Total Entries: 2
PBB MD5 Hash: 220948A440AA43A78294A7733129A1A9
PBB SHA256 Hash: EE48FC18 F6C3FCD DF06AC0 F3C1F81 D66EEF0 B9EFF54 0503A3B 801B32A 016044A

#1	**University of Glasgow** (Memory: Phone)
Work:	+441413302000

#2	**Houses of Parliament** (Memory: Phone)
Work:	+442072193000

Figure 5.1: Contacts extracted by Cellebrite from an unmodified phone

5.1.2 XRY

XRY was also observed to use the content provider in all cases. It makes only two queries in total, retrieving an entire data module with each query.

The two modules are `raw_contacts` and `data`. It is possible for XRY to interpret this data using information found in the Android API reference manual [4]. The full log is in appendix B.2. The extracted contact list, seen in figure 5.2, matched that entered and seen in the phone's built-in contact list application.

Importance	Name	Index	Work
	University of Glasgow	1	+441413302000
	Houses of Parliament	2	+442072193000

Figure 5.2: Contacts extracted by XRY from an unmodified phone

5.2 Triggering

Previous work has triggered anti-forensic behaviour either on finding log entries relating to the installation of forensics tools or on connection of a USB cable. Reading logs requires waiting for the log messages to show up and then spending effort reading and interpreting them, while USB connections are frequent in everyday use.

One attribute common to all examined forensics tools is that they require the phone to be set to USB debug mode. This lets the tool control the phone and e.g. install applications. Setting the phone to debug mode requires going deep into the settings menu and acknowledging a warning that "USB debugging is intended for development purposes only" (see figure 5.3). It also removes the possibility to use the phone as a USB memory and transfer files to and from it using standard file management tools. The fact that activating debug mode removes these normal and desirable features suggests that most users will not have debug mode activated. If so, triggering anti-forensic behaviour upon activation of USB debugging will have a lower false positive rate than triggering on all USB connections.

A stand-alone application will be built to be a receiver for the intent broadcast when the USB state of the phone is changed. Using the extra information contained in the intent, the program determines whether the

Figure 5.3: Warning shown when enabling USB debugging

USB cable is connected or not, and whether the phone is in debugging mode. The source code for this application is in appendix C.1.

5.3 Anti-forensics modules

Based on the examination of the forensics tools and of the USB triggering, several anti-forensics modules will be constructed. Some of the modules will depend on previous ones, and some will be self-contained.

5.3.1 Response delays

Anti-forensic applications which operate outside of the call path of the application uploaded to the phone by the forensics tool have only a small window of time to do their work, between detecting the forensic application and the extraction being completed. For example, Azadegan et al. [46] reported a gap of several seconds between the installation of the forensic application and the start of extraction, while Rouwendal [83] reported being able to make

200–500 small writes in the time from the USB cable was inserted until the forensic extraction was completed.

On the other hand, modifications within the call path, such as those described here, have much more time available, since they can make sure not to return any results to the forensics tool until the anti-forensic work is completed. Any time constraints would come from the tool or the human operator deciding that the delay is excessive, perhaps speculating that the forensics tool or its connection to the phone has malfunctioned. This would only have an impact on the ability of the anti-forensics system to return false data, since a process to delete or hide data can keep going while the operator retries the extraction.

Even if it would be unnecessary for completing the anti-forensic work, a delay on human timescales could still be useful for wasting the investigator's time. Kessler [64] calls this "time-sensitive anti-forensics". Computer criminals have talked about a "17-hour rule" [48]. It works on the principle that, since computer forensics investigators are constantly overloaded with work, it can be assumed that no more than two working days (16 man-hours) will be spent on a routine investigation. Therefore, if the anti-forensic measures hold up for 17 hours, they will have succeeded. If a significant amount of this time can be taken up with artificial delays, less time is available for defeating other measures, which can then be simpler. This is similar to the concept of a tarpit in e-mail spam prevention [34].

The delay should be long enough to incur significant problems for the forensic analyst (i.e. it should not be enough to run the extraction overnight), but should still trickle-feed data at a rate high enough to keep the tool and the analyst interested in continuing the extraction. It must also take into account Android's built-in "Application Not Responding" system [7], which may alert the analyst to the fact that a delay is used.

Building upon the triggering code, delays will be inserted at the beginning of the `query()` function. As was seen when examining the behaviour of the forensics tools, Cellebrite makes several initial queries and then several more per contact, while XRY makes only two queries for the entire database. If trickle-feeding is desired, this places sharp limits on the number of delays

and the length of each delay. The source code is in appendix C.4.

5.3.2 Rejecting installation of forensics tools

If the presence of an application uploaded from a forensics tool can be detected before it is started, it should be possible for the phone to refuse to run the tool altogether. This could be disguised as one of several legitimate problems which could arise from the installation process, such as the application being incompatible with the version of the Android operating system used on the phone, or there not being enough memory available to install the application.

The package manager is the primary (and, on a non-rooted phone, only possible) avenue through which programs are installed on an Android system. Dumps of the USB communication between forensics tools and phones confirm that the package manager is used to install the applications used by the tools to extract data from the phone [46]. In all observed cases during these experiments, Cellebrite and XRY have used the same names for their respective applications. Therefore, the package manager will be modified to check the name of each package being installed and refuse to perform the installation if the name matches one of the forensics tools. The source code is in appendix C.3.

5.3.3 Hardcoded false contact list

Finding no data in a well-used phone would be suspicious. Arranging for the analyst to find plausible, but non-incriminating, data instead increases the chance of them accepting the data as given and concluding that the phone holds no relevant data.

In case this deception fails and a thorough analysis of the phone is performed, it should be as hard as possible to determine where the false data came from, in order to cast doubt on the analysis process. Hiding the data in program code makes it harder to find than if it is in a separate file. String obfuscation techniques, well-known from malware analysis, can be added to

make the data impossible to spot in the code without significant reverse engineering effort. Such obfuscation methods can range from simple bitwise logical operations [91] to customised encryption algorithms [33, 56].

In the examination of the behaviour of the forensics tools, sufficient data was obtained to determine the format they expect for the extracted contact list. Using that information, false data of the correct formats will be constructed and inserted into the code for the contacts provider, to be returned in response to queries from each tool. As a fallback, the USB triggering work will be incorporated as well. If a query comes from an unknown tool, but USB debugging is on, no results will be returned. The source code is in appendix C.5.

5.3.4 False contact list from alternate databases

Hardcoded data is optimised for hiding, but in order to change it the user would have to edit, recompile and reinstall the contacts provider. If the false data instead resides in an alternate database, it can more easily be customised by each user.

This would leave the false data on disk in the same format as the real data, which makes it easier to spot than if it was hardcoded. As an additional precaution, the file names should be switched, so that the file which normally contains real data now contains false, and vice versa. That way, even if the analyst succeeds in making a logical or physical dump of the phone's file system, any tools and experience would still lead them to the false data.

The module for returning hardcoded false data already contains code for answering queries differently depending on which forensics tool made the query. The false response code will be modified to read its data from alternate databases instead of having it hardcoded, one database for Cellebrite and one for XRY. The source code is in appendix C.6.

5.3.5 Delayed restoration

After an analyst has performed an extraction of data from a device, they may want to perform an independent verification. This may be done by using

another tool to perform an extraction, or by manually using the phone's built-in functionality. To preserve the illusion that the false data reported to the first tool is true, subsequent examinations should yield the same results.

To this end, a timer will be added to the triggering code in the module reading false responses from alternate databases. The removal of the USB cable will start the timer, and the same false data will continue to be returned until the timer expires. Subsequent insertions of USB cables will reset the timer, so that the same false data will continue to be delivered to other forensics tools. The source code is in appendix C.7.

5.3.6 Hiding SIM contacts

The previous modules for returning false data only work with the phone's internal contact list. The SIM also contains a contact list. In the standard Android operating system, there is no way to directly manipulate these contacts, but they can be imported to the phone's contact list in bulk, and forensics tools can read them directly from the SIM.

A module for hiding SIM contacts will be included to ensure that all contacts on the phone are hidden. It will also provide a demonstration of the generality of the method used, since the content provider for SIM contacts is not the same as the one for phone contacts, and uses a different data storage mechanism. The source code is in appendix C.8.

5.3.7 Hiding SMS messages

Like contacts, SMS messages are stored both on the SIM and in the phone. Unlike contacts, Android doesn't provide a way to access SMS messages on the SIM independent from those on the phone. All code for handling SMS messages is in the same content provider, which fetches messages from the SIM to the phone as required.

SMS messages are pushed to the SIM from the network, meaning that one could arrive in the middle of an investigation. Best practise guidelines recommend isolating the phone from the network to prevent this. However, it is possible that the guidelines aren't followed, or that the analyst hasn't

had time to isolate the phone before the message arrives. In these cases, it would be desirable to hide the fact that a message has been received, to avoid giving the investigator an incentive to perform a detailed analysis.

There are two content providers in Android which allow access to SMS messages, called "SmsProvider" and "MmsSmsProvider". The SMS provider only returns SMS messages, and is the one used by Cellebrite and XRY. The MMS and SMS provider returns full conversations spanning both SMS and MMS messages, and is used by the built-in messaging application. To make sure SMS messages are hidden from all viewers, both will be modified. USB debug triggering will be implemented, and `query()` function of each will be modified to return no result when USB debugging is active. As a proof of concept, only hiding of messages will be implemented. Should false messages be desired, the techniques for returning false contact lists could be adapted to SMS messages. The source code is in appendix C.9.

5.4 Testing and results

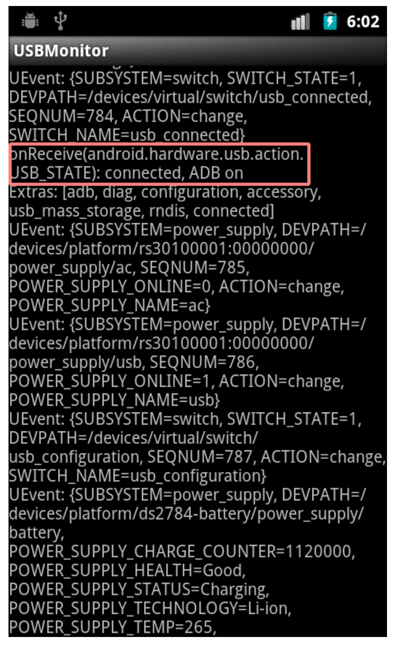
For each test of an anti-forensic module, the operating system was rebuilt to contain the module, uploaded to the phone and installed using the recovery image. This replaced the operating system already on the phone, but preserved user data such as contact lists and SMS messages. Each modified version of CyanogenMod was approximately 90 MB in size and took slightly less than one minute to install.

5.4.1 Triggering

The phone will be reinstalled using a standard CyanogenMod 7.2 operating system. The monitoring application will be installed and started. With the monitoring application running, the USB cable will be plugged and unplugged with the phone in both USB mass storage and USB debugging modes. The messages printed by the application should match the state of the USB cable and settings.

Two screenshots from the running application are in figure 5.4. In 5.4a,

the phone is in USB debugging mode, and in 5.4b it is not. Both images contain mostly raw information, so the processed information indicating USB connection and debugger state[1] has been indicated with outlines. They are different and match the physical actions performed, so the program can distinguish between the different cases.



(a) USB connected, debugging on

(b) USB connected, debugging off

Figure 5.4: Screenshots of USBMonitor output

5.4.2 Response delays

The phone will be reinstalled using a modified CyanogenMod 7.2 operating system. The only modification from default will be the insertion of a delay into each query of the contact list provider. Extractions of the contact list will be performed using Cellebrite and XRY, and the delay increased until the tools present errors instead of performing successful extractions.

Cellebrite has a low tolerance for response delays. It accepts a delay of five seconds at each call to `query()`, but ten seconds is enough to make it

[1] The debugger is indicated by its name "ADB", which stands for "Android Debug Bridge" [13].

abort the extraction and show the error message in figure 5.5. When this error happens, no report is created and no data is extracted, even if other information on the phone could have been extracted successfully. Adding a five second delay to each call lengthens the extraction time from approximately 25 seconds for an unmodified phone (figure 5.6a) to approximately 2 minutes 20 seconds (figure 5.6b).

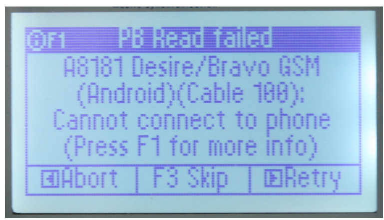


Figure 5.5: Error from Cellebrite with a ten second response delay

Extraction start date/time:	10/08/12 14:46:09
Extraction end date/time:	10/08/12 14:46:33

(a) No delays

Extraction start date/time:	10/08/12 14:10:38
Extraction end date/time:	10/08/12 14:13:00

(b) Delay 5 seconds per call to `query()`

Figure 5.6: Cellebrite extraction logs showing elapsed time

No upper bound has been found for the delay tolerated by XRY. With a six hour delay for each of the two calls, XRY took the expected twelve hours to complete the extraction of the contact list. No error message was given, but the times were recorded in the log (figure 5.7).

18	ANDROID	Success	15:34:14	Reading Contacts
19	ANDROID	Success	03:34:15	Reading Calls

Figure 5.7: Extraction log from XRY with a six hour response delay

5.4.3 Rejecting installation of forensics tools

The phone will be reinstalled using a modified CyanogenMod 7.2 operating system. The only difference from the standard code will be the modifications to the package manager for rejecting installation of applications named "com.client.appA" or "example.helloandroid". Extractions of the contact list will be performed using Cellebrite and XRY, and the extraction result compared to that from an unmodified CyanogenMod.

Cellebrite completely failed to perform the extraction, instead presenting the error message in figure 5.8 to the user.

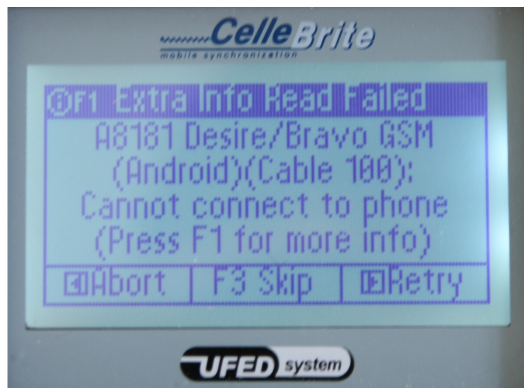


Figure 5.8: Error from Cellebrite when installation was rejected

XRY completed the extraction. However, the summary (figure 5.9a) reports that an error has occurred and that the extraction was incomplete. An error was also reported in the log (figure 5.9b), but this was not particularly clear on what went wrong and what the consequences were. The report was missing the sections "Device/App Usage", "Contacts" and "Web/Bookmarks".

5.4.4 Hardcoded false contact list

The phone will be reinstalled using a standard CyanogenMod 7.2 operating system. Using the built-in contact list application, two contacts will be entered into the phone contact list (see figure 5.10). Extractions of the contact list will be performed using Cellebrite and XRY, and both should extract

Logical Extraction finished with errors

XRY encountered one or more errors during the extraction of data from the device.
Data may not be complete.

Folder	Items
Device / General Information	1
Files / Unrecognized	12

(a) Extraction summary

7	MAIN	Success	11:45:58	Processing device [HTC Desire A8181] connected to DummyPort []...
8	MAIN	Success	11:45:58	Starting process of ANDROID (6.1.1)
9	ANDROID	Success	11:45:58	Connecting
10	ANDROID	Success	11:46:08	Connected
11	ANDROID	Unsuccessful	11:46:08	Receive packet failed
12	ANDROID	Success	11:46:12	Device is rooted. Application data will be extracted.
13	ANDROID	Success	11:46:14	Extracting email data.
14	ANDROID	Success	11:46:18	Extracting Google Talk data.
15	ANDROID	Success	11:46:30	Disconnecting
16	MAIN	Unsuccessful	11:46:35	ANDROID (6.1.1) completed with error
17	MAIN	Success	11:46:35	Starting process of DISKSTOR (6.1.1)

(b) Extraction log

Figure 5.9: Results from XRY when installation was rejected

both contacts.

The contact list database will be copied from the phone to a PC. The phone will be reinstalled using a modified CyanogenMod 7.2, the only difference from the standard operating system being the contact list provider, which will contain hardcoded false data. Three sets of false data will be provided. Cellebrite and XRY will be provided with contact lists containing one contact each, this being the technical support phone number for each tool. Unknown tools will be provided with an empty contact list.

Cellebrite and XRY will be used, in turn, to extract the phone contact list. They should see the same number of contacts as in the real contact list, but each should have the name and number to technical support for the respective tool. Finally, the phone will be connected to a PC, still in USB debugging mode, and the contact list inspected using the built-in application. No contacts should be visible.

Both Cellebrite (figure 5.11) and XRY (figure 5.12) showed the two con-

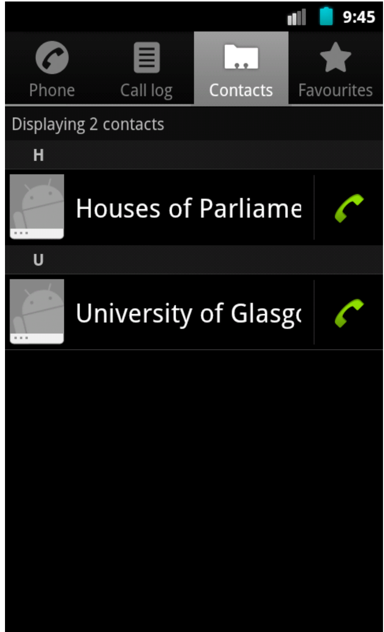


Figure 5.10: Contact list application with real data

tacts with their respective false data. When connected to a PC in USB debugging mode, the built-in application shows an empty contact list, as seen in figure 5.13.

5.4.5 False contact list from alternate databases

The phone will be reinstalled using a modified CyanogenMod 7.2 operating system. The contact list provider will have been modified to retrieve its data from different databases depending on whether the query comes from Cellebrite, XRY or the phone itself outside of USB debugging mode.

When the contact list was entered manually in the previous test, the phone software will have created an SQLite database file on the phone to store it[2]. This file will be copied to a PC in two instances, which will be changed to contain the technical support phone numbers for Cellebrite and XRY, respectively. These changed database files will be uploaded to the phone to be used by the modified contact list provider[3].

[2] `/data/data/com.android.providers.contacts/databases/contacts2.db`
[3] `/data/data/com.android.providers.contacts/databases/cellebrite.db` and

Phone Contacts

Total Entries: 2
PBB MD5 Hash: A5F38C59FC9096D899352CEED848AF0F
PBB SHA256 Hash: 63BFC342 9078845 71A1907 2490120 D9CA979 83AB5D2 2BB3812 6A97718 8D08CCA

#1	**Cellebrite Technical Support** (Memory: Phone)
Work:	+495251546490

#2	**Cellebrite Technical Support** (Memory: Phone)
Work:	+495251546490

Figure 5.11: Cellebrite extraction report showing hardcoded false data

Importance	Name	Index	Work
	XRY Technical Support	1	+4687390270
	XRY Technical Support	2	+4687390270

Figure 5.12: XRY extraction report showing hardcoded false data

Extractions of the phone contact list will be performed using Cellebrite and XRY. They should not see the two contacts in the real contact list, but only their own technical support phone numbers.

Both Cellebrite (figure 5.14) and XRY (figure 5.15) saw only their own single technical support contact.

5.4.6 Delayed restoration

The phone will be reinstalled using a modified CyanogenMod 7.2 operating system. The contact list provider will have been modified to return results from alternate databases, as in the previous test. In addition, the provider will be modified so as to continue returning the same false contact list until the USB cable has been unplugged for thirty seconds, regardless of which tool requests it.

/data/data/com.android.providers.contacts/databases/xry.db

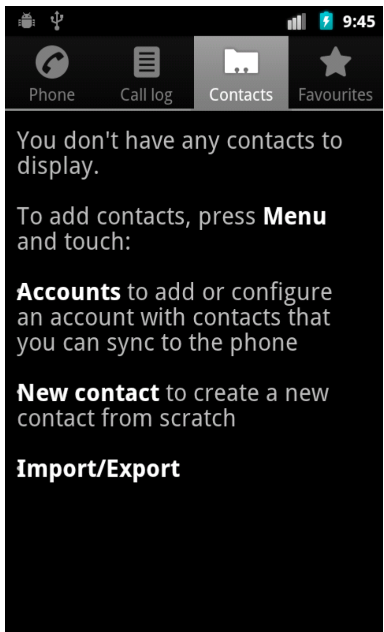


Figure 5.13: Contact list application showing empty contact list when USB debugging is on

Cellebrite will be used to perform an extraction of the phone contact list. This should contain only the number for Cellebrite technical support. Another extraction will be performed using XRY. This should still show the same number for Cellebrite technical support. The phone will be disconnected from the USB cable and the contact list inspected using the built-in contact list application. This should also only show the number for Cellebrite technical support. Thirty seconds should be allowed to pass, and the built-in contact list application opened again. This should now show the two contacts in the real contact list.

The Cellebrite extraction report showed only the false contact intended for Cellebrite (figure 5.16a). The contact list application on the phone saw the same information (figure 5.16b), and so did XRY (figure 5.16c). After disconnecting the phone and waiting thirty seconds, the contact list application on the phone saw the two contacts in the real contact list.

