Digital Forensics and File Carving on the Android Platform

Andrew Folloder

Abstract

In this paper I will cover the prevalence of smartphones in society (especially those running the Android Operating System) and the importance of being able to perform forensics on them. Some general framework of the Android platform will be discussed as well as the digital forensics process, data extraction, and file carving. The techniques used and their results will then be presented, and then some future improvements and work will conclude the paper.

Table of Contents

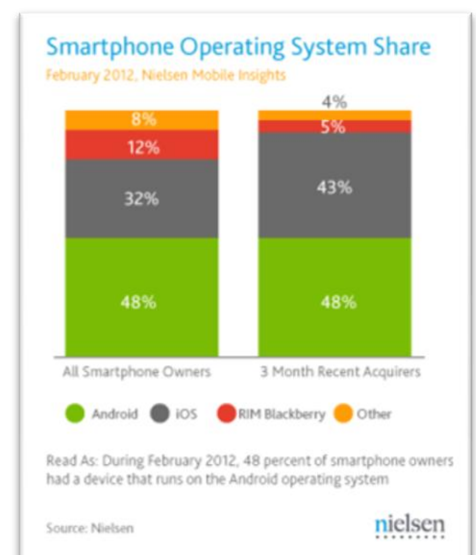
1) Introduction
2) Android Background
- Partitions
- Filesystem
- Boot Process
- Bootloader

3) Data Acquisition
- Android Debug Bridge
- Recovery Partition
- Custom Recovery ROM

4) Data Analysis
- Simple Methods
- File Carving

5) Results
6) Future Work

Introduction



The smartphone's presence within the past few years has exploded and has quickly become integrated into everyday life. And with the capability of smartphones continually growing, the amount and sensitivity of information these devices store also increases. With common features like GPS, Cellular, Wi-Fi, Bluetooth, camera, video recording, email, NFC, and more it is crucial for law enforcement and the private sector to be able to perform forensics because data such as this can be invaluable in an investigation. The Android Operating System was chosen because it is open-source, easily available, widely supported, based off of the Linux

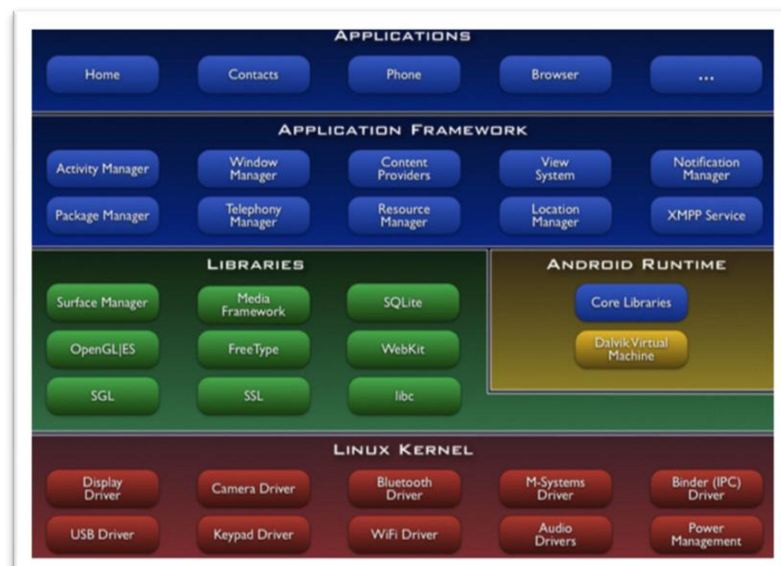
kernel, and the most popular, surpassing the very popular iOS used by all Apple mobile devices.

When performing digital forensics on any device there are always special procedures and precautions that must be taken to ensure the integrity of an investigation. The most important rule is to never modify the original data on the device. This is extremely important because you want to preserve the state of the device and its data as it is from when it was first obtained. Usually this is done by using write-blocker hardware whenever the device is accessed so that modification of the device is physically impossible. However, smartphones pose an interesting problem because their internal disk is not removable like traditional computers, and is usually a flash memory chip soldered onto the circuit board. The next important rule to follow is to always obtain a bit-by-bit copy of the target data. Obtaining a copy of the data this way ensures that no information is lost in transfer, including "free space" on the disk. Lastly, documentation is key when performing any part of forensics particularly when extracted data is accessed or modified in any way. These rules make sure that digital forensic investigations are executed in a way that will hold under scrutiny and are true in their results.