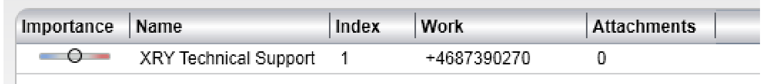
Beginning a new series of extractions with XRY, the XRY extraction report showed only the false contact for XRY (figure 5.17a). The built-in

Phone Contacts

Total Entries: 1
PBB MD5 Hash: 324B323C2612C3A2C685F72FB6C5641D
PBB SHA256 Hash: 34815D95 B25C782 B6387DF 426C414
DF75C2F 657167A 68BD636 DD1C085 8359767

#1	**Cellebrite Technical Support** (Memory: Phone)
Work:	+495251546490

Figure 5.14: Contacts fed to Cellebrite from an alternate database



Importance	Name	Index	Work	Attachments
	XRY Technical Support	1	+4687390270	0

Figure 5.15: Contacts fed to XRY from an alternate database

contact list application (figure 5.17b) and Cellebrite (figure 5.17c) showed the same contact. After disconnecting the phone from the USB cable and waiting thirty seconds, the built-in application again showed the real contact list.

5.4.7 Hiding SIM contacts

The phone will have a SIM for a pay-as-you-go account from Lebara. This contains two contacts by default, one for topping up and one for voicemail. The phone will be reinstalled using a standard CyanogenMod 7.2, and extractions of the SIM contact list performed using Cellebrite and XRY. Both tools should extract both contacts from the SIM.

The phone will be reinstalled using a modified CyanogenMod 7.2 operating system. The only modification will be to the SIM contacts provider, which will be changed to return no contacts when the phone is in USB debugging mode. Extractions of SIM contacts will be performed again using Cellebrite and XRY. Neither should be able to find any contact.

On an unmodified phone, Cellebrite found two phone contacts and two SIM contacts (figure 5.1). With the SIM contact hiding active, only the two

phone contacts were visible (figure 5.18b).

XRY does not have functionality for reading data from the SIM through the phone. It requires the SIM to be removed and inserted into an external reader. Therefore, SIM anti-forensics on the phone have no effect on XRY.

5.4.8 Hiding SMS messages

The phone will be reinstalled using a modified CyanogenMod 7.2 operating system. The only modification will be to the SMS provider, which will be changed to return no SMS messages when the phone is in USB debugging mode.

Before the phone is connected to any analysis tool, an SMS message will be sent to it. This should produce a notification and be visible in the SMS application. Then, the phone will be connected to each of Cellebrite and XRY in turn, and extractions of SMS messages performed. No messages should be extracted. While the phone is still connected to each tool, another SMS will be sent to it. This should not produce a notification and should not be visible to the user. The built-in messaging application should show no messages while the phone is connected using USB debugging.

No notifications were produced when messages were sent to the phone while it was connected to the analysis tools. After the phone was disconnected from the tools, the messages were available through the standard phone application.

Cellebrite found no SMS messages when the anti-forensics was enabled, as seen in figure 5.19b. The message "SMS Information Not Available" seems to be an error message. However, it is also present when the phone is running standard CyanogenMod and legitimately has no SMS messages stored.

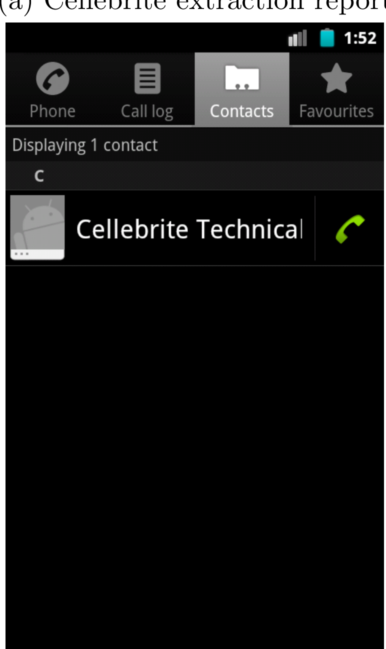
XRY also did not find any SMS messages when the anti-forensics was enabled. The extraction report from XRY does not detail which pieces of information were sought, only which were returned, so the report simply lacks a section for SMS messages. The built-in messaging application showed no messages while the phone's USB debugging connection was active (figure 5.20b).

Phone Contacts

Total Entries: 1
PBB MD5 Hash: 324B323C2612C3A2C685F72FB6C5641D
PBB SHA256 Hash: 34815D95 B25C782 B6387DF 426C414
DF75C2F 657167A 68BD636 DD1C085 8359767

#1	**Cellebrite Technical Support** (Memory: Phone)
Work:	+495251546490

(a) Cellebrite extraction report



(b) Built-in contact list application

Importance	Name	Index	Work	Attachments
	Cellebrite Technical Support	1	+495251546490	0

(c) XRY extraction report

Figure 5.16: Contact lists with delayed restoration, first extracted with Cellebrite

Importance	Name	Index	Work	Attachments
	XRY Technical Support	1	+4687390270	0

(a) XRY extraction report

(b) Built-in contact list application

Phone Contacts

Total Entries: 1
PBB MD5 Hash: 42297811166D7D9E05C6942212D02374
PBB SHA256 Hash: DB7965F9 A98AD44 3EA55BB D380171
D9D5DC1 2FD7CC9 5F994AB 13691D3 9C3BE95

#1	**XRY Technical Support** (Memory: Phone)
Work:	+4687390270

(c) Cellebrite extraction report

Figure 5.17: Contact lists with delayed restoration, first extracted with XRY

Phone Contacts

Total Entries: 2
PBB MD5 Hash: 218EB04F9860342141979F19E75BFA1C
PBB SHA256 Hash: 73566850 44D4D16 7728AFA D428615 D21BA0B F6A479B A15BBA8 E07C34B DD152D0

#1	**Top-up** (Memory: SIM)
General:	5588

#2	**Voicemail** (Memory: SIM)
General:	121

(a) SIM contacts visible

Phone Contacts

Contacts Information Not Available

(b) SIM contacts hidden

Figure 5.18: Cellebrite extraction reports for SIM contact lists

Phone SMS - Text Messages

SMS MD5 Hash: 81161AD5A50B496D9DF55AC52190EBE7

SMS SHA256 Hash: C5357DEE 227EF1D E964105 3F7AF63 1F0B72C FFEFB81 6AF1EC4 DA7F21E ADC5D36

#	Number	Name	Date & Time	SMSC	Status	Folder	Storage	Type	Text
1	+447412995116	N/A	16/07/12 14:11:01 (GMT+1)	+447782000800	Read	Inbox	Phone	Incoming	This is test SMS number 1.
2	+447412995116	N/A	16/07/12 19:49:41 (GMT+1)	+447782000800	Read	Inbox	Phone	Incoming	This is test SMS number 2.
3	+447412995116	N/A	17/07/12 11:43:34 (GMT+1)	+447782000800	Read	Inbox	Phone	Incoming	This SMS will be delivered while the phone is connected to Cellebrite.
4	+447412995116	N/A	17/07/12 11:45:53 (GMT+1)	+447782000800	Read	Inbox	Phone	Incoming	This SMS will be delivered after Cellebrite has been disconnected.

* Phonebook name lookup used to retrieve names

(a) Standard CyanogenMod

Phone SMS - Text Messages

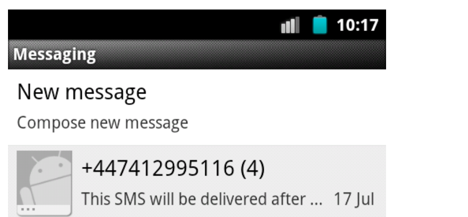
#	Number	Name	Date & Time	SMSC	Status	Folder	Storage	Type	Text

* Phonebook name lookup used to retrieve names

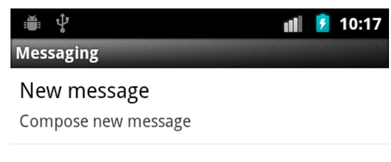
SMS Information Not Available

(b) SMS messages hidden

Figure 5.19: Cellebrite extraction logs showing SMS messages



(a) Normal



(b) SMS messages hidden

Figure 5.20: Screenshots of the built-in messaging application

Chapter 6

Conclusions

This dissertation set out with an hypothesis and a number of research questions. The hypothesis was that *it is possible to modify the Android operating system to present false information to the forensics tools Cellebrite and XRY*, and the research questions were:

- Which components of the Android operating system do the forensics tools trust?
- Is it possible to modify these components to present false information?
- Can the presence of a forensic analysis tool be detected?
- Is it possible to make the presentation of false information reversible, such that the phone will revert to presenting the real information after the forensic analysis?

6.1 Trust

Forensic analysts try to minimise the trust they place in the equipment under examination. This is expressed by disassembling PCs and examining them component by component, and by using forensics tools instead of operating systems to interpret the data on as low a level a possible. This is hard to do on a phone, where disassembling requires much effort, time and resources,

and interpretation of raw data may be impossible due to formats being undocumented. Therefore, phone forensics is mostly performed by letting the phone run its software and having the forensics tools ask that software for information.

This means that the forensics tools trust the phone software to return the correct results. In particular, both Cellebrite and XRY use the standard content provider interfaces to retrieve personal data from Android phones. Both tools can also perform logical acquisitions of the phone's entire memory, thereby bypassing the high-level phone software and only trusting the phone's file system driver to return the correct files. However, using that mode of acquisition requires that the tool or the analyst perform data interpretation themselves, without the help of the phone software. Both tools also use standard methods of installing software, thereby trusting the package manager to install that software correctly. These high-level software packages, in turn, trust the lower levels to function correctly. Therefore, the forensics tools also trust, by extension, all lower levels of the Android stack, including the Dalvik virtual machine, the Linux kernel and the hardware.

Cellebrite and XRY place slightly different levels of trust in different components of the Android phone. When the phone refused to install the forensics application, Cellebrite refused to run and presented an error message to the analyst, while XRY partially completed the extraction, reported a partial error, and produced a report containing very little data. Cellebrite was also more time-sensitive, with a ten second delay for each query being enough to cause errors. XRY, in contrast, did not report any errors even after six hours, and completed the extraction successfully if left alone for that time.

6.2 Modification

Any component of a system under forensic analysis that is trusted by one party is a point of attack for their opponent. Since the content providers and package manager are trusted by Cellebrite and XRY, they are natural targets for anti-forensics.

Android is an open system, with specifications and source code freely available. Several projects use that source code to build community distributions of Android which can be installed on many different models of phones. The installation requires that the phone be rooted, which is possible to do on many phones and popular among technologically sophisticated customers. Step-by-step guides available on the Internet describe how to root phones and install community distributions of Android.

These community distributions depend upon contributions of code from the general public. They therefore make it easy to modify their code and install the modified versions. While programming skills are a prerequisite, this dissertation has shown that it is possible to modify and replace content providers and the package manager.

6.3 Detection

Behaviour that is repeated is grounds for identification. If that behaviour is not the same as that produced by regular use, it is grounds for anomaly detection. Cellebrite and XRY provide both. Every time the behaviour of either was observed, each used the same name its uploaded application, and that application queried the content providers in the same way. They both also require the phone to be in USB debugging mode, which is unlikely to be the case for a phone in regular use.

The experiments performed here have shown that it is possible to distinguish between normal use and forensic analysis by looking at whether USB debugging is enabled, and that it is possible to distinguish between different forensics tools by looking at the names of their applications.

6.4 Reverting

Most phone anti-forensics is concerned with overwriting or deleting information. While this hopefully makes the data completely unavailable to the forensic analyst, it also makes it unavailable to the legitimate user if the

phone is eventually returned. Presenting false data to the analyst instead opens the possibility of hiding the real data and reverting to it after the analysis has been completed.

The anti-forensics implemented for this dissertation show that this is possible. The real information is left in its place, while false information is fed to forensics tools from other sources. The real information is made available again either immediately or after a set time.

6.5 Hypothesis

Forensics tools place a great deal of trust in the Android software, but that software can easily be modified and replaced. When suitably modified and replaced, that software can feed false information to the tool. Neither Cellebrite nor XRY detects this subterfuge, and so present the false information to the analyst as if it was real. The anti-forensics software modules are present on the phone and can be seen by the analyst should they do a logical extraction of the phone's file system. However, their presence and function is not obvious, and even if they are detected reverse-engineering them would require significant time and effort from the analyst.

This dissertation has presented a proof of the hypothesis by construction. It is possible to modify the Android operating system to present false information to Cellebrite and XRY, because such modifications have been performed and tested. The modifications:

- Hide the phone contact list from the forensics tools
- Present different false contacts to the forensics tools depending on which tool is used
- Continue to present false contacts for a set time after the forensics tool has been disconnected
- Delay the extraction of data by the forensics tools by an arbitrary amount of time

- Hide SIM contacts from the forensics tools
- Hide SMS messages from the forensics tools
- Prevent the forensics tools from being installed

Chapter 7

Future work

7.1 Decompilation of forensic applications

Standard Android applications are written in Java and compiled to Java bytecode. They are then translated to the Android-specific Dalvik [20] virtual machine, which runs the application on the phone. The application can be turned back into Java bytecode using tools such as dex2jar [21], after which generic Java reverse engineering techniques are available, such as the JD decompiler [52] which can turn Java bytecode into human-readable Java source code.

Both Cellebrite and XRY upload an application to the phone through the USB debug connection. This application reads the requested data and sends it to the tool. After extraction, the tool stops and deletes the application. Study of these applications could reveal avenues for more accurate detection of forensic tools and vulnerabilities in the applications. Experiments could be conducted to see if it is possible to make the application accept commands from an anti-forensic application running on the phone as well as the external forensic tool, any logs would implicate the forensic tool as responsible for the resulting behaviour. It's plausible, based on previous research pointing to forensic software being insecure and unprepared to deal with a hostile environment [75], that such vulnerabilities would be found.

Even if no actual vulnerabilities are found, there is information suggesting

that it is possible to use Java reflection to invoke arbitrary functions in other applications [76]. For example, Cellebrite sells products for phone backup, restoration and data transfer in addition to phone forensics. These would have the ability to write data to the phone as well as read it. An investigation could be conducted to see if the application has been reused between these different products, such that code for writing data is also present in the application uploaded by the forensics tool. If this is so, and an anti-forensics module could call that code, it would be possible to make it look like the forensics tool intentionally destroyed evidence. Even if no such extraneous functionality is present, simply calling legitimate forensic functions is an unexpected order might be enough to confuse the analysis tool and ruin any data extraction.

In order to decompile these applications, they must first be captured. Initial experiments with Cellebrite suggests that this is very easy. The deletion is triggered by the external Cellebrite tool, not internally from the uploaded application itself. To prevent the application from being deleted at the end of an extraction, simply unplug the USB cable between the Cellebrite tool and the phone before extraction is complete. The extraction will be aborted, but the application will still be running. The application could then be downloaded to a PC by connecting to the phone using a standard USB debug connection.

The legality of this procedure would depend on the specific licence for the forensic tool and its uploaded application, and the legal provisions for academic reverse engineering in the jurisdiction where the research is taking place.

7.2 Detection of forensics tools

The triggers used to detect the presence of a forensics tool currently use the name of the application uploaded to the phone, which would be easy for the tool vendor to change in the future.

More sophisticated triggers are possible. Two are immediately apparent from the logs of Cellebrite and XRY behaviour in appendices B.1 and B.2:

`raw_contacts` and a query-following state machine.

The `raw_contacts` method would look at queries for the URI[1] `content://com.android.contacts/raw_contacts`. Such queries return contacts data in a format suited for automated processing, while a different URI (`content://com.android.contacts/contacts`) returns data suitable for humans and is used by the built-in contact list application[2]. This suggests that triggering on the use of `raw_contacts` would be a good fallback method, but care must be taken not to trigger on legitimate applications. According to the documentation, it is primarily intended for use by applications that perform contact list synchronisation and other contact list management tasks.

A query-following state machine would take advantage of the fact that the forensics tools make the same set of queries in the same order each time. XRY makes only two queries, each for an entire dataset, and presumably does all processing in the analysis tool. This does not leave much structure to trigger on. Cellebrite, on the other hand, makes seven preparatory queries and then nine queries per contact (one each for the contact's name, phone number, e-mail address, etc.). A state machine could look at the seven preparatory queries, or the nine queries for one contact, and from that series of queries determine that the queries come from Cellebrite and serve false data in response to the following queries.

Another idea for a triggering mechanism can be had from the ACPO digital forensics guidelines [53]. This recommends that mobile phones be disconnected from the network before analysis, preferably using a Faraday cage. Presenting false data whenever the network connection is lost may therefore be a valid anti-forensic strategy. The connection may be legitimately lost during everyday use, for example by the user walking into a cellar. In these cases, the phone functionality is unavailable to the user anyway, so the unavailability of phone-related data may not be a significant drawback. For example, no phone calls can be made, so hiding the contact list may not inconvenience the user.

---

[1] Uniform Resource Identifier, a standard for specifying the name and location of data items [49].

[2] The format of the results returned are documented as ContactsContract.RawContacts [6] and ContactsContract.Contacts [5], respectively.

7.3 Encryption

Starting with version 3.0 (also known as "Honeycomb"), Android implements full-disk encryption using the Linux standard "dm-crypt" system [28]. Since Android series 3 was only available for tablets [14], encryption functionality was not available for smartphones until the release of Android 4.0 ("Ice Cream Sandwich") in October of 2011 [70]. No academic publications of the forensics consequences of this encryption have been found.

According to the documentation for the encryption feature [12], encryption requires a boot-time password and a screen lock password, which have to be identical. This password would probably be simple, since it needs to be entered frequently on a device not intended for quick, accurate input of long strings of characters. Since it is entered frequently, it might also be vulnerable to touch-screen attacks such as fingerprint smudge recognition [45].

Using encryption may increase the viability of anti-forensic applications. If an analyst knows they will need a password to boot the phone, they may be forced to leave it running instead of turning it off in order to remove components for separate analysis (such as the SIM card). This would increase the relevance of anti-forensics for SIM contacts and SMS messages.

7.4 Unrooting

Current systems for rooting Android phones are permanent, in that they allow unrestricted root access after they have been installed. However, there is no technical reason why this has to be so. Since forensic tools can take advantage of root access to make logical or physical copies of the phone's memory, removing this access would force these tools to use the content provider interfaces, which can have anti-forensic modifications.

After modifying the operating system not to grant root access automatically, this root access can be made more or less difficult to recover, depending on the user's needs. More difficulty makes the device more safe from forensic examination, but harder for the user to modify further.

The standard UNIX utility `su` allows a user to change their user ID, and thus gain root access. It is available in the user's search path[3] by default in CyanogenMod. If moved outside the standard search path, or renamed, it would still be available to a user who knows where to find it (or a sufficiently thorough forensic analyst), but unavailable to automatic tools which expect it to be in the default location.

Removing the `su` application completely still leaves the alternate recovery image. Using this, a complete new operating system can be installed on the device. Whether any user data survives this operation depends on the exact installation process. For example, using ClockworkMod to upgrade from CyanogenMod 7.1 to CyanogenMod 7.2 preserves all user data. For anti-forensics purposes, it would be recommended to modify the recovery image to require wiping all data before installing a new operating system.

For maximum protection against forensic examination, the recovery image should be reverted to the standard Android one. There would then be no intentional ways to gain root access, which would place a forensic investigator in the same position as a hacker with a newly released phone, trying to get an alternative operating system installed. However, to cater to the enthusiast market, many phone manufacturers began to include supported ways of unlocking the bootloader in 2011 [72]. This means that the investigator can easily install their own recovery image and use that to install an operating system granting root access with minimum alterations to the data already present on the device. The question would remain of how acceptable such a procedure would be to a court, since it involves substantial modification of the device.

7.5 SEAndroid

Standard Android uses just the traditional UNIX discretionary access control mechanism, which allows the superuser (root) to override any restrictions. When the phone is rooted, the user is allowed to install modified versions of

[3] A list of directories where the system automatically looks for programs when the user gives the name of a program to execute.

operating system components, but this also allows a forensics tool to access database files directly in the phone's file system instead of going through a content provider.

SELinux [86] is an initiative from the US National Security Agency to implement mandatory access control on Linux. It has been included in standard Linux since 2003 [93], and has been used by major desktop distributions such as Fedora [22] and Ubuntu [36] for several years.

SEAndroid [31] is a project for using SELinux on Android. While using it would require that the phone first be rooted to replace the operating system with one implementing SEAndroid, once running it would be able to restrict even the root user from certain actions. It would then be possible to forbid forensic tools from reading database files from the file system, thereby forcing them to go through the content provider to get the data.

7.6 Data destruction

The modifications performed for this dissertation were all non-destructive, meaning that the original data was still left on the phone, even when it was not presented to the forensics tools. However, this is not inherent in the design. The improved hiding and triggering properties found by implementing anti-forensics in the operating system over using a standalone application would also be able to hide destructive anti-forensics routines. For example, the package manager could be extended to not only reject the installation of forensics tools, but use the installation attempt as a trigger to perform a complete wipe of the phone. This would free the anti-forensics routines from the timing constraints which apply when they run as a separate application.

Appendix A

Software and hardware environments

The development of the tool, and all experiments, were carried out using the following software and hardware:

Phone HTC Desire GSM (also known as Bravo and A8181), Hboot version 0.93.0001

Rootkit Revolutionary version 0.4pre4 [30]

Recovery image Revolutionary ClockworkMod 4.0.1.4

Android 2.3.7 (Gingerbread)

Distribution CyanogenMod 7.2

CyanogenMod development environment Set up as documented on the CyanogenMod wiki [23], for branch `gb-release-7.2`.

Rooting and CyanogenMod installation Procedure as documented on the CyanogenMod wiki [24]

Cellebrite Versions: App: 1.1.9.4 UFED, Full: 1.0.2.7, Tiny: 1.0.2.1

XRY Version 6.1.1

Appendix B

Tool behaviour

The following are logs of the calls made by the forensics tools Cellebrite and XRY when told to extract the contact list from the phone. The contact list contained two entries, each with a name and a phone number. The fields correspond to the arguments to the main query function, `com.android.providers.contacts.ContactsProvider2::query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)`.

Documentation on the format of the data returned by these queries can be found in the Android developer documentation for the ContactsContract class [4].

B.1 Cellebrite

The application making these queries is called "com.client.appA".

```
URI: content://com.android.contacts/raw_contacts
Projection: null
Selection: null
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts
Projection: null
Selection: null
Selection arguments: null
Sort order: null
```

```
URI: content://com.android.contacts/settings
Projection: [account_type]
Selection: null
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts
Projection: null
Selection: deleted = ? AND ( account_type IS NULL )
Selection arguments: [0]
Sort order: null

URI: content://com.android.contacts/raw_contacts
Projection: null
Selection: null
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/settings
Projection: [account_type]
Selection: null
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts
Projection: [_id]
Selection: deleted < ?
Selection arguments: [1]
Sort order: null

URI: content://com.android.contacts/raw_contacts/1/entity
Projection: [data1, data3, data2, data5, data4, data6,
   is_primary, account_type, account_name]
Selection: _id = 1 AND
   mimetype = 'vnd.android.cursor.item/name'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/1/entity
Projection: [data1, data2, is_primary]
Selection: _id = 1 AND
   mimetype = 'vnd.android.cursor.item/phone_v2'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/1/entity
Projection: [data1, data2, is_primary]
Selection: _id = 1 AND
   mimetype = 'vnd.android.cursor.item/dispatch_v2'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/1/entity
Projection: [data1, data2, is_primary]
Selection: _id = 1 AND
   mimetype = 'vnd.android.cursor.item/email_v2'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/1/entity
Projection: [data5, data6, data4, data7, data8, data9,
   data10, data2, is_primary]
```

```
Selection: _id = 1 AND
   mimetype = 'vnd.android.cursor.item/postal-address_v2'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/1/entity
Projection: [data1, data2, data5, data6, is_primary]
Selection: _id = 1 AND
   mimetype = 'vnd.android.cursor.item/im'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/1/entity
Projection: [data2, data1, data4, is_primary]
Selection: _id = 1 AND
   mimetype = 'vnd.android.cursor.item/organization'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/1/entity
Projection: [data1]
Selection: _id = 1 AND
   mimetype = 'vnd.android.cursor.item/note'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/1/entity
Projection: [data1, data2, is_primary]
Selection: _id = 1 AND
   mimetype = 'vnd.android.cursor.item/website'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/2/entity
Projection: [data1, data3, data2, data5, data4, data6,
   is_primary, account_type, account_name]
Selection: _id = 2 AND
   mimetype = 'vnd.android.cursor.item/name'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/2/entity
Projection: [data1, data2, is_primary]
Selection: _id = 2 AND
   mimetype = 'vnd.android.cursor.item/phone_v2'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/2/entity
Projection: [data1, data2, is_primary]
Selection: _id = 2 AND
   mimetype = 'vnd.android.cursor.item/dispatch_v2'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/2/entity
Projection: [data1, data2, is_primary]
Selection: _id = 2 AND
   mimetype = 'vnd.android.cursor.item/email_v2'
Selection arguments: null
Sort order: null
```

```
URI: content://com.android.contacts/raw_contacts/2/entity
Projection: [data5, data6, data4, data7, data8, data9,
   data10, data2, is_primary]
Selection: _id = 2 AND
   mimetype = 'vnd.android.cursor.item/postal-address_v2'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/2/entity
Projection: [data1, data2, data5, data6, is_primary]
Selection: _id = 2 AND
   mimetype = 'vnd.android.cursor.item/im'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/2/entity
Projection: [data2, data1, data4, is_primary]
Selection: _id = 2 AND
   mimetype = 'vnd.android.cursor.item/organization'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/2/entity
Projection: [data1]
Selection: _id = 2 AND
   mimetype = 'vnd.android.cursor.item/note'
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/raw_contacts/2/entity
Projection: [data1, data2, is_primary]
Selection: _id = 2 AND
   mimetype = 'vnd.android.cursor.item/website'
Selection arguments: null
Sort order: null
```

B.2 XRY behaviour

The application making these queries is called “example.helloandroid”.

```
URI: content://com.android.contacts/raw_contacts
Projection: null
Selection: null
Selection arguments: null
Sort order: null

URI: content://com.android.contacts/data
Projection: null
Selection: null
Selection arguments: null
Sort order: null
```

Appendix C

Source code

C.1 USBMonitor

The USBMonitor application was built as a proof of concept for detecting when USB debugging is in use. It listens to the intent broadcast [2] used by the Android system to inform applications that the USB state has changed (called `UsbManager.ACTION_USB_STATE`), and the low-level Uevents generated by the Linux kernel when the hardware configuration changes. This information is decoded and printed to the screen.

C.1.1 AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
        "http://schemas.android.com/apk/res/android"
    package="uk.ac.gla.arts.hatii.usbmonitor"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="10" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".USBMonitorActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name=
                        "android.intent.action.MAIN" />
                <category android:name=
```

```
                                "android.intent.category.LAUNCHER"
                                />
                </intent-filter>
            </activity>

    </application>

</manifest>
```

C.1.2 res/layout/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:id="@+id/logText" />

</LinearLayout>
```

C.1.3 res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">USBMonitor</string>

</resources>
```

C.1.4 USBMonitorActivity.java

```
package uk.ac.gla.arts.hatii.usbmonitor;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.IntentFilter;
import android.hardware.usb.UsbManager;
import android.os.Bundle;
import android.os.UEventObserver;
import android.text.method.ScrollingMovementMethod;
import android.widget.TextView;

public class USBMonitorActivity extends Activity {

    private TextView logText;

    /**
     * The end-of-line string for this platform.
```

```java
 */
private String eol;

private BroadcastReceiver receiver = null;
private IntentFilter filter = null;

private UEventObserver observer = null;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    logText = (TextView)
            findViewById(R.id.logText);
    logText.setMovementMethod(
            new ScrollingMovementMethod());

    eol = System.getProperty("line.separator");

    this.addLogMessage("Waiting for events...");

    receiver = new USBBroadcastReceiver(this);
    filter = new IntentFilter();

    // This is the CyanogenMod 7.1 UsbManager,
    // not the one from stock Android 2.3 or
    // the backported Google API:s.
    filter.addAction(UsbManager.ACTION_USB_STATE);

    observer = new USBUeventObserver(this);
}

@Override
protected void onPause() {
    this.addLogMessage("Pausing");
    unregisterReceiver(receiver);
    observer.stopObserving();
    super.onPause();
}

@Override
protected void onResume() {
    this.addLogMessage("Resuming");
    registerReceiver(receiver, filter);
    observer.startObserving("");
    super.onResume();
}

/**
 * Adds a message to the log text field,
 * automatically adding a terminating newline.
 *
 * @param message
 *            The log message.
 *
 */
public void addLogMessage(String message) {
    logText.append(message + eol);
}

/**
```

```
     * Like addLogMessage(String), but callable from
     * other threads.
     *
     * @param string
     *            The log message.
     *
     */
    public void addLogMessageFromOtherThread(
            final String string) {
        // http://stackoverflow.com/questions/3050937/
        //     java-thread-message-passing
        runOnUiThread(new Runnable() {
            public void run() {
                addLogMessage(string);
            }
        });
    }
}
```

C.1.5 USBBroadcastReceiver.java

```
package uk.ac.gla.arts.hatii.usbmonitor;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.hardware.usb.UsbManager;
import android.os.Bundle;

public class USBBroadcastReceiver extends BroadcastReceiver {

    private static final String TAG = "USBBroadcastReceiver";

    /**
     * The activity that started us.
     */
    private USBMonitorActivity activity = null;

    /**
     * @param parent
     *            The activity that started us and will get
     *            the log messages.
     */
    public USBBroadcastReceiver(USBMonitorActivity parent) {
        activity = parent;
    }

    /**
     * @see android.content.BroadcastReceiver#
     * onReceive(android.content.Context,
     * android.content.Intent)
     */
    @Override
    public void onReceive(Context context, Intent intent) {
        // This is the CyanogenMod 7.1 UsbManager, not the one
        // from stock Android 2.3 or the backported Google
        // API:s.
        if (intent.getAction().equals(
                UsbManager.ACTION_USB_STATE)) {
```

```
            Bundle extras = intent.getExtras();
            boolean usbConnected =
                extras.getBoolean(UsbManager.USB_CONNECTED);
            boolean adbEnabled =
                extras.getString(UsbManager.USB_FUNCTION_ADB)
                        .equals(UsbManager.USB_FUNCTION_ENABLED);
            activity.addLogMessage(
                "onReceive(" + intent.getAction() + "): " +
                (usbConnected ? "" : "dis") + "connected, " +
                (adbEnabled ? "ADB on" : "ADB off"));
            activity.addLogMessage(
                "Extras: " + extras.keySet().toString());
        } else {
            activity.addLogMessage(
                "onReceive(" + intent.getAction() + ")");
        }
    }
}
```

C.1.6 USBUeventObserver.java

```
package uk.ac.gla.arts.hatii.usbmonitor;

import android.os.UEventObserver;

public class USBUeventObserver extends UEventObserver {

    /**
     * The activity that started us.
     */
    private USBMonitorActivity activity = null;

    public USBUeventObserver(USBMonitorActivity parent) {
        activity = parent;
    }

    @Override
    public void onUEvent(UEvent event) {
        // UEventObservers run in their own implicit
        // thread, and cannot touch the UI by themselves.
        // Post a message to the activity to have the UI
        // thread add the log message for us.
        activity.addLogMessageFromOtherThread(
            "UEvent: " + event.toString());
    }

}
```

C.2 Instrumentation of contacts provider

This is the code which instruments the contacts provider, providing a log of the calls made from forensics tools. It is provided as a patch on top of CyanogenMod. The original CyanogenMod source code repository is available at `git://github.com/CyanogenMod/`

android_packages_providers_ContactsProvider, and this patch is on top of the branch gb-release-7.2. The patched file is src/com/android/providers/contacts/ContactsProvider2.java, which is installed on the phone as part of the contacts provider package, /system/app/ContactsProvider.apk.

For each call to query(), the contact list query function, the code outputs the name of the calling process, the query arguments and which branch of the original program logic handles the code. The results are written to the system log.

```
index 3bee54d..00be75e 100644
--- gb-release-7.2/src/com/android/providers/contacts/ContactsProvider2.java
+++ instrumentation/src/com/android/providers/contacts/ContactsProvider2.java
@@ -16,32 +16,28 @@

 package com.android.providers.contacts;

-import com.android.internal.content.SyncStateContentProviderHelper;
-import com.android.providers.contacts.ContactLookupKey.LookupKeySegment;
-import com.android.providers.contacts.ContactsDatabaseHelper.AggregatedPresenceColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.AggregationExceptionColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.Clauses;
-import com.android.providers.contacts.ContactsDatabaseHelper.ContactsColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.ContactsStatusUpdatesColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.DataColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.GroupsColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.MimetypesColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.NameLookupColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.NameLookupType;
-import com.android.providers.contacts.ContactsDatabaseHelper.PhoneColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.PhoneLookupColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.PresenceColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.RawContactsColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.SettingsColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.StatusUpdatesColumns;
-import com.android.providers.contacts.ContactsDatabaseHelper.Tables;
-import com.google.android.collect.Lists;
-import com.google.android.collect.Maps;
-import com.google.android.collect.Sets;
+import java.io.ByteArrayOutputStream;
+import java.io.FileNotFoundException;
+import java.io.IOException;
+import java.io.OutputStream;
+import java.text.SimpleDateFormat;
+import java.util.ArrayList;
+import java.util.Arrays;
+import java.util.Collections;
+import java.util.Date;
+import java.util.HashMap;
+import java.util.HashSet;
```

```
+import java.util.List;
+import java.util.Locale;
+import java.util.Map;
+import java.util.Set;
+import java.util.concurrent.CountDownLatch;

 import android.accounts.Account;
 import android.accounts.AccountManager;
 import android.accounts.OnAccountsUpdateListener;
+import android.app.Activity;
+import android.app.ActivityManager;
 import android.app.Notification;
 import android.app.NotificationManager;
 import android.app.PendingIntent;
@@ -73,6 +69,7 @@ import android.database.sqlite.SQLiteQueryBuilder;
 import android.database.sqlite.SQLiteStatement;
 import android.net.Uri;
 import android.os.AsyncTask;
+import android.os.Binder;
 import android.os.Bundle;
 import android.os.MemoryFile;
 import android.os.RemoteException;
@@ -83,10 +80,17 @@ import android.pim.vcard.VCardConfig;
 import android.preference.PreferenceManager;
 import android.provider.BaseColumns;
 import android.provider.ContactsContract;
-import android.provider.LiveFolders;
-import android.provider.OpenableColumns;
-import android.provider.SyncStateContract;
 import android.provider.ContactsContract.AggregationExceptions;
+import android.provider.ContactsContract.CommonDataKinds.BaseTypes;
+import android.provider.ContactsContract.CommonDataKinds.Email;
+import android.provider.ContactsContract.CommonDataKinds.GroupMembership;
+import android.provider.ContactsContract.CommonDataKinds.Im;
+import android.provider.ContactsContract.CommonDataKinds.Nickname;
+import android.provider.ContactsContract.CommonDataKinds.Organization;
+import android.provider.ContactsContract.CommonDataKinds.Phone;
+import android.provider.ContactsContract.CommonDataKinds.Photo;
+import android.provider.ContactsContract.CommonDataKinds.StructuredName;
+import android.provider.ContactsContract.CommonDataKinds.StructuredPostal;
```

```
 import android.provider.ContactsContract.ContactCounts;
 import android.provider.ContactsContract.Contacts;
 import android.provider.ContactsContract.Data;
@@ -101,35 +105,35 @@ import android.provider.ContactsContract.RawContacts;
 import android.provider.ContactsContract.SearchSnippetColumns;
 import android.provider.ContactsContract.Settings;
 import android.provider.ContactsContract.StatusUpdates;
-import android.provider.ContactsContract.CommonDataKinds.BaseTypes;
-import android.provider.ContactsContract.CommonDataKinds.Email;
-import android.provider.ContactsContract.CommonDataKinds.GroupMembership;
-import android.provider.ContactsContract.CommonDataKinds.Im;
-import android.provider.ContactsContract.CommonDataKinds.Nickname;
-import android.provider.ContactsContract.CommonDataKinds.Organization;
-import android.provider.ContactsContract.CommonDataKinds.Phone;
-import android.provider.ContactsContract.CommonDataKinds.Photo;
-import android.provider.ContactsContract.CommonDataKinds.StructuredName;
-import android.provider.ContactsContract.CommonDataKinds.StructuredPostal;
+import android.provider.LiveFolders;
+import android.provider.OpenableColumns;
+import android.provider.SyncStateContract;
 import android.telephony.PhoneNumberUtils;
 import android.text.TextUtils;
 import android.util.Log;

-import java.io.ByteArrayOutputStream;
-import java.io.FileNotFoundException;
-import java.io.IOException;
-import java.io.OutputStream;
-import java.text.SimpleDateFormat;
-import java.util.ArrayList;
-import java.util.Collections;
-import java.util.Date;
-import java.util.HashMap;
-import java.util.HashSet;
-import java.util.List;
-import java.util.Locale;
-import java.util.Map;
-import java.util.Set;
-import java.util.concurrent.CountDownLatch;
+import com.android.internal.content.SyncStateContentProviderHelper;
```

```
+import com.android.providers.contacts.ContactLookupKey.LookupKeySegment;
+import com.android.providers.contacts.ContactsDatabaseHelper.AggregatedPresenceColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.AggregationExceptionColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.Clauses;
+import com.android.providers.contacts.ContactsDatabaseHelper.ContactsColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.ContactsStatusUpdatesColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.DataColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.GroupsColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.MimetypesColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.NameLookupColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.NameLookupType;
+import com.android.providers.contacts.ContactsDatabaseHelper.PhoneColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.PhoneLookupColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.PresenceColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.RawContactsColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.SettingsColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.StatusUpdatesColumns;
+import com.android.providers.contacts.ContactsDatabaseHelper.Tables;
+import com.google.android.collect.Lists;
+import com.google.android.collect.Maps;
+import com.google.android.collect.Sets;

 /**
  * Contacts content provider. The contract between this provider and applications
@@ -4192,6 +4196,22 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
          return false;
     }

+    private String getProcessNameFromPid(int givenPid)
+    {
+       ActivityManager am = (ActivityManager)
+          getContext().getSystemService(Activity.ACTIVITY_SERVICE);
+
+       List<ActivityManager.RunningAppProcessInfo> lstAppInfo =
+          am.getRunningAppProcesses();
+
+       for(ActivityManager.RunningAppProcessInfo ai : lstAppInfo) {
+          if (ai.pid == givenPid) {
+             return ai.processName;
+          }
```

```
+           }
+           return null;
+       }
+
        @Override
        public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
                String sortOrder) {
@@ -4199,6 +4219,13 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                Log.v(TAG, "query: " + uri);
            }

+           Log.i(TAG, "Query from: " + getProcessNameFromPid(Binder.getCallingPid()));
+           Log.i(TAG, "    URI: " + uri.toString());
+           Log.i(TAG, "    Projection: " + Arrays.toString(projection));
+           Log.i(TAG, "    Selection: " + selection);
+           Log.i(TAG, "    Selection arguments: " + Arrays.toString(selectionArgs));
+           Log.i(TAG, "    Sort order: " + sortOrder);
+
            final SQLiteDatabase db = mDbHelper.getReadableDatabase();

            SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
@@ -4210,15 +4237,18 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
            final int match = sUriMatcher.match(uri);
            switch (match) {
                case SYNCSTATE:
+                   Log.i(TAG, "    Branch SYNCSTATE");
                    return mDbHelper.getSyncState().query(db, projection, selection,  selectionArgs,
                            sortOrder);

                case CONTACTS: {
+                   Log.i(TAG, "    Branch CONTACTS");
                    setTablesAndProjectionMapForContacts(qb, uri, projection);
                    break;
                }

                case CONTACTS_ID: {
+                   Log.i(TAG, "    Branch CONTACTS_ID");
                    long contactId = ContentUris.parseId(uri);
                    setTablesAndProjectionMapForContacts(qb, uri, projection);
                    selectionArgs = insertSelectionArg(selectionArgs, String.valueOf(contactId));
```

```
@@ -4228,6 +4258,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun

                 case CONTACTS_LOOKUP:
                 case CONTACTS_LOOKUP_ID: {
+                    Log.i(TAG, "    Branch CONTACTS_LOOKUP(_ID)");
                     List<String> pathSegments = uri.getPathSegments();
                     int segmentCount = pathSegments.size();
                     if (segmentCount < 3) {
@@ -4267,6 +4298,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                 }

                 case CONTACTS_AS_VCARD: {
+                    Log.i(TAG, "    Branch CONTACTS_AS_VCARD");
                     // When reading as vCard always use restricted view
                     final String lookupKey = Uri.encode(uri.getPathSegments().get(2));
                     qb.setTables(mDbHelper.getContactView(true /* require restricted */));
@@ -4278,6 +4310,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                 }

                 case CONTACTS_AS_MULTI_VCARD: {
+                    Log.i(TAG, "    Branch CONTACTS_AS_MULTI_VCARD");
                     SimpleDateFormat dateFormat = new SimpleDateFormat("yyyyMMdd_HHmmss");
                     String currentDateString = dateFormat.format(new Date()).toString();
                     return db.rawQuery(
@@ -4288,6 +4321,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                 }

                 case CONTACTS_FILTER: {
+                    Log.i(TAG, "    Branch CONTACTS_FILTER");
                     String filterParam = "";
                     if (uri.getPathSegments().size() > 2) {
                         filterParam = uri.getLastPathSegment();
@@ -4298,6 +4332,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun

                 case CONTACTS_STREQUENT_FILTER:
                 case CONTACTS_STREQUENT: {
+                    Log.i(TAG, "    Branch CONTACTS_STREQUENT(_FILTER)");
                     String filterSql = null;
                     if (match == CONTACTS_STREQUENT_FILTER
                             && uri.getPathSegments().size() > 3) {
```

```
@@ −4349,6 +4384,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
             }

             case CONTACTS_GROUP: {
+                Log.i(TAG, "    Branch CONTACTS_GROUP");
                 setTablesAndProjectionMapForContacts(qb, uri, projection);
                 if (uri.getPathSegments().size() > 2) {
                     qb.appendWhere(CONTACTS_IN_GROUP_SELECT);
@@ −4358,6 +4394,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
             }

             case CONTACTS_DATA: {
+                Log.i(TAG, "    Branch CONTACTS_DATA");
                 long contactId = Long.parseLong(uri.getPathSegments().get(1));
                 setTablesAndProjectionMapForData(qb, uri, projection, false);
                 selectionArgs = insertSelectionArg(selectionArgs, String.valueOf(contactId));
@@ −4366,6 +4403,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
             }

             case CONTACTS_PHOTO: {
+                Log.i(TAG, "    Branch CONTACTS_PHOTO");
                 long contactId = Long.parseLong(uri.getPathSegments().get(1));
                 setTablesAndProjectionMapForData(qb, uri, projection, false);
                 selectionArgs = insertSelectionArg(selectionArgs, String.valueOf(contactId));
@@ −4375,12 +4413,14 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
             }

             case PHONES: {
+                Log.i(TAG, "    Branch PHONES");
                 setTablesAndProjectionMapForData(qb, uri, projection, false);
                 qb.appendWhere(" AND " + Data.MIMETYPE + " = '" + Phone.CONTENT_ITEM_TYPE + "'");
                 break;
             }

             case PHONES_ID: {
+                Log.i(TAG, "    Branch PHONES_ID");
                 setTablesAndProjectionMapForData(qb, uri, projection, false);
                 selectionArgs = insertSelectionArg(selectionArgs, uri.getLastPathSegment());
                 qb.appendWhere(" AND " + Data.MIMETYPE + " = '" + Phone.CONTENT_ITEM_TYPE + "'");
@@ −4389,6 +4429,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
```

```
                }

                case PHONES_FILTER: {
+                   Log.i(TAG, "    Branch PHONES_FILTER");
                    setTablesAndProjectionMapForData(qb, uri, projection, true);
                    qb.appendWhere(" AND " + Data.MIMETYPE + " = '" + Phone.CONTENT_ITEM_TYPE + "'");
                    if (uri.getPathSegments().size() > 2) {
@@ -4437,12 +4478,14 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                }

                case EMAILS: {
+                   Log.i(TAG, "    Branch EMAILS");
                    setTablesAndProjectionMapForData(qb, uri, projection, false);
                    qb.appendWhere(" AND " + Data.MIMETYPE + " = '" + Email.CONTENT_ITEM_TYPE + "'");
                    break;
                }

                case EMAILS_ID: {
+                   Log.i(TAG, "    Branch EMAILS_ID");
                    setTablesAndProjectionMapForData(qb, uri, projection, false);
                    selectionArgs = insertSelectionArg(selectionArgs, uri.getLastPathSegment());
                    qb.appendWhere(" AND " + Data.MIMETYPE + " = '" + Email.CONTENT_ITEM_TYPE + "'"
@@ -4451,6 +4494,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                }

                case EMAILS_LOOKUP: {
+                   Log.i(TAG, "    Branch EMAILS_LOOKUP");
                    setTablesAndProjectionMapForData(qb, uri, projection, false);
                    qb.appendWhere(" AND " + Data.MIMETYPE + " = '" + Email.CONTENT_ITEM_TYPE + "'");
                    if (uri.getPathSegments().size() > 2) {
@@ -4463,6 +4507,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                }

                case EMAILS_FILTER: {
+                   Log.i(TAG, "    Branch EMAILS_FILTER");
                    setTablesAndProjectionMapForData(qb, uri, projection, true);
                    String filterParam = null;
                    if (uri.getPathSegments().size() > 3) {
@@ -4515,6 +4560,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                }
```

```
            case POSTALS: {
+               Log.i(TAG, "    Branch POSTALS");
                setTablesAndProjectionMapForData(qb, uri, projection, false);
                qb.appendWhere(" AND " + Data.MIMETYPE + " = '"
                        + StructuredPostal.CONTENT_ITEM_TYPE + "'");
@@ -4522,6 +4568,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
            }

            case POSTALS_ID: {
+               Log.i(TAG, "    Branch POSTALS_ID");
                setTablesAndProjectionMapForData(qb, uri, projection, false);
                selectionArgs = insertSelectionArg(selectionArgs, uri.getLastPathSegment());
                qb.appendWhere(" AND " + Data.MIMETYPE + " = '"
@@ -4531,11 +4578,13 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
            }

            case RAW_CONTACTS: {
+               Log.i(TAG, "    Branch RAW_CONTACTS");
                setTablesAndProjectionMapForRawContacts(qb, uri);
                break;
            }

            case RAW_CONTACTS_ID: {
+               Log.i(TAG, "    Branch RAW_CONTACTS_ID");
                long rawContactId = ContentUris.parseId(uri);
                setTablesAndProjectionMapForRawContacts(qb, uri);
                selectionArgs = insertSelectionArg(selectionArgs, String.valueOf(rawContactId));
@@ -4544,6 +4593,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
            }

            case RAW_CONTACTS_DATA: {
+               Log.i(TAG, "    Branch RAW_CONTACTS_DATA");
                long rawContactId = Long.parseLong(uri.getPathSegments().get(1));
                setTablesAndProjectionMapForData(qb, uri, projection, false);
                selectionArgs = insertSelectionArg(selectionArgs, String.valueOf(rawContactId));
@@ -4552,11 +4602,13 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
            }

            case DATA: {
```

```
+                   Log.i(TAG, "    Branch DATA");
                    setTablesAndProjectionMapForData(qb, uri, projection, false);
                    break;
                }

                case DATA_ID: {
+                   Log.i(TAG, "    Branch DATA_ID");
                    setTablesAndProjectionMapForData(qb, uri, projection, false);
                    selectionArgs = insertSelectionArg(selectionArgs, uri.getLastPathSegment());
                    qb.appendWhere(" AND " + Data._ID + "=?");
@@ -4564,6 +4616,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                }

                case PHONE_LOOKUP: {
+                   Log.i(TAG, "    Branch PHONE_LOOKUP");

                    if (TextUtils.isEmpty(sortOrder)) {
                        // Default the sort order to something reasonable so we get consistent
@@ -4582,6 +4635,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                }

                case GROUPS: {
+                   Log.i(TAG, "    Branch GROUPS");
                    qb.setTables(mDbHelper.getGroupView());
                    qb.setProjectionMap(sGroupsProjectionMap);
                    appendAccountFromParameter(qb, uri);
@@ -4589,6 +4643,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                }

                case GROUPS_ID: {
+                   Log.i(TAG, "    Branch GROUPS_ID");
                    qb.setTables(mDbHelper.getGroupView());
                    qb.setProjectionMap(sGroupsProjectionMap);
                    selectionArgs = insertSelectionArg(selectionArgs, uri.getLastPathSegment());
@@ -4597,6 +4652,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                }

                case GROUPS_SUMMARY: {
+                   Log.i(TAG, "    Branch GROUPS_SUMMARY");
                    qb.setTables(mDbHelper.getGroupView() + " AS groups");
```

```
                qb.setProjectionMap(sGroupsSummaryProjectionMap);
                appendAccountFromParameter(qb, uri);
@@ -4605,12 +4661,14 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
            }

            case AGGREGATION_EXCEPTIONS: {
+                Log.i(TAG, "    Branch AGGREGATION_EXCEPTIONS");
                qb.setTables(Tables.AGGREGATION_EXCEPTIONS);
                qb.setProjectionMap(sAggregationExceptionsProjectionMap);
                break;
            }

            case AGGREGATION_SUGGESTIONS: {
+                Log.i(TAG, "    Branch AGGREGATION_SUGGESTIONS");
                long contactId = Long.parseLong(uri.getPathSegments().get(1));
                String filter = null;
                if (uri.getPathSegments().size() > 3) {
@@ -4630,6 +4688,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
            }

            case SETTINGS: {
+                Log.i(TAG, "    Branch SETTINGS");
                qb.setTables(Tables.SETTINGS);
                qb.setProjectionMap(sSettingsProjectionMap);
                appendAccountFromParameter(qb, uri);
@@ -4651,11 +4710,13 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
            }

            case STATUS_UPDATES: {
+                Log.i(TAG, "    Branch STATUS_UPDATES");
                setTableAndProjectionMapForStatusUpdates(qb, projection);
                break;
            }

            case STATUS_UPDATES_ID: {
+                Log.i(TAG, "    Branch STATUS_UPDATES_ID");
                setTableAndProjectionMapForStatusUpdates(qb, projection);
                selectionArgs = insertSelectionArg(selectionArgs, uri.getLastPathSegment());
                qb.appendWhere(DataColumns.CONCRETE_ID + "=?");
@@ -4663,32 +4724,38 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
```

```
            }

            case SEARCH_SUGGESTIONS: {
+               Log.i(TAG, "    Branch SEARCH_SUGGESTIONS");
                return mGlobalSearchSupport.handleSearchSuggestionsQuery(db, uri, limit);
            }

            case SEARCH_SHORTCUT: {
+               Log.i(TAG, "    Branch SEARCH_SHORTCUT");
                String lookupKey = uri.getLastPathSegment();
                return mGlobalSearchSupport.handleSearchShortcutRefresh(db, lookupKey, projection);
            }

            case LIVE_FOLDERS_CONTACTS:
+               Log.i(TAG, "    Branch LIVE_FOLDERS_CONTACTS");
                qb.setTables(mDbHelper.getContactView());
                qb.setProjectionMap(sLiveFoldersProjectionMap);
                break;

            case LIVE_FOLDERS_CONTACTS_WITH_PHONES:
+               Log.i(TAG, "    Branch LIVE_FOLDERS_CONTACTS_WITH_PHONES");
                qb.setTables(mDbHelper.getContactView());
                qb.setProjectionMap(sLiveFoldersProjectionMap);
                qb.appendWhere(Contacts.HAS_PHONE_NUMBER + "=1");
                break;

            case LIVE_FOLDERS_CONTACTS_FAVORITES:
+               Log.i(TAG, "    Branch LIVE_FOLDERS_CONTACTS_FAVORITES");
                qb.setTables(mDbHelper.getContactView());
                qb.setProjectionMap(sLiveFoldersProjectionMap);
                qb.appendWhere(Contacts.STARRED + "=1");
                break;

            case LIVE_FOLDERS_CONTACTS_GROUP_NAME:
+               Log.i(TAG, "    Branch LIVE_FOLDERS_CONTACTS_GROUP_NAME");
                qb.setTables(mDbHelper.getContactView());
                qb.setProjectionMap(sLiveFoldersProjectionMap);
                qb.appendWhere(CONTACTS_IN_GROUP_SELECT);
@@ -4696,11 +4763,13 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                break;
```

```
                case RAW_CONTACT_ENTITIES: {
+                   Log.i(TAG, "    Branch RAW_CONTACT_ENTITIES");
                    setTablesAndProjectionMapForRawContactsEntities(qb, uri);
                    break;
                }

                case RAW_CONTACT_ENTITY_ID: {
+                   Log.i(TAG, "    Branch RAW_CONTACT_ENTITY_ID");
                    long rawContactId = Long.parseLong(uri.getPathSegments().get(1));
                    setTablesAndProjectionMapForRawContactsEntities(qb, uri);
                    selectionArgs = insertSelectionArg(selectionArgs, String.valueOf(rawContactId));
@@ -4709,10 +4778,12 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
                }

                case PROVIDER_STATUS: {
+                   Log.i(TAG, "    Branch PROVIDER_STATUS");
                    return queryProviderStatus(uri, projection);
                }

                default:
+                   Log.i(TAG, "    Branch DEFAULT");
                    return mLegacyApiSupport.query(uri, projection, selection, selectionArgs,
                            sortOrder, limit);
            }
```

C.3 Rejecting installation of forensics tools

This is the code for rejecting forensics tools by checking their name at install time. It is provided as a patch on top of CyanogenMod. The original CyanogenMod source code repository is available at `git://github.com/CyanogenMod/android_frameworks_base`, and this patch is on top of the branch `gb-release-7.2`. The patched file is `cmds/pm/src/com/android/commands/pm/Pm.java`, which is installed on the phone as part of the package manager, `/system/framework/pm.jar`.

Before the package manager actually performs an installation, this code unpacks the APK file, checks the name of the application and returns an error if the name is recognised as belonging to a known forensics tool.

```
index f2b6fce..1eb2972 100644
--- gb-release-7.2/cmds/pm/src/com/android/commands/pm/Pm.java
+++ reject-by-name/cmds/pm/src/com/android/commands/pm/Pm.java
@@ -36,11 +36,12 @@ import android.content.pm.PermissionInfo;
 import android.content.res.AssetManager;
 import android.content.res.Resources;
 import android.net.Uri;
-import android.os.Parcel;
 import android.os.RemoteException;
 import android.os.ServiceManager;

 import java.io.File;
+import java.io.IOException;
+import java.io.InputStream;
 import java.lang.reflect.Field;
 import java.lang.reflect.Modifier;
 import java.util.ArrayList;
@@ -48,6 +49,7 @@ import java.util.Collections;
 import java.util.Comparator;
 import java.util.List;
 import java.util.WeakHashMap;
+import java.util.jar.JarFile;

 public final class Pm {
     IPackageManager mPm;
@@ -714,6 +716,172 @@ public final class Pm {
         }
     }

+    // XML decompression from
+    // http://stackoverflow.com/questions/2097813/how-to-parse-the-androidmanifest-xml-file-inside-an-apk-package
+
+    // decompressXML -- Parse the 'compressed' binary form of Android XML docs
+    // such as for AndroidManifest.xml in .apk files
+    public static int endDocTag = 0x00100101;
+    public static int startTag =  0x00100102;
+    public static int endTag =    0x00100103;
+    public void decompressXML(byte[] xml) {
+       // Compressed XML file/bytes starts with 24x bytes of data,
+       // 9 32 bit words in little endian order (LSB first):
```

```
+       //   0th word is 03 00 08 00
+       //   3rd word SEEMS TO BE:  Offset at then of StringTable
+       //   4th word is: Number of strings in string table
+       // WARNING: Sometime I indiscriminently display or refer to word in
+       //    little endian storage format, or in integer format (ie MSB first).
+       int numbStrings = LEW(xml, 4*4);
+
+       // StringIndexTable starts at offset 24x, an array of 32 bit LE offsets
+       // of the length/string data in the StringTable.
+       int sitOff = 0x24;  // Offset of start of StringIndexTable
+
+       // StringTable, each string is represented with a 16 bit little endian
+       // character count, followed by that number of 16 bit (LE) (Unicode) chars.
+       int stOff = sitOff + numbStrings*4;  // StringTable follows StrIndexTable
+
+       // XMLTags, The XML tag tree starts after some unknown content after the
+       // StringTable.  There is some unknown data after the StringTable, scan
+       // forward from this point to the flag for the start of an XML start tag.
+       int xmlTagOff = LEW(xml, 3*4);  // Start from the offset in the 3rd word.
+       // Scan forward until we find the bytes: 0x02011000(x00100102 in normal int)
+       for (int ii=xmlTagOff; ii<xml.length-4; ii+=4) {
+               if (LEW(xml, ii) == startTag) {
+                       xmlTagOff = ii;  break;
+               }
+       } // end of hack, scanning for start of first start tag
+
+       // XML tags and attributes:
+       // Every XML start and end tag consists of 6 32 bit words:
+       //   0th word: 02011000 for startTag and 03011000 for endTag
+       //   1st word: a flag?, like 38000000
+       //   2nd word: Line of where this tag appeared in the original source file
+       //   3rd word: FFFFFFFF ??
+       //   4th word: StringIndex of NameSpace name, or FFFFFFFF for default NS
+       //   5th word: StringIndex of Element Name
+       //   (Note: 01011000 in 0th word means end of XML document, endDocTag)
+
+       // Start tags (not end tags) contain 3 more words:
+       //   6th word: 14001400 meaning??
+       //   7th word: Number of Attributes that follow this tag(follow word 8th)
+       //   8th word: 00000000 meaning??
```

```
+
+        // Attributes consist of 5 words:
+        //   0th word: StringIndex of Attribute Name's Namespace, or FFFFFFFF
+        //   1st word: StringIndex of Attribute Name
+        //   2nd word: StringIndex of Attribute Value, or FFFFFFF if ResourceId used
+        //   3rd word: Flags?
+        //   4th word: str ind of attr value again, or ResourceId of value
+
+        // TMP, dump string table to tr for debugging
+        //tr.addSelect("strings", null);
+        //for (int ii=0; ii<numbStrings; ii++) {
+        //  // Length of string starts at StringTable plus offset in StrIndTable
+        //  String str = compXmlString(xml, sitOff, stOff, ii);
+        //  tr.add(String.valueOf(ii), str);
+        //}
+        //tr.parent();
+
+        // Step through the XML tree element tags and attributes
+        int off = xmlTagOff;
+        int indent = 0;
+        int startTagLineNo = -2;
+        while (off < xml.length) {
+                int tag0 = LEW(xml, off);
+                //int tag1 = LEW(xml, off+1*4);
+                int lineNo = LEW(xml, off+2*4);
+                //int tag3 = LEW(xml, off+3*4);
+                int nameNsSi = LEW(xml, off+4*4);
+                int nameSi = LEW(xml, off+5*4);
+
+                if (tag0 == startTag) { // XML START TAG
+                        int tag6 = LEW(xml, off+6*4);  // Expected to be 14001400
+                        int numbAttrs = LEW(xml, off+7*4);  // Number of Attributes to follow
+                        //int tag8 = LEW(xml, off+8*4);  // Expected to be 00000000
+                        off += 9*4;  // Skip over 6+3 words of startTag data
+                        String name = compXmlString(xml, sitOff, stOff, nameSi);
+                        //tr.addSelect(name, null);
+                        startTagLineNo = lineNo;
+
+                        // Look for the Attributes
+                        StringBuffer sb = new StringBuffer();
```

```
+                       for (int ii=0; ii<numbAttrs; ii++) {
+                               int attrNameNsSi = LEW(xml, off);  // AttrName Namespace Str Ind, or FFFFFFFF
+                               int attrNameSi = LEW(xml, off+1*4);  // AttrName String Index
+                               int attrValueSi = LEW(xml, off+2*4); // AttrValue Str Ind, or FFFFFFFF
+                               int attrFlags = LEW(xml, off+3*4);
+                               int attrResId = LEW(xml, off+4*4);  // AttrValue ResourceId or dup AttrValue StrInd
+                               off += 5*4;  // Skip over the 5 words of an attribute
+
+                               String attrName = compXmlString(xml, sitOff, stOff, attrNameSi);
+                               String attrValue = attrValueSi!=-1
+                                       ? compXmlString(xml, sitOff, stOff, attrValueSi)
+                                               : "resourceID 0x"+Integer.toHexString(attrResId);
+                                       sb.append(" "+attrName+"=\""+attrValue+"\"");
+                                       //tr.add(attrName, attrValue);
+                       }
+                       prtIndent(indent, "<"+name+sb+">");
+                       indent++;
+
+               } else if (tag0 == endTag) { // XML END TAG
+                       indent--;
+                       off += 6*4;  // Skip over 6 words of endTag data
+                       String name = compXmlString(xml, sitOff, stOff, nameSi);
+                       prtIndent(indent, "</"+name+">  (line "+startTagLineNo+"-"+lineNo+")");
+                       //tr.parent();  // Step back up the NobTree
+
+               } else if (tag0 == endDocTag) {  // END OF XML DOC TAG
+                       break;
+
+               } else {
+                       prt("  Unrecognized tag code '"+Integer.toHexString(tag0)
+                               +"' at offset "+off);
+                       break;
+               }
+       } // end of while loop scanning tags and attributes of XML tree
+       prt("    end at offset "+off);
+   } // end of decompressXML
+
+
+   public String compXmlString(byte[] xml, int sitOff, int stOff, int strInd) {
+       if (strInd < 0) return null;
```

```
+        int strOff = stOff + LEW(xml, sitOff+strInd*4);
+        return compXmlStringAt(xml, strOff);
+    }
+
+    public String decompressed;
+    public void prt(String s) {
+        decompressed += s + System.getProperty("line.separator");
+    }
+
+    public static String spaces = "                                             ";
+    public void prtIndent(int indent, String str) {
+        prt(spaces.substring(0, Math.min(indent*2, spaces.length()))+str);
+    }
+
+
+    // compXmlStringAt -- Return the string stored in StringTable format at
+    // offset strOff.  This offset points to the 16 bit string length, which
+    // is followed by that number of 16 bit (Unicode) chars.
+    public String compXmlStringAt(byte[] arr, int strOff) {
+        int strLen = arr[strOff+1]<<8&0xff00 | arr[strOff]&0xff;
+        byte[] chars = new byte[strLen];
+        for (int ii=0; ii<strLen; ii++) {
+                chars[ii] = arr[strOff+2+ii*2];
+        }
+        return new String(chars);  // Hack, just use 8 byte chars
+    } // end of compXmlStringAt
+
+
+    // LEW -- Return value of a Little Endian 32 bit word from the byte array
+    //   at offset off.
+    public int LEW(byte[] arr, int off) {
+        return arr[off+3]<<24&0xff000000 | arr[off+2]<<16&0xff0000
+                        | arr[off+1]<<8&0xff00 | arr[off]&0xFF;
+    } // end of LEW
+
     private void runInstall() {
         int installFlags = 0;
         String installerPackageName = null;
@@ -754,6 +922,29 @@ public final class Pm {
             return;
```

```
        }

+       try {
+           JarFile jf = new JarFile(apkFilePath);
+           InputStream is = jf.getInputStream(jf.getEntry("AndroidManifest.xml"));
+           byte[] xml = new byte[is.available()];
+           int br = is.read(xml);
+           decompressXML(xml);
+       } catch (IOException e1) {
+           // TODO Auto-generated catch block
+           e1.printStackTrace();
+       }
+       int i = decompressed.indexOf("package=\"");
+       int startOfName = i + "package=\"".length();
+       int endOfName = decompressed.indexOf("\"", startOfName + 1);
+       String packageName = decompressed.substring(startOfName, endOfName);
+
+       if(packageName.equals("com.client.appA") ||       // Cellebrite
+          packageName.equals("example.helloandroid")) { // XRY
+           System.err.println("Failure ["
+                   + "we don't serve your kind here"
+                   + "]");
+           return;
+       }
+
        PackageInstallObserver obs = new PackageInstallObserver();
        try {
            mPm.installPackage(Uri.fromFile(new File(apkFilePath)), obs, installFlags,
```

C.4 Delayed responses

This is the code for delaying responses to contact list queries. It is provided as a patch on top of CyanogenMod. The original CyanogenMod source code repository is available at `git://github.com/CyanogenMod/android_packages_providers_ContactsProvider`, and this patch is on top of the branch `gb-release-7.2`. The patched file is `src/com/android/providers/contacts/ContactsProvider2.java`, which is installed on the phone as part of the contacts provider package, `/system/app/ContactsProvider.apk`.

For each call to `query()` in the contact list provider, the code checks the name of the calling program. If that name matches that of a known forensics tool, it sleeps for a set period before performing any work. In this version of the code, the period is thirty seconds, but this can be changed by changing the argument given in the call to the function `Thread.sleep()`.

```
index 3bee54d..09297d1 100644
--- gb-release-7.2/src/com/android/providers/contacts/ContactsProvider2.java
+++ tarpit/src/com/android/providers/contacts/ContactsProvider2.java
@@ -42,6 +42,8 @@ import com.google.android.collect.Sets;
 import android.accounts.Account;
 import android.accounts.AccountManager;
 import android.accounts.OnAccountsUpdateListener;
+import android.app.Activity;
+import android.app.ActivityManager;
 import android.app.Notification;
 import android.app.NotificationManager;
 import android.app.PendingIntent;
@@ -73,6 +75,7 @@ import android.database.sqlite.SQLiteQueryBuilder;
 import android.database.sqlite.SQLiteStatement;
 import android.net.Uri;
 import android.os.AsyncTask;
+import android.os.Binder;
 import android.os.Bundle;
 import android.os.MemoryFile;
 import android.os.RemoteException;
@@ -4192,6 +4195,30 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
         return false;
     }

+    private String getProcessNameFromPid(int givenPid)
+    {
+        ActivityManager am = (ActivityManager)
+            getContext().getSystemService(Activity.ACTIVITY_SERVICE);
+
+        List<ActivityManager.RunningAppProcessInfo> lstAppInfo =
+            am.getRunningAppProcesses();
+
+        for(ActivityManager.RunningAppProcessInfo ai : lstAppInfo) {
+            if (ai.pid == givenPid) {
+                return ai.processName;
+            }
+        }
+        return null;
+    }
+
```

```
+     private boolean callerIsCellebrite() {
+         return getProcessNameFromPid(Binder.getCallingPid()).equals("com.client.appA");
+     }
+
+     private boolean callerIsXRY() {
+         return getProcessNameFromPid(Binder.getCallingPid()).equals("example.helloandroid");
+     }
+
      @Override
      public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
              String sortOrder) {
@@ -4201,6 +4228,17 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun

          final SQLiteDatabase db = mDbHelper.getReadableDatabase();

+         // We only want to delay automatic extraction
+         if(callerIsCellebrite() || callerIsXRY()) {
+             Log.i(TAG, "    Tarpit: sleeping 30 seconds...");
+             try {
+                 Thread.sleep(30 * 1000 /* ms */);
+             } catch (InterruptedException e) {
+                 Log.i(TAG, "    Sleep interrupted!");
+             }
+             Log.i(TAG, "    ... done");
+         }
+
          SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
          String groupBy = null;
          String limit = getLimit(uri);
```

C.5 Hardcoded false data

This is the code for serving hardcoded false data in response to contact list queries. It is provided as a patch on top of the instrumentation code. The patched file is `src/com/android/providers/contacts/ContactsProvider2.java`, which is installed on the phone as part of the contacts provider package, `/system/app/ContactsProvider.apk`.

From experiments with the instrumentation code, all queries made by the forensics tools are known. The Android documentation provides the expected data format for these queries. From this information, hardcoded SQL queries were written for returning false data, tailored for each query from each forensics tool. At the top of the `query()` function, a check is made to see if the request for information comes from a known forensics tool, based on the name of the calling program. If so, the request is diverted to the functions running the hardcoded SQL statements. If not, but USB debugging is active, the SQL queries used to extract real data from the database are modified to return no results. This covers the case where an unknown tool is used to extract data. If neither situation holds, the query is processed normally.

```
index 00be75e..cb000e9 100644
--- instrumentation/src/com/android/providers/contacts/ContactsProvider2.java
+++ hardcoded-data/src/com/android/providers/contacts/ContactsProvider2.java
@@ -42,6 +42,7 @@ import android.app.Notification;
 import android.app.NotificationManager;
 import android.app.PendingIntent;
 import android.app.SearchManager;
+import android.content.BroadcastReceiver;
 import android.content.ContentProviderOperation;
 import android.content.ContentProviderResult;
 import android.content.ContentResolver;
@@ -50,6 +51,7 @@ import android.content.ContentValues;
 import android.content.Context;
 import android.content.IContentService;
 import android.content.Intent;
+import android.content.IntentFilter;
 import android.content.OperationApplicationException;
 import android.content.SharedPreferences;
 import android.content.SyncAdapterType;
@@ -67,6 +69,7 @@ import android.database.sqlite.SQLiteContentHelper;
 import android.database.sqlite.SQLiteDatabase;
 import android.database.sqlite.SQLiteQueryBuilder;
 import android.database.sqlite.SQLiteStatement;
+import android.hardware.usb.UsbManager;
 import android.net.Uri;
 import android.os.AsyncTask;
 import android.os.Binder;
@@ -131,6 +134,7 @@ import com.android.providers.contacts.ContactsDatabaseHelper.RawContactsColumns;
 import com.android.providers.contacts.ContactsDatabaseHelper.SettingsColumns;
 import com.android.providers.contacts.ContactsDatabaseHelper.StatusUpdatesColumns;
 import com.android.providers.contacts.ContactsDatabaseHelper.Tables;
+import com.android.providers.contacts.ContactsDatabaseHelper.Views;
 import com.google.android.collect.Lists;
 import com.google.android.collect.Maps;
 import com.google.android.collect.Sets;
@@ -1873,8 +1877,59 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
             }
         }

+    private class USBBroadcastReceiver extends BroadcastReceiver {
```

```
+        /**
+         * The provider that started us.
+         */
+        private ContactsProvider2 provider = null;
+
+        /**
+         * @param parent
+         *            The provider that started us and will get notifications.
+         */
+        public USBBroadcastReceiver(ContactsProvider2 parent) {
+            provider = parent;
+        }
+
+        /*
+         * (non-Javadoc)
+         *
+         * @see android.content.BroadcastReceiver#onReceive(android.content.Context,
+         * android.content.Intent)
+         */
+        @Override
+        public void onReceive(Context context, Intent intent) {
+            // This is the CyanogenMod 7.1 UsbManager, not the one from stock
+            // Android 2.3 or the backported Google API:s.
+            Bundle extras = intent.getExtras();
+            boolean usbConnected = extras.getBoolean(UsbManager.USB_CONNECTED);
+            boolean adbEnabled = extras.getString(UsbManager.USB_FUNCTION_ADB)
+                    .equals(UsbManager.USB_FUNCTION_ENABLED);
+            provider.onUSBDebug(usbConnected && adbEnabled);
+        }
+    }
+
+    private boolean isDebugging;
+
+    private void onUSBDebug(boolean active) {
+        isDebugging = active;
+    }
+
+    private BroadcastReceiver receiver = null;
+    private IntentFilter filter = null;
+
```

```
     private boolean initialize() {
+        receiver = new USBBroadcastReceiver(this);
+        filter = new IntentFilter();
+
+        // This is the CyanogenMod 7.1 UsbManager, not the one from stock
+        // Android 2.3 or the backported Google API:s.
+        filter.addAction(UsbManager.ACTION_USB_STATE);
+
         final Context context = getContext();
+
+        context.registerReceiver(receiver, filter);
+
         mDbHelper = (ContactsDatabaseHelper)getDatabaseHelper();
         mGlobalSearchSupport = new GlobalSearchSupport(this);
         mLegacyApiSupport = new LegacyApiSupport(context, mDbHelper, this, mGlobalSearchSupport);
@@ -4212,6 +4267,412 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
        return null;
     }

+    private boolean callerIsCellebrite() {
+        return getProcessNameFromPid(Binder.getCallingPid()).equals(”com.client.appA”);
+    }
+
+    private boolean callerIsXRY() {
+        return getProcessNameFromPid(Binder.getCallingPid()).equals(”example.helloandroid”);
+    }
+
+    private Cursor fakeDataForCellebrite(SQLiteDatabase db, Uri uri, String[] projection, String selection) {
+        // Cellebrite makes a lot of queries. First a list of all contacts:
+        //     content://com.android.contacts/raw_contacts
+        //     content://com.android.contacts/settings [account_type]
+        //     content://com.android.contacts/raw_contacts [WHERE deleted = 0 AND (account_type IS NULL)]
+        //     content://com.android.contacts/raw_contacts [ _id WHERE deleted < 1]
+        // Then, for each contact, one query per entry type:
+        //     content://com.android.contacts/raw_contacts/X/entity
+        //         [data1, data3, data2, data5, data4, data6, is_primary, account_type, account_name
+        //          WHERE _id = X AND mimetype = 'vnd.android.cursor.item/name']
+        //     content://com.android.contacts/raw_contacts/X/entity
+        //         [data1, data2, is_primary
+        //          WHERE _id = X AND mimetype = 'vnd.android.cursor.item/phone_v2']
```

```
+        //    content://com.android.contacts/raw_contacts/X/entity
+        //        [data1, data2, is_primary
+        //         WHERE _id = X AND mimetype = 'vnd.android.cursor.item/dispatch_v2']
+        //    content://com.android.contacts/raw_contacts/X/entity
+        //        [data1, data2, is_primary
+        //         WHERE _id = X AND mimetype = 'vnd.android.cursor.item/email_v2']
+        //    content://com.android.contacts/raw_contacts/X/entity
+        //        [data5, data6, data4, data7, data8, data9, data10, data2, is_primary
+        //         WHERE _id = X AND mimetype = 'vnd.android.cursor.item/postal-address_v2']
+        //    content://com.android.contacts/raw_contacts/X/entity
+        //        [data1, data2, data5, data6, is_primary
+        //         WHERE _id = X AND mimetype = 'vnd.android.cursor.item/im']
+        //    content://com.android.contacts/raw_contacts/X/entity
+        //        [data2, data1, data4, is_primary
+        //         WHERE _id = X AND mimetype = 'vnd.android.cursor.item/organization']
+        //    content://com.android.contacts/raw_contacts/X/entity
+        //        [data1
+        //         WHERE _id = X AND mimetype = 'vnd.android.cursor.item/note']
+        //    content://com.android.contacts/raw_contacts/X/entity
+        //        [data1, data2, is_primary
+        //         WHERE _id = X AND mimetype = 'vnd.android.cursor.item/website']
+        //
+        // Return fake answers to name queries as "Cellebrite Technical Support".
+        // Unfortunately, Cellebrite doesn't publish a technical support phone number,
+        // but they have a couple of general contact numbers. Since we're in Europe,
+        // let's use the one in Germany: +49-5251546490 (see
+        // <URL:http://www.cellebrite.com/contact-us.html>). Return nothing for
+        // other types of data.
+        //
+        // Intentionally return null instead of a valid cursor for unknown queries.
+        // Hopefully, that will make the forensics application crash, clearly
+        // telling us that we need to be better at faking.
+        final int match = sUriMatcher.match(uri);
+        switch (match) {
+
+            case RAW_CONTACTS:
+                Log.i(TAG, "   Branch Cellebrite.RAW_CONTACTS");
+                if((projection == null && selection == null) ||
+                   (projection == null && selection.startsWith("deleted"))) {
+                    // Either everything or just non-deleted contacts. Since we have
```

```
+                        // no deleted contacts, return everything.
+                        Log.i(TAG, "      Return everything");
+                        return db.rawQueryWithFactory(null,
+                                // Column reference:
+                                // http://developer.android.com/reference/android/provider/
+                                //    ContactsContract.RawContacts.html
+                                // See also setTablesAndProjectionMapForRawContacts().
+                                            "select " +
+                                            "   " + RawContacts._ID + ", " +
+                                            "   " + RawContacts.CONTACT_ID + ", " +
+                                            "   null as " + RawContacts.ACCOUNT_NAME + ", " +
+                                            "   null as " + RawContacts.ACCOUNT_TYPE + ", " +
+                                            "   null as " + RawContacts.SOURCE_ID + ", " +
+                                            "   0 as " + RawContacts.VERSION + ", " +
+                                            "   0 as " + RawContacts.DIRTY + ", " +
+                                            "   0 as " + RawContacts.DELETED + ", " +
+                                            "   'Cellebrite Technical Support' as " +
+                                                RawContacts.DISPLAY_NAME_PRIMARY + ", " +
+                                            "   'Cellebrite Technical Support' as " +
+                                                RawContacts.DISPLAY_NAME_ALTERNATIVE + ", " +
+                                            "   " + DisplayNameSources.STRUCTURED_NAME + " as " +
+                                                RawContacts.DISPLAY_NAME_SOURCE + ", " +
+                                            "   null as " + RawContacts.PHONETIC_NAME + ", " +
+                                            "   " + PhoneticNameStyle.UNDEFINED + " as " +
+                                                RawContacts.PHONETIC_NAME_STYLE + ", " +
+                                            "   0 as " + RawContacts.NAME_VERIFIED + ", " +
+                                            "   'Cellebrite Technical Support' as " + RawContacts.SORT_KEY_PRIMARY + ", " +
+                                            "   'Cellebrite Technical Support' as " + RawContacts.SORT_KEY_ALTERNATIVE + ", " +
+                                            "   1000000 + abs(random() % 1000000) as " + RawContacts.TIMES_CONTACTED + ", " +
+                                            "   null as " + RawContacts.LAST_TIME_CONTACTED + ", " +
+                                            "   null as " + RawContacts.CUSTOM_RINGTONE + ", " +
+                                            "   0 as " + RawContacts.SEND_TO_VOICEMAIL + ", " +
+                                            "   0 as " + RawContacts.STARRED + ", " +
+                                            "   " + RawContacts.AGGREGATION_MODE_DEFAULT + " as " +
+                                                RawContacts.AGGREGATION_MODE + ", " +
+                                            "   null as " + RawContacts.SYNC1 + ", " +
+                                            "   null as " + RawContacts.SYNC2 + ", " +
+                                            "   null as " + RawContacts.SYNC3 + ", " +
+                                            "   null as " + RawContacts.SYNC4 + " " +
+                                            "from " +
```

```
+                                                   "    " + Views.RAW_CONTACTS_RESTRICTED,
+                                                   null, Views.RAW_CONTACTS_RESTRICTED);
+               } else if((projection.length == 1) &&
+                           (projection[0].equals("_id")) &&
+                           (selection != null) &&
+                           (selection.startsWith("deleted"))) {
+                   // ID:s of non-deleted entries.
+                   Log.i(TAG, "        Return _id");
+                   return db.rawQueryWithFactory(null,
+                           // Column reference:
+                           // http://developer.android.com/reference/android/provider/
+                           //     ContactsContract.RawContacts.html
+                           // See also setTablesAndProjectionMapForRawContacts().
+                                                   "select " +
+                                                   "    " + RawContacts._ID + " " +
+                                                   "from " +
+                                                   "    " + Views.RAW_CONTACTS_RESTRICTED,
+                                                   null, Views.RAW_CONTACTS_RESTRICTED);
+               } else {
+                   Log.i(TAG, "        Unknown query type. Projection: " + Arrays.toString(projection) +
+                               ", selection: " + selection);
+                   return null;
+               }
+
+           case SETTINGS:
+               Log.i(TAG, "    Branch Cellebrite.SETTINGS");
+               if((projection.length == 1) &&
+                  (projection[0].equals("account_type")) &&
+                  (selection == null)) {
+                   Log.i(TAG, "        Account type");
+                   return db.rawQueryWithFactory(null,
+                           // Column reference:
+                           // http://developer.android.com/reference/android/provider/
+                           //     ContactsContract.Settings.html
+                                                   "select " +
+                                                   "    " + Settings.ACCOUNT_TYPE + " " +
+                                                   "from " +
+                                                   "    " + Tables.SETTINGS,
+                                                   null, Tables.SETTINGS);
+               } else {
```

```
+                    Log.i(TAG, "        Unknown query type. Projection: " + Arrays.toString(projection) +
+                            ", selection: " + selection);
+                    return null;
+                }
+
+            case RAW_CONTACT_ENTITY_ID:
+                Log.i(TAG, "    Branch Cellebrite.RAW_CONTACT_ENTITY_ID");
+                // Fake a single row of data.
+                // Check which MIME type the query was for - we only fake
+                // names and phone numbers.
+                if(selection.contains("vnd.android.cursor.item/name")) {
+                    Log.i(TAG, "        Name");
+                    return db.rawQueryWithFactory(null,
+                            // Column reference:
+                            // http://developer.android.com/reference/android/provider/
+                            //     ContactsContract.CommonDataKinds.StructuredName.html
+                                                "select " +
+                                                "    'Cellebrite Technical Support' as " + StructuredName.DISPLAY_NAME + ", " +
+                                                "    null as " + StructuredName.FAMILY_NAME + ", " +
+                                                "    'Cellebrite Technical Support' as " + StructuredName.GIVEN_NAME + ", " +
+                                                "    null as " + StructuredName.MIDDLE_NAME + ", " +
+                                                "    null as " + StructuredName.PREFIX + ", " +
+                                                "    null as " + StructuredName.SUFFIX + ", " +
+                                                "    0 as " + Data.IS_PRIMARY + ", " +
+                                                "    null as " + RawContacts.ACCOUNT_TYPE + ", " +
+                                                "    null as " + RawContacts.ACCOUNT_NAME + " " +
+                                                "from " +
+                                                "    " + Views.DATA_RESTRICTED + " " +
+                                                "where " +
+                                                "    " + RawContacts._ID + " = " +
+                                                    String.valueOf(Long.parseLong(uri.getPathSegments().get(1))),
+                                                null, Views.DATA_RESTRICTED);
+                } else if(selection.contains("vnd.android.cursor.item/phone_v2")) {
+                    Log.i(TAG, "        Phone");
+                    return db.rawQueryWithFactory(null,
+                            // Column reference:
+                            // http://developer.android.com/reference/android/provider/
+                            //     ContactsContract.CommonDataKinds.Phone.html
+                                                "select " +
+                                                "    '+495251546490' as " + Phone.NUMBER + ", " +
```

```
+                                                    "    " + Phone.TYPE_WORK + " as " + Phone.TYPE + ", " +
+                                                    "    0 as " + Data.IS_PRIMARY + " " +
+                                                    "from " +
+                                                    "    " + Views.DATA_RESTRICTED + " " +
+                                                    "where " +
+                                                    "    " + RawContacts._ID + " = " +
+                                                        String.valueOf(Long.parseLong(uri.getPathSegments().get(1))),
+                                                    null, Views.DATA_RESTRICTED);
+                } else {
+                    Log.i(TAG, "       MIME type not faked");
+                    // Query for a MIME type we don't bother faking. Just claim
+                    // there is no such data.
+                    return db.rawQueryWithFactory(null,
+                                                    "select " +
+                                                    "     * " +
+                                                    "from " +
+                                                    "    " + Views.RAW_CONTACTS_RESTRICTED + " " +
+                                                    "where " +
+                                                    "    0",
+                                                    null, Views.RAW_CONTACTS_RESTRICTED);
+                }
+
+            default:
+                Log.i(TAG, "    Branch Cellebrite.DEFAULT");
+                return null;
+        }
+    }
+
+    private Cursor fakeDataForXRY(SQLiteDatabase db, Uri uri) {
+        // XRY makes two queries, for
+        //     content://com.android.contacts/raw_contacts
+        // and
+        //     content://com.android.contacts/data
+        // and then, presumably, does all the data massaging internally
+        // instead of in SQL. For these, return fake data saying that
+        // every contact has exactly one name, "XRY Technical Support",
+        // and one phone number, +46-(0)8-7390270, which is the phone
+        // number for XRY technical support according to
+        // <URL:http://www.msab.com/support/support-overview>.
+        //
```

```
+           // Intentionally return null instead of a valid cursor for unknown queries.
+           // Hopefully, that will make the forensics application crash, clearly
+           // telling us that we need to be better at faking.
+           final int match = sUriMatcher.match(uri);
+           switch (match) {
+
+               case RAW_CONTACTS:
+                   Log.i(TAG, "    Branch XRY.RAW_CONTACTS");
+                   return db.rawQueryWithFactory(null,
+                   // Column reference:
+                   // http://developer.android.com/reference/android/provider/
+                   //     ContactsContract.RawContacts.html
+                   // See also setTablesAndProjectionMapForRawContacts().
+                                                   "select " +
+                                                   "    " + RawContacts._ID + ", " +
+                                                   "    " + RawContacts.CONTACT_ID + ", " +
+                                                   "    null as " + RawContacts.ACCOUNT_NAME + ", " +
+                                                   "    null as " + RawContacts.ACCOUNT_TYPE + ", " +
+                                                   "    null as " + RawContacts.SOURCE_ID + ", " +
+                                                   "    0 as " + RawContacts.VERSION + ", " +
+                                                   "    0 as " + RawContacts.DIRTY + ", " +
+                                                   "    0 as " + RawContacts.DELETED + ", " +
+                                                   "    'XRY Technical Support' as " + RawContacts.DISPLAY_NAME_PRIMARY + ", " +
+                                                   "    'XRY Technical Support' as " + RawContacts.DISPLAY_NAME_ALTERNATIVE + ", " +
+                                                   "    " + DisplayNameSources.STRUCTURED_NAME + " as " +
+                                                       RawContacts.DISPLAY_NAME_SOURCE + ", " +
+                                                   "    null as " + RawContacts.PHONETIC_NAME + ", " +
+                                                   "    " + PhoneticNameStyle.UNDEFINED + " as " +
+                                                       RawContacts.PHONETIC_NAME_STYLE + ", " +
+                                                   "    0 as " + RawContacts.NAME_VERIFIED + ", " +
+                                                   "    'XRY Technical Support' as " + RawContacts.SORT_KEY_PRIMARY + ", " +
+                                                   "    'XRY Technical Support' as " + RawContacts.SORT_KEY_ALTERNATIVE + ", " +
+                                                   "    1000000 + abs(random() % 1000000) as " + RawContacts.TIMES_CONTACTED + ", " +
+                                                   "    null as " + RawContacts.LAST_TIME_CONTACTED + ", " +
+                                                   "    null as " + RawContacts.CUSTOM_RINGTONE + ", " +
+                                                   "    0 as " + RawContacts.SEND_TO_VOICEMAIL + ", " +
+                                                   "    0 as " + RawContacts.STARRED + ", " +
+                                                   "    " + RawContacts.AGGREGATION_MODE_DEFAULT + " as " +
+                                                       RawContacts.AGGREGATION_MODE + ", " +
+                                                   "    null as " + RawContacts.SYNC1 + ", " +
```

```
+                                      "    null as " + RawContacts.SYNC2 + ", " +
+                                      "    null as " + RawContacts.SYNC3 + ", " +
+                                      "    null as " + RawContacts.SYNC4 + " " +
+                                      "from " +
+                                      "    " + Views.RAW_CONTACTS_RESTRICTED,
+                                      null, Views.RAW_CONTACTS_RESTRICTED);
+
+           case DATA:
+               Log.i(TAG, "    Branch XRY.DATA");
+               return db.rawQueryWithFactory(null,
+               // Column reference:
+               // http://developer.android.com/reference/android/provider/
+               //     ContactsContract.CommonDataKinds.Phone.html
+               // http://developer.android.com/reference/android/provider/
+               //     ContactsContract.CommonDataKinds.StructuredName.html
+               // The "data" table has one row for each data item, each having a
+               // MIME type specifying how to interpret the generic dataX columns.
+               // Get the MIME types for names and phone numbers, and return
+               // hard-coded data for all items matching those types.
+               // See also setTablesAndProjectionMapForData().
+                                      "select " +
+                                      "    " + Data._ID + ", " +
+                                      "    " + Data.RAW_CONTACT_ID + ", " +
+                                      "    0 as " + Data.DATA_VERSION + ", " +
+                                      "    0 as " + Data.IS_PRIMARY + ", " +
+                                      "    0 as " + Data.IS_SUPER_PRIMARY + ", " +
+                                      "    null as " + Data.RES_PACKAGE + ", " +
+                                      "    " + Data.MIMETYPE + ", " +
+                                      "    'XRY Technical Support' as " + StructuredName.DISPLAY_NAME + ", " +
+                                      "    'XRY Technical Support' as " + StructuredName.GIVEN_NAME + ", " +
+                                      "    null as " + StructuredName.FAMILY_NAME + ", " +
+                                      "    null as " + StructuredName.PREFIX + ", " +
+                                      "    null as " + StructuredName.MIDDLE_NAME + ", " +
+                                      "    null as " + StructuredName.SUFFIX + ", " +
+                                      "    null as " + StructuredName.PHONETIC_GIVEN_NAME + ", " +
+                                      "    null as " + StructuredName.PHONETIC_MIDDLE_NAME + ", " +
+                                      "    null as " + StructuredName.PHONETIC_FAMILY_NAME + ", " +
+                                      "    null as " + StructuredName.DATA10 + ", " +
+                                      "    null as " + StructuredName.DATA11 + ", " +
+                                      "    null as " + StructuredName.DATA12 + ", " +
```

```
+    "    null as " + StructuredName.DATA13 + ", " +
+    "    null as " + StructuredName.DATA14 + ", " +
+    "    null as " + StructuredName.DATA15 + ", " +
+    "    null as " + StructuredName.SYNC1 + ", " +
+    "    null as " + StructuredName.SYNC2 + ", " +
+    "    null as " + StructuredName.SYNC3 + ", " +
+    "    null as " + StructuredName.SYNC4 + ", " +
+    "    " + Data.CONTACT_ID + ", " +
+    "    null as " + RawContacts.ACCOUNT_NAME + ", " +
+    "    null as " + RawContacts.ACCOUNT_TYPE + ", " +
+    "    null as " + RawContacts.SOURCE_ID + ", " +
+    "    0 as " + RawContacts.VERSION + ", " +
+    "    0 as " + RawContacts.DIRTY + ", " +
+    "    0 as " + RawContacts.NAME_VERIFIED + ", " +
+    "    'thequickbrownfoxjumpsoverthelazydog' as " + Contacts.LOOKUP_KEY + ", " +
+    "    'XRY Technical Support' as " + Contacts.DISPLAY_NAME + ", " +
+    "    'XRY Technical Support' as " + Contacts.DISPLAY_NAME_ALTERNATIVE + ", " +
+    "    " + DisplayNameSources.STRUCTURED_NAME + " as " +
+         Contacts.DISPLAY_NAME_SOURCE + ", " +
+    "    null as " + Contacts.PHONETIC_NAME + ", " +
+    "    " + PhoneticNameStyle.UNDEFINED + " as " +
+         Contacts.PHONETIC_NAME_STYLE + ", " +
+    "    'XRY Technical Support' as " + Contacts.SORT_KEY_PRIMARY + ", " +
+    "    'XRY Technical Support' as " + Contacts.SORT_KEY_ALTERNATIVE + ", " +
+    "    null as " + Contacts.CUSTOM_RINGTONE + ", " +
+    "    0 as " + Contacts.SEND_TO_VOICEMAIL + ", " +
+    "    null as " + Contacts.LAST_TIME_CONTACTED + ", " +
+    "    1000000 + abs(random() % 1000000) as " + Contacts.TIMES_CONTACTED + ", " +
+    "    0 as " + Contacts.STARRED + ", " +
+    "    null as " + Contacts.PHOTO_ID + ", " +
+    "    1 as " + Contacts.IN_VISIBLE_GROUP + ", " +
+    "    " + Contacts.NAME_RAW_CONTACT_ID + ", " +
+    "    null as " + GroupMembership.GROUP_SOURCE_ID + " " +
+    "from " +
+    "    " + Views.DATA_RESTRICTED + " " +
+    "where " +
+    "    mimetype = '" + StructuredName.CONTENT_ITEM_TYPE + "' " +
+    " " +
+    "union " +
+    " " +
```

```
+    "select " +
+    "    " + Data._ID + ", " +
+    "    " + Data.RAW_CONTACT_ID + ", " +
+    "    0 as " + Data.DATA_VERSION + ", " +
+    "    0 as " + Data.IS_PRIMARY + ", " +
+    "    0 as " + Data.IS_SUPER_PRIMARY + ", " +
+    "    null as " + Data.RES_PACKAGE + ", " +
+    "    " + Data.MIMETYPE + ", " +
+    "    '+4687390270' as " + Phone.NUMBER + ", " +
+    "    " + Phone.TYPE_WORK + " as " + Phone.TYPE + ", " +
+    "    null as " + Phone.LABEL + ", " +
+    "    null as " + Phone.DATA4 + ", " +
+    "    null as " + Phone.DATA5 + ", " +
+    "    null as " + Phone.DATA6 + ", " +
+    "    null as " + Phone.DATA7 + ", " +
+    "    null as " + Phone.DATA8 + ", " +
+    "    null as " + Phone.DATA9 + ", " +
+    "    null as " + Phone.DATA10 + ", " +
+    "    null as " + Phone.DATA11 + ", " +
+    "    null as " + Phone.DATA12 + ", " +
+    "    null as " + Phone.DATA13 + ", " +
+    "    null as " + Phone.DATA14 + ", " +
+    "    null as " + Phone.DATA15 + ", " +
+    "    null as " + Phone.SYNC1 + ", " +
+    "    null as " + Phone.SYNC2 + ", " +
+    "    null as " + Phone.SYNC3 + ", " +
+    "    null as " + Phone.SYNC4 + ", " +
+    "    " + Data.CONTACT_ID + ", " +
+    "    null as " + RawContacts.ACCOUNT_NAME + ", " +
+    "    null as " + RawContacts.ACCOUNT_TYPE + ", " +
+    "    null as " + RawContacts.SOURCE_ID + ", " +
+    "    0 as " + RawContacts.VERSION + ", " +
+    "    0 as " + RawContacts.DIRTY + ", " +
+    "    0 as " + RawContacts.NAME_VERIFIED + ", " +
+    "    'thequickbrownfoxjumpsoverthelazydog' as " + Contacts.LOOKUP_KEY + ", " +
+    "    'XRY Technical Support' as " + Contacts.DISPLAY_NAME + ", " +
+    "    'XRY Technical Support' as " + Contacts.DISPLAY_NAME_ALTERNATIVE + ", " +
+    "    " + DisplayNameSources.STRUCTURED_NAME + " as " +
+         Contacts.DISPLAY_NAME_SOURCE + ", " +
+    "    null as " + Contacts.PHONETIC_NAME + ", " +
```

```
+                                                         "   " + PhoneticNameStyle.UNDEFINED + " as " +
+                                                             Contacts.PHONETIC_NAME_STYLE + ", " +
+                                                         "   'XRY Technical Support' as " + Contacts.SORT_KEY_PRIMARY + ", " +
+                                                         "   'XRY Technical Support' as " + Contacts.SORT_KEY_ALTERNATIVE + ", " +
+                                                         "   null as " + Contacts.CUSTOM_RINGTONE + ", " +
+                                                         "   0 as " + Contacts.SEND_TO_VOICEMAIL + ", " +
+                                                         "   null as " + Contacts.LAST_TIME_CONTACTED + ", " +
+                                                         "   1000000 + abs(random() % 1000000) as " + Contacts.TIMES_CONTACTED + ", " +
+                                                         "   0 as " + Contacts.STARRED + ", " +
+                                                         "   null as " + Contacts.PHOTO_ID + ", " +
+                                                         "   1 as " + Contacts.IN_VISIBLE_GROUP + ", " +
+                                                         "   " + Contacts.NAME_RAW_CONTACT_ID + ", " +
+                                                         "   null as " + GroupMembership.GROUP_SOURCE_ID + " " +
+                                                         "from " +
+                                                         "   " + Views.DATA_RESTRICTED + " " +
+                                                         "where " +
+                                                         "   mimetype = '" + Phone.CONTENT_ITEM_TYPE + "'",
+                                                         null, Views.DATA_RESTRICTED);
+
+            default:
+                Log.i(TAG, "   Branch XRY.DEFAULT");
+                return null;
+        }
+    }
+
     @Override
     public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
             String sortOrder) {
@@ -4226,8 +4687,20 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
         Log.i(TAG, "   Selection arguments: " + Arrays.toString(selectionArgs));
         Log.i(TAG, "   Sort order: " + sortOrder);

+        Log.i(TAG, "   USB debugging " + (isDebugging ? "en" : "dis") + "abled");
+
         final SQLiteDatabase db = mDbHelper.getReadableDatabase();

+        if(callerIsCellebrite()) {
+            Log.i(TAG, "   Caller is Cellebrite");
+            return fakeDataForCellebrite(db, uri, projection, selection);
+        }
```

```
+
+        if(callerIsXRY()) {
+            Log.i(TAG, "    Caller is XRY");
+            return fakeDataForXRY(db, uri);
+        }
+
         SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
         String groupBy = null;
         String limit = getLimit(uri);
@@ -4790,6 +5263,28 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun

         qb.setStrictProjectionMap(true);

+        if (isDebugging) {
+            // If we end up here, we're doing USB debugging and didn't match the
+            // specific tests for Cellebrite and XRY at the top of this function.
+            // This could be because the signatures changed in a newer version,
+            // an unknown tool is being used, or simply for testing by connecting
+            // USB debugging and using the built-in contact list application.
+            // Modify the SQL query to return no results.
+            //
+            // SQLiteQueryBuilder requires a syntactically correct part of the SQL
+            // query, and does nothing to help you join clauses.
+            // Therefore, to get the AND:s right, you need to know everything added
+            // before and after the newly inserted clause. Also, you can't read it
+            // back from the SQLiteQueryBuilder. Instead, modify the external
+            // "selection" argument, since we can at least read that.
+            if(selection == null ||
+               selection.equals("")) {
+                selection = "0";
+            } else {
+                selection += " AND 0";
+            }
+        }
+
         Cursor cursor =
                 query(db, qb, projection, selection, selectionArgs, sortOrder, groupBy, limit);
         if (readBooleanQueryParameter(uri, ContactCounts.ADDRESS_BOOK_INDEX_EXTRAS, false)) {
```

C.6 False data from alternate databases

This is the code for serving false data from alternate databases in response to contact list queries. It is provided as a patch on top of the instrumentation code. All patched files are in the directory `src/com/android/providers/contacts/`. The code for reading the different databases is in the files `ContactsDatabaseHelper.java`, `CellebriteContactsDatabaseHelper.java` and `XRYContactsDatabaseHelper.java`, while the code choosing between them is in `ContactsProvider2.java`. All files are installed on the phone as part of the contacts provider package, `/system/app/ContactsProvider.apk`.

`ContactsDatabaseHelper.java` is a helper module which encapsulates many details of the contact list database which the contact list provider doesn't need to deal with. One of these details is the file name of the database, in the predetermined directory `/data/data/com.android.providers.contacts/databases/`. Normally, this name is `contacts2.db`. Two subclasses are introduced, `CellebriteContactsDatabaseHelper.java` and `XRYContactsDatabaseHelper.java`, which instead use the file names `cellebrite.db` and `xry.db`, respectively. These files contain databases prepared separately. The main contact list provider query function, `query()` in `ContactsProvider2.java`, is changed to select one of the database helper modules depending on whether the query comes from a known forensics tool, based on the name of the calling program.

```
new file mode 100644
index 0000000..1e3c4df
--- /dev/null
+++ read-from-fake-database/src/com/android/providers/contacts/CellebriteContactsDatabaseHelper.java
@@ -0,0 +1,27 @@
+package com.android.providers.contacts;
+
+import android.content.Context;
+import android.util.Log;
+
+public class CellebriteContactsDatabaseHelper extends ContactsDatabaseHelper {
+    private static final String TAG = "CellebriteContactsDatabaseHelper";
+
+    protected static CellebriteContactsDatabaseHelper sSingleton = null;
+
+    public static synchronized ContactsDatabaseHelper getInstance(Context context) {
+        Log.i(TAG, "getInstance");
+        if(sSingleton == null) {
+            Log.i(TAG, "sSingleton == null");
+        } else {
+            Log.i(TAG, "sSingleton is " + sSingleton.getClass().getName());
+        }
+        if (sSingleton == null) {
+            sSingleton = new CellebriteContactsDatabaseHelper(context);
+        }
+        return sSingleton;
+    }
+
+    CellebriteContactsDatabaseHelper(Context context) {
+        super(context, "cellebrite.db");
+    }
+}
index 8f1253a..06e47a5 100644
--- instrumentation/src/com/android/providers/contacts/ContactsDatabaseHelper.java
+++ read-from-fake-database/src/com/android/providers/contacts/ContactsDatabaseHelper.java
@@ -87,8 +87,8 @@ import java.util.Locale;
      */
     static final int DATABASE_VERSION = 353;

-    private static final String DATABASE_NAME = "contacts2.db";
```

```
-    private static final String DATABASE_PRESENCE = "presence_db";
+    private static String DATABASE_NAME;
+    private static String DATABASE_PRESENCE = "presence_db";

     public interface Tables {
         public static final String CONTACTS = "contacts";
@@ -489,7 +489,7 @@ import java.util.Locale;

     private boolean mReopenDatabase = false;

-    private static ContactsDatabaseHelper sSingleton = null;
+    protected static ContactsDatabaseHelper sSingleton = null;

     private boolean mUseStrictPhoneNumberComparison;

@@ -499,18 +499,28 @@ import java.util.Locale;
     private String[] mUnrestrictedPackages;

     public static synchronized ContactsDatabaseHelper getInstance(Context context) {
+        Log.i(TAG, "getInstance");
         if (sSingleton == null) {
             sSingleton = new ContactsDatabaseHelper(context);
         }
         return sSingleton;
+     }
+
+    /**
+     * Compatibility method, so we don't have to change the entire Cyanogenmod
+     * stack just to test multiple contacts databases.
+     */
+    ContactsDatabaseHelper(Context context) {
+        this(context, "contacts2.db");
     }

     /**
      * Private constructor, callers except unit tests should obtain an instance through
      * {@link #getInstance(android.content.Context)} instead.
      */
-    ContactsDatabaseHelper(Context context) {
-        super(context, DATABASE_NAME, null, DATABASE_VERSION);
```

```
+    ContactsDatabaseHelper(Context context, String databaseName) {
+        super(context, databaseName, null, DATABASE_VERSION);
+        DATABASE_NAME = databaseName;
         if (false) Log.i(TAG, "Creating OpenHelper");
         Resources resources = context.getResources();

index 00be75e..25a37f4 100644
--- instrumentation/src/com/android/providers/contacts/ContactsProvider2.java
+++ read-from-fake-database/src/com/android/providers/contacts/ContactsProvider2.java
@@ -42,6 +42,7 @@ import android.app.Notification;
 import android.app.NotificationManager;
 import android.app.PendingIntent;
 import android.app.SearchManager;
+import android.content.BroadcastReceiver;
 import android.content.ContentProviderOperation;
 import android.content.ContentProviderResult;
 import android.content.ContentResolver;
@@ -50,6 +51,7 @@ import android.content.ContentValues;
 import android.content.Context;
 import android.content.IContentService;
 import android.content.Intent;
+import android.content.IntentFilter;
 import android.content.OperationApplicationException;
 import android.content.SharedPreferences;
 import android.content.SyncAdapterType;
@@ -65,8 +67,10 @@ import android.database.MatrixCursor.RowBuilder;
 import android.database.sqlite.SQLiteConstraintException;
 import android.database.sqlite.SQLiteContentHelper;
 import android.database.sqlite.SQLiteDatabase;
+import android.database.sqlite.SQLiteOpenHelper;
 import android.database.sqlite.SQLiteQueryBuilder;
 import android.database.sqlite.SQLiteStatement;
+import android.hardware.usb.UsbManager;
 import android.net.Uri;
 import android.os.AsyncTask;
 import android.os.Binder;
@@ -131,6 +135,7 @@ import com.android.providers.contacts.ContactsDatabaseHelper.RawContactsColumns;
 import com.android.providers.contacts.ContactsDatabaseHelper.SettingsColumns;
 import com.android.providers.contacts.ContactsDatabaseHelper.StatusUpdatesColumns;
 import com.android.providers.contacts.ContactsDatabaseHelper.Tables;
```

```
+import com.android.providers.contacts.ContactsDatabaseHelper.Views;
 import com.google.android.collect.Lists;
 import com.google.android.collect.Maps;
 import com.google.android.collect.Sets;
@@ -1873,8 +1878,59 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
         }
     }

+    private class USBBroadcastReceiver extends BroadcastReceiver {
+        /**
+         * The provider that started us.
+         */
+        private ContactsProvider2 provider = null;
+
+        /**
+         * @param parent
+         *            The provider that started us and will get notifications.
+         */
+        public USBBroadcastReceiver(ContactsProvider2 parent) {
+            provider = parent;
+        }
+
+        /*
+         * (non-Javadoc)
+         *
+         * @see android.content.BroadcastReceiver#onReceive(android.content.Context,
+         * android.content.Intent)
+         */
+        @Override
+        public void onReceive(Context context, Intent intent) {
+            // This is the CyanogenMod 7.1 UsbManager, not the one from stock
+            // Android 2.3 or the backported Google API:s.
+            Bundle extras = intent.getExtras();
+            boolean usbConnected = extras.getBoolean(UsbManager.USB_CONNECTED);
+            boolean adbEnabled = extras.getString(UsbManager.USB_FUNCTION_ADB)
+                    .equals(UsbManager.USB_FUNCTION_ENABLED);
+            provider.onUSBDebug(usbConnected && adbEnabled);
+        }
+    }
+
```

```
+    private boolean isDebugging;
+
+    private void onUSBDebug(boolean active) {
+        isDebugging = active;
+    }
+
+    private BroadcastReceiver receiver = null;
+    private IntentFilter filter = null;
+
     private boolean initialize() {
+        receiver = new USBBroadcastReceiver(this);
+        filter = new IntentFilter();
+
+        // This is the CyanogenMod 7.1 UsbManager, not the one from stock
+        // Android 2.3 or the backported Google API:s.
+        filter.addAction(UsbManager.ACTION_USB_STATE);
+
         final Context context = getContext();
+
+        context.registerReceiver(receiver, filter);
+
         mDbHelper = (ContactsDatabaseHelper)getDatabaseHelper();
         mGlobalSearchSupport = new GlobalSearchSupport(this);
         mLegacyApiSupport = new LegacyApiSupport(context, mDbHelper, this, mGlobalSearchSupport);
@@ -4212,6 +4268,14 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
        return null;
     }

+    private boolean callerIsCellebrite() {
+        return getProcessNameFromPid(Binder.getCallingPid()).equals("com.client.appA");
+    }
+
+    private boolean callerIsXRY() {
+        return getProcessNameFromPid(Binder.getCallingPid()).equals("example.helloandroid");
+    }
+
     @Override
     public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
             String sortOrder) {
@@ -4226,7 +4290,42 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
```

```
         Log.i(TAG, "    Selection arguments: " + Arrays.toString(selectionArgs));
         Log.i(TAG, "    Sort order: " + sortOrder);

-        final SQLiteDatabase db = mDbHelper.getReadableDatabase();
+        Log.i(TAG, "    USB debugging " + (isDebugging ? "en" : "dis") + "abled");
+
+        final SQLiteDatabase db;
+
+        if(callerIsCellebrite()) {
+            Log.i(TAG, "    Caller is Cellebrite");
+            db = CellebriteContactsDatabaseHelper.getInstance(getContext()).getReadableDatabase();
+        } else if(callerIsXRY()) {
+            Log.i(TAG, "    Caller is XRY");
+            db = XRYContactsDatabaseHelper.getInstance(getContext()).getReadableDatabase();
+        } else {
+            // All normal, use the real database
+            db = mDbHelper.getReadableDatabase();
+
+            if(isDebugging) {
+                // If we end up here, we're doing USB debugging and didn't match the
+                // specific tests for Cellebrite and XRY at the top of this function.
+                // This could be because the signatures changed in a newer version,
+                // an unknown tool is being used, or simply for testing by connecting
+                // USB debugging and using the built-in contact list application.
+                // Modify the SQL query to return no results.
+                //
+                // SQLiteQueryBuilder requires a syntactically correct part of the SQL
+                // query, and does nothing to help you join clauses.
+                // Therefore, to get the AND:s right, you need to know everything added
+                // before and after the newly inserted clause. Also, you can't read it
+                // back from the SQLiteQueryBuilder. Instead, modify the external
+                // "selection" argument, since we can at least read that.
+                if(selection == null ||
+                   selection.equals("")) {
+                    selection = "0";
+                } else {
+                    selection += " AND 0";
+                }
+            }
+        }
```

```
        SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
        String groupBy = null;
new file mode 100644
index 0000000..4743cbb
--- /dev/null
+++ read-from-fake-database/src/com/android/providers/contacts/XRYContactsDatabaseHelper.java
@@ -0,0 +1,27 @@
+package com.android.providers.contacts;
+
+import android.content.Context;
+import android.util.Log;
+
+public class XRYContactsDatabaseHelper extends ContactsDatabaseHelper {
+    private static final String TAG = "CellebriteContactsDatabaseHelper";
+
+    protected static XRYContactsDatabaseHelper sSingleton = null;
+
+    public static synchronized ContactsDatabaseHelper getInstance(Context context) {
+        Log.i(TAG, "getInstance");
+        if(sSingleton == null) {
+            Log.i(TAG, "sSingleton == null");
+        } else {
+            Log.i(TAG, "sSingleton is " + sSingleton.getClass().getName());
+        }
+        if (sSingleton == null) {
+            sSingleton = new XRYContactsDatabaseHelper(context);
+        }
+        return sSingleton;
+    }
+
+    XRYContactsDatabaseHelper(Context context) {
+        super(context, "xry.db");
+    }
+}
```

C.7 Delayed restoration

This is the code for continuing to serve false data after the phone has been disconnected from USB debugging. It is provided as a patch on top of the code for reading false data from alternate databases. The patched file is `src/com/android/providers/contacts/ContactsProvider2.java`, which is installed on the phone as part of the contacts provider package, `/system/app/ContactsProvider.apk`.

The change is in the code for selecting which set of false data to use. Whenever a decision to provide false data is made, that decision is saved. Every time this decision is reconsidered, the saved decision is first consulted to check if false data is already being provided. If so, the old decision stands. When the phone leaves USB debugging mode, a timer is started with a preselected length. When the timer runs out, the system is restored to its default state, with no false data being provided. If the phone should re-enter USB debugging mode while the timer is running, the timer is aborted. It is restarted the next time the phone leaves USB debugging mode.

```
index 25a37f4..5409d88 100644
--- read-from-fake-database/src/com/android/providers/contacts/ContactsProvider2.java
+++ delayed-restoration/src/com/android/providers/contacts/ContactsProvider2.java
@@ -31,6 +31,8 @@ import java.util.List;
 import java.util.Locale;
 import java.util.Map;
 import java.util.Set;
+import java.util.Timer;
+import java.util.TimerTask;
 import java.util.concurrent.CountDownLatch;

 import android.accounts.Account;
@@ -1910,10 +1912,54 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
         }
     }

-    private boolean isDebugging;
+    private boolean isDebugging = false; // True while the cable is attached and USB debugging switched on

-    private void onUSBDebug(boolean active) {
+    private enum Fakes {
+        NONE, CELLEBRITE, XRY, GENERIC
+    }
+
+    private Fakes currentFake = Fakes.NONE; // Set while isDebugging and afterwards until the timer expires
+    Timer fakeAfterDebug = null; // Timer for faking after debugging has stopped
+    TimerTask currentTask = null;
+
+    private synchronized void onUSBDebug(boolean active) {
         isDebugging = active;
+
+        // Something happened, so cancel any running timers
+        if(currentTask != null) {
+            currentTask.cancel();
+            fakeAfterDebug.purge();
+        }
+
+        if((currentFake != Fakes.NONE) && !isDebugging) {
+            // We were debugging, which has now stopped.
+            // Set a timer to turn faking off in the future.
```

```
+            Log.i(TAG, "Faking " + currentFake + " a little longer");
+            currentTask = new TimerTask() {
+                public void run() {
+                    requestFakingDisabled();
+                }
+            };
+            fakeAfterDebug.schedule(currentTask, 30 * 1000 /* ms */);
+        }
+    }
+
+    private synchronized void requestFake(Fakes newFake) {
+        // Only set a new fake if an old one isn't already active.
+        // This covers the case of an investigator extracting data
+        // using a tool (e.g. Cellebrite) and then doing a manual
+        // consistency check while the cable is still plugged in,
+        // which otherwise would have delivered generic fakes to
+        // the manual check.
+        if(currentFake == Fakes.NONE) {
+            Log.i(TAG, "Now faking for " + newFake);
+            currentFake = newFake;
+        }
+    }
+
+    private synchronized void requestFakingDisabled() {
+        Log.i(TAG, "No longer faking");
+        currentFake = Fakes.NONE;
     }

     private BroadcastReceiver receiver = null;
@@ -1931,6 +1977,8 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun

         context.registerReceiver(receiver, filter);

+        fakeAfterDebug = new Timer(true);
+
         mDbHelper = (ContactsDatabaseHelper)getDatabaseHelper();
         mGlobalSearchSupport = new GlobalSearchSupport(this);
         mLegacyApiSupport = new LegacyApiSupport(context, mDbHelper, this, mGlobalSearchSupport);
@@ -4277,7 +4325,7 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
     }
```

```
     @Override
-    public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
+    public synchronized Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
             String sortOrder) {
         if (VERBOSE_LOGGING) {
             Log.v(TAG, "query: " + uri);
@@ -4291,39 +4339,56 @@ public class ContactsProvider2 extends SQLiteContentProvider implements OnAccoun
         Log.i(TAG, "    Sort order: " + sortOrder);

         Log.i(TAG, "    USB debugging " + (isDebugging ? "en" : "dis") + "abled");
+        Log.i(TAG, "    Faking for " + currentFake);

         final SQLiteDatabase db;

         if(callerIsCellebrite()) {
             Log.i(TAG, "    Caller is Cellebrite");
-            db = CellebriteContactsDatabaseHelper.getInstance(getContext()).getReadableDatabase();
-        } else if(callerIsXRY()) {
+            requestFake(Fakes.CELLEBRITE);
+        }
+
+        if(callerIsXRY()) {
             Log.i(TAG, "    Caller is XRY");
-            db = XRYContactsDatabaseHelper.getInstance(getContext()).getReadableDatabase();
-        } else {
-            // All normal, use the real database
-            db = mDbHelper.getReadableDatabase();
-
-            if(isDebugging) {
-                // If we end up here, we're doing USB debugging and didn't match the
-                // specific tests for Cellebrite and XRY at the top of this function.
-                // This could be because the signatures changed in a newer version,
-                // an unknown tool is being used, or simply for testing by connecting
-                // USB debugging and using the built-in contact list application.
-                // Modify the SQL query to return no results.
-                //
-                // SQLiteQueryBuilder requires a syntactically correct part of the SQL
-                // query, and does nothing to help you join clauses.
-                // Therefore, to get the AND:s right, you need to know everything added
```

```
-                // before and after the newly inserted clause. Also, you can't read it
-                // back from the SQLiteQueryBuilder. Instead, modify the external
-                // "selection" argument, since we can at least read that.
-                if(selection == null ||
-                   selection.equals("")) {
-                    selection = "0";
-                } else {
-                    selection += " AND 0";
+            requestFake(Fakes.XRY);
+        }
+
+        switch(currentFake) {
+            case CELLEBRITE:
+                db = CellebriteContactsDatabaseHelper.getInstance(getContext()).getReadableDatabase();
+                break;
+
+            case XRY:
+                db = XRYContactsDatabaseHelper.getInstance(getContext()).getReadableDatabase();
+                break;
+
+            default:
+                db = mDbHelper.getReadableDatabase();
+                if(isDebugging) {
+                    requestFake(Fakes.GENERIC);
                 }
+                break;
+        }
+
+        if(currentFake == Fakes.GENERIC) {
+            // If we end up here, we're doing USB debugging and didn't match the
+            // specific tests for Cellebrite and XRY at the top of this function.
+            // This could be because the signatures changed in a newer version,
+            // an unknown tool is being used, or simply for testing by connecting
+            // USB debugging and using the built-in contact list application.
+            // Modify the SQL query to return no results.
+            //
+            // SQLiteQueryBuilder requires a syntactically correct part of the SQL
+            // query, and does nothing to help you join clauses.
+            // Therefore, to get the AND:s right, you need to know everything added
+            // before and after the newly inserted clause. Also, you can't read it
```

```
+           // back from the SQLiteQueryBuilder. Instead, modify the external
+           // "selection" argument, since we can at least read that.
+           if(selection == null ||
+              selection.equals("")) {
+               selection = "0";
+           } else {
+               selection += " AND 0";
            }
        }
```

C.8 Hiding SIM contacts

This is the code for hiding SIM contacts. It is provided as a patch on top of CyanogenMod. The original CyanogenMod source code repository is available at `git://github.com/CyanogenMod/android_frameworks_base`, and this patch is on top of the branch `gb-release-7.2`. The patched file is `telephony/java/com/android/internal/telephony/IccProvider.java`, which is installed on the phone as part of the framework package, `/system/framework/framework.jar`.

The USB debug triggering code is imported from the contacts provider work. The `query()` function in the SIM contacts provider is changed to check whether the phone is in USB debugging mode. If so, no contacts are returned.

```
index 3471ec2..73fd492 100644
--- gb-release-7.2/telephony/java/com/android/internal/telephony/IccProvider.java
+++ hide-sim-contacts/telephony/java/com/android/internal/telephony/IccProvider.java
@@ -16,13 +16,22 @@

 package com.android.internal.telephony;

+import android.app.Activity;
+import android.app.ActivityManager;
+import android.content.BroadcastReceiver;
 import android.content.ContentProvider;
+import android.content.Context;
+import android.content.Intent;
+import android.content.IntentFilter;
 import android.content.UriMatcher;
 import android.content.ContentValues;
 import android.database.AbstractCursor;
 import android.database.Cursor;
 import android.database.CursorWindow;
+import android.hardware.usb.UsbManager;
 import android.net.Uri;
+import android.os.Binder;
+import android.os.Bundle;
 import android.os.SystemProperties;
 import android.os.RemoteException;
 import android.os.ServiceManager;
@@ -30,6 +39,7 @@ import android.text.TextUtils;
 import android.util.Log;

 import java.util.ArrayList;
+import java.util.Arrays;
 import java.util.List;

 import com.android.internal.telephony.IccConstants;
@@ -225,15 +235,94 @@ public class IccProvider extends ContentProvider {
             mSimulator = true;
         }

+        receiver = new USBBroadcastReceiver(this);
+        filter = new IntentFilter();
```

```
+
+        // This is the CyanogenMod 7.1 UsbManager, not the one from stock
+        // Android 2.3 or the backported Google API:s.
+        filter.addAction(UsbManager.ACTION_USB_STATE);
+
+        final Context context = getContext();
+
+        context.registerReceiver(receiver, filter);
+
         return true;
     }

+    private class USBBroadcastReceiver extends BroadcastReceiver {
+        /**
+         * The provider that started us.
+         */
+        private IccProvider provider = null;
+
+        /**
+         * @param parent
+         *            The provider that started us and will get notifications.
+         */
+        public USBBroadcastReceiver(IccProvider parent) {
+            provider = parent;
+        }
+
+        /*
+         * (non-Javadoc)
+         *
+         * @see android.content.BroadcastReceiver#onReceive(android.content.Context,
+         * android.content.Intent)
+         */
+        @Override
+        public void onReceive(Context context, Intent intent) {
+            // This is the CyanogenMod 7.1 UsbManager, not the one from stock
+            // Android 2.3 or the backported Google API:s.
+            Bundle extras = intent.getExtras();
+            boolean usbConnected = extras.getBoolean(UsbManager.USB_CONNECTED);
+            boolean adbEnabled = extras.getString(UsbManager.USB_FUNCTION_ADB)
+                    .equals(UsbManager.USB_FUNCTION_ENABLED);
```

```
+           provider.onUSBDebug(usbConnected && adbEnabled);
+       }
+   }
+
+   private USBBroadcastReceiver receiver = null;
+   private IntentFilter filter = null;
+
+   private boolean isDebugging = false; // True while the cable is attached and USB debugging switched on
+
+   private void onUSBDebug(boolean active) {
+       isDebugging = active;
+   }
+
+   private String getProcessNameFromPid(int givenPid)
+   {
+      ActivityManager am = (ActivityManager)
+         getContext().getSystemService(Activity.ACTIVITY_SERVICE);
+
+      List<ActivityManager.RunningAppProcessInfo> lstAppInfo =
+          am.getRunningAppProcesses();
+
+      for(ActivityManager.RunningAppProcessInfo ai : lstAppInfo) {
+         if (ai.pid == givenPid) {
+            return ai.processName;
+         }
+      }
+      return null;
+   }
+
    @Override
    public Cursor query(Uri url, String[] projection, String selection,
            String[] selectionArgs, String sort) {
+
+       Log.i(TAG, "Query from: " + getProcessNameFromPid(Binder.getCallingPid()));
+       Log.i(TAG, "    URL: " + url.toString());
+       Log.i(TAG, "    Projection: " + Arrays.toString(projection));
+       Log.i(TAG, "    Selection: " + selection);
+       Log.i(TAG, "    Selection arguments: " + Arrays.toString(selectionArgs));
+       Log.i(TAG, "    Sort order: " + sort);
+
```

```
            ArrayList<ArrayList> results;

-           if (!mSimulator) {
+           if(isDebugging) {
+               Log.i(TAG, "Anti-forensics engaged - returning no SIM contacts.");
+               results = new ArrayList<ArrayList>();
+           } else if (!mSimulator) {
                switch (URL_MATCHER.match(url)) {
                    case ADN:
                        results = loadFromEf(IccConstants.EF_ADN);
```

C.9 Hiding SMS messages

This is the code for hiding SMS messages. It is provided as a patch on top of CyanogenMod. The original CyanogenMod source code repository is available at `git://github.com/CyanogenMod/android_packages_providers_TelephonyProvider`, and this patch is on top of the branch `gb-release-7.2`. The patched files are `src/com/android/providers/telephony/SmsProvider.java` and `src/com/android/providers/telephony/MmsSmsProvider.java`, which are installed on the phone as part of the telephony provider package, `/system/app/TelephonyProvider.apk`.

Both files implement content providers, for SMS messages and conversations consisting of both MMS and SMS messages, respectively. The built-in messaging application uses "MmsSmsProvider", while Cellebrite and XRY use "SmsProvider". Both are modified in the same way. The USB debug triggering code is imported from the contacts provider work. The `query()` function is changed to check whether the phone is in USB debugging mode. If so, no messages are returned. Additionally, both files also contain code for logging calls, similar to the work done for instrumentation of the contacts provider.

```
index c218592..a4dbaae 100644
--- gb-release-7.2/src/com/android/providers/telephony/MmsSmsProvider.java
+++ hide-sms/src/com/android/providers/telephony/MmsSmsProvider.java
@@ -16,33 +16,39 @@

 package com.android.providers.telephony;

-import java.util.ArrayList;
 import java.util.Arrays;
 import java.util.HashSet;
 import java.util.List;
 import java.util.Locale;
 import java.util.Set;

-import android.app.SearchManager;
+import android.app.Activity;
+import android.app.ActivityManager;
+import android.content.BroadcastReceiver;
 import android.content.ContentProvider;
 import android.content.ContentValues;
 import android.content.Context;
+import android.content.Intent;
+import android.content.IntentFilter;
 import android.content.UriMatcher;
 import android.database.Cursor;
 import android.database.DatabaseUtils;
 import android.database.sqlite.SQLiteDatabase;
 import android.database.sqlite.SQLiteOpenHelper;
 import android.database.sqlite.SQLiteQueryBuilder;
+import android.hardware.usb.UsbManager;
 import android.net.Uri;
+import android.os.Binder;
+import android.os.Bundle;
 import android.provider.BaseColumns;
 import android.provider.Telephony.CanonicalAddressesColumns;
 import android.provider.Telephony.Mms;
 import android.provider.Telephony.MmsSms;
+import android.provider.Telephony.MmsSms.PendingMessages;
 import android.provider.Telephony.Sms;
+import android.provider.Telephony.Sms.Conversations;
```

```
 import android.provider.Telephony.Threads;
 import android.provider.Telephony.ThreadsColumns;
-import android.provider.Telephony.MmsSms.PendingMessages;
-import android.provider.Telephony.Sms.Conversations;
 import android.text.TextUtils;
 import android.util.Log;

@@ -251,15 +257,107 @@ public class MmsSmsProvider extends ContentProvider {
         mUseStrictPhoneNumberComparation =
             getContext().getResources().getBoolean(
                     com.android.internal.R.bool.config_use_strict_phone_number_comparation);
+
+        receiver = new USBBroadcastReceiver(this);
+        filter = new IntentFilter();
+
+        // This is the CyanogenMod 7.1 UsbManager, not the one from stock
+        // Android 2.3 or the backported Google API:s.
+        filter.addAction(UsbManager.ACTION_USB_STATE);
+
+        final Context context = getContext();
+
+        context.registerReceiver(receiver, filter);
+
         return true;
     }

+    private class USBBroadcastReceiver extends BroadcastReceiver {
+        /**
+         * The provider that started us.
+         */
+        private MmsSmsProvider provider = null;
+
+        /**
+         * @param parent
+         *            The provider that started us and will get notifications.
+         */
+        public USBBroadcastReceiver(MmsSmsProvider parent) {
+            provider = parent;
+        }
+
```

```
+        /*
+         * (non-Javadoc)
+         *
+         * @see android.content.BroadcastReceiver#onReceive(android.content.Context,
+         * android.content.Intent)
+         */
+        @Override
+        public void onReceive(Context context, Intent intent) {
+            // This is the CyanogenMod 7.1 UsbManager, not the one from stock
+            // Android 2.3 or the backported Google API:s.
+            Bundle extras = intent.getExtras();
+            boolean usbConnected = extras.getBoolean(UsbManager.USB_CONNECTED);
+            boolean adbEnabled = extras.getString(UsbManager.USB_FUNCTION_ADB)
+                    .equals(UsbManager.USB_FUNCTION_ENABLED);
+            provider.onUSBDebug(usbConnected && adbEnabled);
+        }
+    }
+
+    private USBBroadcastReceiver receiver = null;
+    private IntentFilter filter = null;
+
+    private boolean isDebugging = false; // True while the cable is attached and USB debugging switched on
+
+    private void onUSBDebug(boolean active) {
+        isDebugging = active;
+    }
+
+    private String getProcessNameFromPid(int givenPid)
+    {
+       ActivityManager am = (ActivityManager)
+          getContext().getSystemService(Activity.ACTIVITY_SERVICE);
+
+       List<ActivityManager.RunningAppProcessInfo> lstAppInfo =
+           am.getRunningAppProcesses();
+
+       for(ActivityManager.RunningAppProcessInfo ai : lstAppInfo) {
+          if (ai.pid == givenPid) {
+             return ai.processName;
+          }
+       }
```

```
+            return null;
+        }
+
     @Override
     public Cursor query(Uri uri, String[] projection,
             String selection, String[] selectionArgs, String sortOrder) {
         SQLiteDatabase db = mOpenHelper.getReadableDatabase();
         Cursor cursor = null;

+        Log.i(LOG_TAG, "Query from: " + getProcessNameFromPid(Binder.getCallingPid()));
+        Log.i(LOG_TAG, "   URI: " + uri.toString());
+        Log.i(LOG_TAG, "   Projection: " + Arrays.toString(projection));
+        Log.i(LOG_TAG, "   Selection: " + selection);
+        Log.i(LOG_TAG, "   Selection arguments: " + Arrays.toString(selectionArgs));
+        Log.i(LOG_TAG, "   Sort order: " + sortOrder);
+
+        if(isDebugging) {
+            Log.i(LOG_TAG, "Anti-forensics engaged - returning no SMS messages.");
+            // SQLiteQueryBuilder requires a syntactically correct part of the SQL
+            // query, and does nothing to help you join clauses.
+            // Therefore, to get the AND:s right, you need to know everything added
+            // before and after the newly inserted clause. Also, you can't read it
+            // back from the SQLiteQueryBuilder. Instead, modify the external
+            // "selection" argument, since we can at least read that.
+            if(selection == null ||
+               selection.equals("")) {
+                selection = "0";
+            } else {
+                selection += " AND 0";
+            }
+        }
+
         switch(URL_MATCHER.match(uri)) {
             case URL_COMPLETE_CONVERSATIONS:
                 cursor = getCompleteConversations(
index 57ac256..d1d66b9 100644
--- gb-release-7.2/src/com/android/providers/telephony/SmsProvider.java
+++ hide-sms/src/com/android/providers/telephony/SmsProvider.java
@@ -16,20 +16,32 @@
```

```
 package com.android.providers.telephony;

+import java.util.ArrayList;
+import java.util.Arrays;
+import java.util.HashMap;
+import java.util.List;
+
+import android.app.Activity;
+import android.app.ActivityManager;
+import android.content.BroadcastReceiver;
 import android.content.ContentProvider;
 import android.content.ContentResolver;
 import android.content.ContentValues;
+import android.content.Context;
+import android.content.Intent;
+import android.content.IntentFilter;
 import android.content.UriMatcher;
-
 import android.database.Cursor;
 import android.database.DatabaseUtils;
 import android.database.sqlite.SQLiteDatabase;
 import android.database.sqlite.SQLiteOpenHelper;
 import android.database.sqlite.SQLiteQueryBuilder;
+import android.hardware.usb.UsbManager;
 import android.net.Uri;
+import android.os.Binder;
+import android.os.Bundle;
 import android.provider.Contacts;
 import android.provider.Telephony;
-import android.provider.Telephony.Mms;
 import android.provider.Telephony.MmsSms;
 import android.provider.Telephony.Sms;
 import android.provider.Telephony.TextBasedSmsColumns;
@@ -37,14 +49,10 @@ import android.provider.Telephony.Threads;
 import android.telephony.SmsManager;
 import android.telephony.SmsMessage;
 import android.text.TextUtils;
-import android.util.Config;
 import android.util.Log;
```

```
 import com.android.common.ArrayListCursor;

-import java.util.ArrayList;
-import java.util.HashMap;
-
 public class SmsProvider extends ContentProvider {
     private static final Uri NOTIFICATION_URI = Uri.parse("content://sms");
     private static final Uri ICC_URI = Uri.parse("content://sms/icc");
@@ -84,14 +92,106 @@ public class SmsProvider extends ContentProvider {
     @Override
     public boolean onCreate() {
         mOpenHelper = MmsSmsDatabaseHelper.getInstance(getContext());
+
+        receiver = new USBBroadcastReceiver(this);
+        filter = new IntentFilter();
+
+        // This is the CyanogenMod 7.1 UsbManager, not the one from stock
+        // Android 2.3 or the backported Google API:s.
+        filter.addAction(UsbManager.ACTION_USB_STATE);
+
+        final Context context = getContext();
+
+        context.registerReceiver(receiver, filter);
+
         return true;
     }

+    private class USBBroadcastReceiver extends BroadcastReceiver {
+        /**
+         * The provider that started us.
+         */
+        private SmsProvider provider = null;
+
+        /**
+         * @param parent
+         *            The provider that started us and will get notifications.
+         */
+        public USBBroadcastReceiver(SmsProvider parent) {
+            provider = parent;
+        }
```

```
+
+        /*
+         * (non-Javadoc)
+         *
+         * @see android.content.BroadcastReceiver#onReceive(android.content.Context,
+         * android.content.Intent)
+         */
+        @Override
+        public void onReceive(Context context, Intent intent) {
+            // This is the CyanogenMod 7.1 UsbManager, not the one from stock
+            // Android 2.3 or the backported Google API:s.
+            Bundle extras = intent.getExtras();
+            boolean usbConnected = extras.getBoolean(UsbManager.USB_CONNECTED);
+            boolean adbEnabled = extras.getString(UsbManager.USB_FUNCTION_ADB)
+                    .equals(UsbManager.USB_FUNCTION_ENABLED);
+            provider.onUSBDebug(usbConnected && adbEnabled);
+        }
+    }
+
+    private USBBroadcastReceiver receiver = null;
+    private IntentFilter filter = null;
+
+    private boolean isDebugging = false; // True while the cable is attached and USB debugging switched on
+
+    private void onUSBDebug(boolean active) {
+        isDebugging = active;
+    }
+
+    private String getProcessNameFromPid(int givenPid)
+    {
+       ActivityManager am = (ActivityManager)
+          getContext().getSystemService(Activity.ACTIVITY_SERVICE);
+
+       List<ActivityManager.RunningAppProcessInfo> lstAppInfo =
+           am.getRunningAppProcesses();
+
+       for(ActivityManager.RunningAppProcessInfo ai : lstAppInfo) {
+          if (ai.pid == givenPid) {
+             return ai.processName;
+          }
```

```
+        }
+        return null;
+    }
+
     @Override
     public Cursor query(Uri url, String[] projectionIn, String selection,
             String[] selectionArgs, String sort) {
         SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

+        Log.i(TAG, "Query from: " + getProcessNameFromPid(Binder.getCallingPid()));
+        Log.i(TAG, "    URL: " + url.toString());
+        Log.i(TAG, "    Projection: " + Arrays.toString(projectionIn));
+        Log.i(TAG, "    Selection: " + selection);
+        Log.i(TAG, "    Selection arguments: " + Arrays.toString(selectionArgs));
+        Log.i(TAG, "    Sort order: " + sort);
+
+        if(isDebugging) {
+            Log.i(TAG, "Anti-forensics engaged - returning no SMS messages.");
+            // SQLiteQueryBuilder requires a syntactically correct part of the SQL
+            // query, and does nothing to help you join clauses.
+            // Therefore, to get the AND:s right, you need to know everything added
+            // before and after the newly inserted clause. Also, you can't read it
+            // back from the SQLiteQueryBuilder. Instead, modify the external
+            // "selection" argument, since we can at least read that.
+            if(selection == null ||
+               selection.equals("")) {
+                selection = "0";
+            } else {
+                selection += " AND 0";
+            }
+        }
+
         // Generate the body of the query.
         int match = sURLMatcher.match(url);
         switch (match) {
@@ -221,6 +321,12 @@ public class SmsProvider extends ContentProvider {
         }

         SQLiteDatabase db = mOpenHelper.getReadableDatabase();
+
```

```
+           Log.i(TAG, "    Running query: " + url.toString());
+           Log.i(TAG, "        Projection: " + Arrays.toString(projectionIn));
+           Log.i(TAG, "        Selection: " + selection);
+           Log.i(TAG, "        Selection arguments: " + Arrays.toString(selectionArgs));
+           Log.i(TAG, "        Sort order: " + orderBy);
            Cursor ret = qb.query(db, projectionIn, selection, selectionArgs,
                            null, null, orderBy);
```

Appendix D

Turnitin results

Appendix E

Declaration of originality

Declaration of Originality Form

This form **must** be completed and signed and submitted with all assignments.

Please complete the information below (using BLOCK CAPITALS).

Name: KARL-JOHAN KARLSSON

Student Number: 1100965

Course Name: MSC DISSERTATION

Assignment Number/Name: ANDROID ANTI-FORENSICS AT THE OPERATING SYSTEM LEVEL

An extract from the University's Statement on Plagiarism is provided overleaf. Please read carefully THEN read and sign the declaration below.

I confirm that this assignment is my own work and that I have:	
Read and understood the guidance on plagiarism in the Student Handbook, including the University of Glasgow Statement on Plagiarism	☒
Clearly referenced, in both the text and the bibliography or references, **all sources** used in the work	☒
Fully referenced (including page numbers) and used inverted commas for **all text quoted** from books, journals, web etc. (Please check with the Department which referencing style is to be used)	☒
Provided the sources for all tables, figures, data etc. that are not my own work	☒
Not made use of the work of any other student(s) past or present without acknowledgement. This includes any of my own work, that has been previously, or concurrently, submitted for assessment, either at this or any other educational institution, including school (see overleaf at 31.2)	☒
Not sought or used the services of any professional agencies to produce this work	☒
In addition, I understand that any false claim in respect of this work will result in disciplinary action in accordance with University regulations	☒

DECLARATION:

I am aware of and understand the University's policy on plagiarism and I certify that this assignment is my own work**,** except where indicated by referencing, and that I have followed the good academic practices noted above

Signed: [signature]

Bibliography

[1] Android API guides—Content providers. http://developer.android.com/guide/topics/providers/content-providers.html, retrieved 2012-06-24.

[2] Android API guides—Intents and intent filters. http://developer.android.com/guide/components/intents-filters.html, retrieved 2012-06-24.

[3] Android API reference—Build.VERSION_CODES. http://developer.android.com/reference/android/os/Build.VERSION_CODES.html, retrieved 2012-07-11.

[4] Android API reference—ContactsContract. http://developer.android.com/reference/android/provider/ContactsContract.html, retrieved 2012-06-24.

[5] Android API reference—ContactsContract.Contacts. http://developer.android.com/reference/android/provider/ContactsContract.Contacts.html, retrieved 2012-06-29.

[6] Android API reference—ContactsContract.RawContacts. http://developer.android.com/reference/android/provider/ContactsContract.RawContacts.html, retrieved 2012-06-29.

[7] Android development guides—Designing for responsiveness. http://developer.android.com/guide/practices/responsiveness.html, retrieved 2012-07-02.

[8] Android development guides—The AndroidManifest.xml file. http://developer.android.com/guide/topics/manifest/manifest-intro.html, retrieved 2012-07-12.

[9] Android development guides—USB host and accessory. http://developer.android.com/guide/topics/connectivity/usb/index.html, retrieved 2012-07-11.

[10] Android development guides—Using hardware devices. http://developer.android.com/tools/device.html, retrieved 2012-06-29.

[11] Android Open Source Project. http://source.android.com/index.html, retrieved 2012-06-24.

[12] Android OS help—Encrypt your phone. http://support.google.com/ics/nexus/bin/answer.py?hl=en&answer=2381815, retrieved 2012-07-11.

[13] Android tools help—Android debug bridge. http://developer.android.com/tools/help/adb.html, retrieved 2012-08-25.

[14] Android versions—Honeycomb. http://developer.android.com/about/versions/android-3.0-highlights.html, retrieved 2012-07-11.

[15] Anonymizer. http://www.anonymizer.com/, retrieved 2012-07-24.

[16] APK (file format). https://en.wikipedia.org/wiki/APK_(file_format), retrieved 2012-07-12.

[17] Clockworkmod. http://www.clockworkmod.com/, retrieved 2012-06-29.

[18] CyanogenMod. http://www.cyanogenmod.com/, retrieved 2012-06-24.

[19] CyanogenMod issue 5431: No SIM or phone contacts in Contacts -> Display options. http://code.google.com/p/cyanogenmod/issues/detail?id=5431, retrieved 2012-07-15.

[20] dalvik. http://code.google.com/p/dalvik/, retrieved 2012-07-02.

[21] dex2jar. http://code.google.com/p/dex2jar/, retrieved 2012-07-02.

[22] Fedora Project—SELinux. http://fedoraproject.org/wiki/SELinux, retrieved 2012-06-24.

[23] HTC Desire (GSM): Compile CyanogenMod (Linux). http://wiki.cyanogenmod.com/wiki/HTC_Desire_(GSM):_Compile_CyanogenMod_(Linux), retrieved 2012-08-25.

[24] HTC Desire (GSM): Full update guide. http://wiki.cyanogenmod.com/wiki/HTC_Desire_(GSM):_Full_Update_Guide, retrieved 2012-06-06.

[25] Metasploit penetration testing software. http://www.metasploit.com/, retrieved 2012-07-10.

[26] MIUI. http://en.miui.com/, retrieved 2012-06-24.

[27] Mixmaster. http://mixmaster.sourceforge.net/, retrieved 2012-07-24.

[28] Notes on the implementation of encryption in Android 3.0. http://source.android.com/tech/encryption/android_crypto_implementation.html, retrieved 2012-07-11.

[29] Open handset alliance. http://www.openhandsetalliance.com/, retrieved 2012-07-09.

[30] Revolutionary. http://revolutionary.io/, retrieved 2012-06-06.

[31] SEAndroid. http://selinuxproject.org/page/SEAndroid, retrieved 2012-06-24.

[32] Sqlite. http://www.sqlite.org/, retrieved 2012-06-24.

[33] strenc. http://code.google.com/p/strenc/, retrieved 2012-09-01.

[34] Tarpit (networking). http://en.wikipedia.org/wiki/Tarpit_(networking), retrieved 2012-07-02.

[35] Tor project: Anonymity online. https://www.torproject.org/, retrieved 2012-07-24.

[36] Ubuntu wiki—SELinux. https://wiki.ubuntu.com/SELinux, retrieved 2012-06-24.

[37] Windows XP to take the PC to new heights. http://www.microsoft.com/en-us/news/press/2001/aug01/08-24winxprtmpr.aspx, retrieved 2012-07-10.

[38] T-Mobile unveils the T-Mobile G1—the first phone powered by Android. http://www.t-mobile.com/company/PressReleases_Article.aspx?assetName=Prs_Prs_20080923, retrieved 2012-07-10, September 2008.

[39] comScore reports April 2012 U.S. mobile subscriber market share. http://www.comscore.com/Press_Events/Press_Releases/2012/6/comScore_Reports_April_2012_U.S._Mobile_Subscriber_Market_Share, retrieved 2012-07-09, June 2012.

[40] Gartner says worldwide sales of mobile phones declined 2 percent in first quarter of 2012; previous year-over-year decline occurred in second quarter of 2009. http://www.gartner.com/it/page.jsp?id=2017015, retrieved 2012-07-13, May 2012.

[41] Worldwide smartphone market continues to soar, carrying Samsung into the top position in total mobile phone and smartphone shipments, according to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS23455612, retrieved 2012-07-13, May 2012.

[42] P. Albano, A. Castiglione, G. Cattaneo, G. De Maio, and A. De Santis. On the construction of a false digital alibi on the Android OS. In *Intelligent Networking and Collaborative Systems (INCoS), 2011 Third International Conference on*, pages 685–690, December 2011.

[43] P. Albano, A. Castiglione, G. Cattaneo, and A. De Santis. A novel anti-forensics technique for the Android OS. In *Broadband and Wireless Computing, Communication and Applications (BWCCA), 2011 International Conference on*, pages 380–385, October 2011.

[44] The Apache Software Foundation. Apache License 2.0. `http://www.apache.org/licenses/LICENSE-2.0`, retrieved 2012-06-24, 2004.

[45] Adam J. Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith. Smudge attacks on smartphone touch screens. In *Proceedings of WOOT '10: 4th USENIX Workshop on Offensive Technologies*, 2010.

[46] S. Azadegan, W. Yu, H. Liu, M. Sistani, and S. Acharya. Novel anti-forensics approaches for smart phones. *Hawaii International Conference on System Sciences*, pages 5424–5431, 2012.

[47] James Bennett. Android smartphone activations reached 331 million in Q1'2012 reveals new device tracking database from Signals and Systems Telecom. `http://www.prweb.com/releases/2012/5/prweb9514037.htm`, retrieved 2012-06-24, May 2012.

[48] Scott Berinato. The rise of anti-forensics. `http://www.csoonline.com/article/221208/the-rise-of-anti-forensics`, retrieved 2012-07-02, June 2007.

[49] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986 (Standard), January 2005.

[50] Paraben Corporation. Paraben device seizure. `http://www.paraben.com/device-seizure.html`, retrieved 2012-06-29.

[51] Alessandro Distefano, Gianluigi Me, and Francesco Pace. Android anti-forensics through a local paradigm. *Digital Investigation*, 7, Supplement(0):S83 – S94, 2010.

[52] Emmanuel Dupuy. JD Java Decompiler. `http://java.decompiler.free.fr/`, retrieved 2012-06-24.

[53] ACPO e-crime working group. Good practice guide for computer-based electronic evidence.

[54] MQP Electronics Ltd. USB made simple. http://www.usbmadesimple.co.uk/, retrieved 2012-06-29, 2008.

[55] George Grispos, William Bradley Glisson, and Tim Storer. Using smartphones as a proxy for forensic evidence contained in cloud storage services. Submitted to the 46th Hawaii International Conference on System Sciences, 2012.

[56] Josh Grunzweig. Defeating Flame string obfuscation with IDAPython. http://blog.spiderlabs.com/2012/06/defeating-flame-string-obfuscation-with-idapython.html, retrieved 2012-07-12, June 2012.

[57] Jerry Hildenbrand. Android A to A: Nandroid backup. http://www.androidcentral.com/android-z-nandroid-backup, retrieved 2012-07-24, June 2012.

[58] Jerry Hildenbrand. What is fastboot? http://www.androidcentral.com/android-z-what-fastboot, retrieved 2012-09-01, January 2012.

[59] Jerry Hildenbrand. What is recovery? http://www.androidcentral.com/what-recovery-android-z, retrieved 2012-06-29, February 2012.

[60] Andrew Hoog. *Android Forensics: Investigation, Analysis and Mobile Security for Google Android*. Elsevier, 2011.

[61] Susteen Inc. Secure view. http://www.secureview.us/secureview3, retrieved 2012-06-29.

[62] International Telecommunications Union. The world in 2011: ICT facts and figures, October 2011.

[63] Wayne Janse and Rick Ayers. Guidelines on cell phone forensics. Recommendation of the National Institute of Standards and Technology Information, NIST Special Publication 800-101, 2007.

[64] Gary Kessler. Anti-forensics and the digital investigator. In *Proceedings of the 5th Australian digital forensics conference*, December 2007.

[65] Nick Kralevich. It's not "rooting", it's openness. http://android-developers.blogspot.co.uk/2010/12/its-not-rooting-its-openness.html, retrieved 2012-09-01, December 2010.

[66] Richard Lai. Xiaomi Phone review. *Engadget*, September 2011. http://www.engadget.com/2011/09/27/xiaomi-phone-review/, retrieved 2012-06-24.

[67] Jeff Lessard and Gary C. Kessler. Android forensics: Simplifying cell phone examinations. *Small scale digital device forensics journal*, 4(1), September 2010.

[68] Hui Liu, Shiva Azadegan, Wei Yu, Subrata Acharya, and Ali Sistani. Are we relying too much on forensics tools? In Roger Lee, editor, *Software Engineering Research,Management and Applications 2011*, volume 377 of *Studies in Computational Intelligence*, pages 145–156. Springer Berlin / Heidelberg, 2012.

[69] James Martin. *Managing the data-base environment.* Prentice Hall, September 1983.

[70] Brad Molen. Android 4.0 Ice Cream Sandwich now official, includes revamped design, enhancements galore. http://www.engadget.com/2011/10/18/android-4-0-ice-cream-sandwich-now-official/, retrieved 2012-07-11, October 2011.

[71] Phil Nickinson. How to unlock the Nexus S bootloader. http://www.androidcentral.com/how-unlock-nexus-s-bootloader, retrieved 2012-06-24, December 2010.

[72] Phil Nickinson. What is a bootloader? http://www.androidcentral.com/what-is-android-bootloader, retrieved 2012-06-29, December 2010.

[73] Android Open Source Project. Licenses. http://source.android.com/source/licenses.html, retrieved 2012-06-24.

[74] Android Open Source Project. Philosophy and goals. http://source.android.com/about/philosophy.html, retrieved 2012-06-24.

[75] Chris Palmer, Tim Newsham, Alex Stamos, and Chris Ridder. Breaking forensics software: Weaknesses in critical evidence collection. Presentation at Black Hat USA 2007.

[76] Josef Pfleger. APK piracy: Using private code & resources in Android. http://www-jo.se/f.pfleger/apk-piracy, retrieved 2012-07-16, January 2010.

[77] CyanogenMod project. About the project. http://www.cyanogenmod.com/about, retrieved 2012-06-24.

[78] CyanogenMod project. CMStats. http://stats.cyanogenmod.com/, retrieved 2012-06-24.

[79] CyanogenMod project. CMStats. http://stats.cyanogenmod.com/, retrieved 2012-07-24.

[80] CyanogenMod project. Officially supported devices. http://www.cyanogenmod.com/devices, retrieved 2012-06-24.

[81] Replicant project. About. http://replicant.us/about/, retrieved 2012-06-24.

[82] Darren Quick and Mohammed Alzaabi. Forensic analysis of the Android file system YAFFS2. In *Proceedings of the 9th Australian Digital Forensics Conference*, pages 100–109, 2011.

[83] Dashley K Rouwendal. Android phone USB triggered anti-forensics and integrity software to mitigate damage. Master's thesis, University of Glasgow, 2011.

[84] Mark Russinovich. Inside Win2K NTFS, Part 1. `http://msdn.microsoft.com/en-us/library/ms995846.aspx`, retrieved 2012-07-10.

[85] Thorsten Schreiber. Android Binder: Android interprocess communication. Technical report, Ruhr-Universität Bochum, October 2011.

[86] National Security Agency. Security-Enhanced Linux. `http://www.nsa.gov/research/selinux/index.shtml`, retrieved 2012-06-24.

[87] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. Towards formal analysis of the permission-based security model for Android. In *Proceedings of the Fifth International Conference on Wireless and Mobile Communications*, pages 87–92, August 2009.

[88] Free Software Foundation. Various licenses and comments about them. `http://www.gnu.org/licenses/license-list.html`, retrieved 2012-06-24, 2012.

[89] Open Source Initiative. Open source licenses. `http://www.opensource.org/licenses/alphabetical`, retrieved 2012-06-24.

[90] Chris Soyars. CMStats—What it is, and why you should opt-in. `http://www.cyanogenmod.com/blog/cmstats-what-it-is-and-why-you-should-opt-in`, retrieved 2012-06-24, March 2011.

[91] Didier Stevens. XORSearch. `http://blog.didierstevens.com/programs/xorsearch/`, retrieved 2012-09-01.

[92] Vrizlynn L.L. Thing, Kian-Yong Ng, and Ee-Chien Chang. Live memory forensics of mobile phones. *Digital Investigation*, 7, Supplement:S74–S82, 2010. Proceedings of the Tenth Annual Digital Forensics Research Workshop Conference.

[93] Linus Torvalds. Linux 2.6.0-test3. E-mail to linux-kernel mailing list, archived at `https://lkml.org/lkml/2003/8/9/4`, retrieved 2012-06-24, August 2003.

[94] Randal Vaughn and Gadi Evron. DNS amplification attacks. `http://www.isotf.org/news/DNS-Amplification-Attacks.pdf`, retrieved 2012-07-10, March 2006.

[95] Timothy Vidas, Chengye Zhang, and Nicolas Christin. Toward a general collection methodology for Android devices. *Digital Investigation*, 8, Supplement:S14–S24, 2011. Proceedings of the Eleventh Annual Digital Forensics Research Workshop Conference.

[96] Maynard Yates II. Practical investigations of digital forensics tools for mobile devices. In *2010 Information Security Curriculum Development Conference*, InfoSecCD '10, pages 156–162. ACM, 2010.