One of the key techniques that is used in data analysis is file carving. File carving is the process of reassembling data file fragments by using known file structures, heuristics, and other available information. This process allows investigators to recover data that was "deleted" from a device or from just pieces of data recovered. This is possible because when files are deleted from a device the physical bits of information are not immediately erased/overwritten but only the record of them is deleted. With newer disk technology such as flash devices this is not strictly true and will be discussed later.

Android Background

As stated before, AndroidOS runs off the Linux kernel which is open-source and widely studied, allowing us to be able to utilize preexisting knowledge and utilities. AndroidOS is also open-source itself which allows us to download its source code and analyze its operation down to the lowest level so that we are able to fully understand its operation. Here, I will go over a few basic aspects of the Android platform that are particularly important to my research.



Partitions

Like in traditional computers, the disk space on an Android device can contain be divided into multiple partitions for better organization. Typically, any Android device will contain six important partitions:

```
dev: size erasesize name
mtd0: 000a0000 00020000 "misc"
mtd1: 00480000 00020000 "recovery"
mtd2: 00300000 00020000 "boot"
mtd3: 0f800000 00020000 "system"
mtd4: 000a0000 00020000 "local"
mtd5: 02800000 00020000 "cache"
mtd6: 09500000 00020000 "data"
```

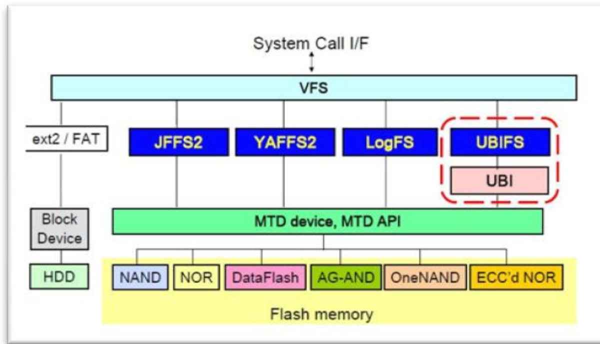
1. misc: Contains miscellaneous system settings such as carrier or region id, usb configuration, and certain hardware settings. The device will not operate properly without this partition.
2. recovery: A very important partition that acts as an alternate boot partition that can be used to performa advanced recovery or maintainence operations. (We will discuss this partition much more later).
3. boot: This enables the phone to boot, and includes the kernel and ramdisk. If this partition is corrupted or missing the phone will not be able to boot at all.
4. system: The entire operating system is contained here (other than the kernel and ramdisk), including the Android interface and the preinstalled applications that come on the phone.
5. cache: This partition (as the name suggest) is used by the OS to cache various information such as frequently accessed data and application components.
6. data: User's data such as contacts, messages, settings, and user installed application are contained in this partition. Erasing this partition essentially restores the device back to its initial factory state, making the device boot how it did the first time a user powered it on.

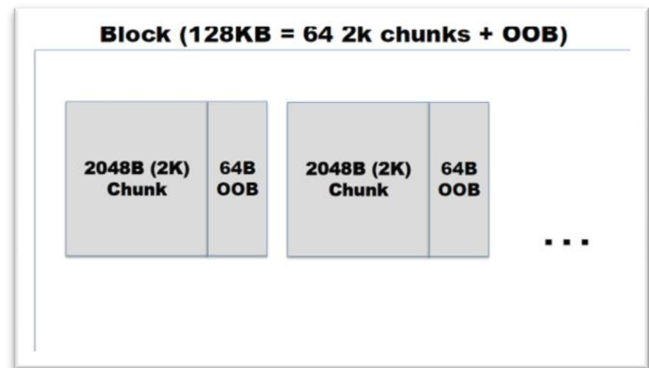
Filesystem

The Filesystem that Android uses is called Yet Another Flash Filesystem (YAFFS2). The reason why a conventional filesystem like ext or FAT isn't used has to do with the underlying flash memory used in these devices. Traditionally, the Linux kernel only recognized character devices like the mouse and keyboard where data is read from in real-time, and block devices like hard disks which have a fixed sized and can be "seeked". The problem is that flash memory doesn't fit either of those categories but is in its own individual group named Memory Technology Devices (MTD), though it is similar to block devices in some ways.

Block device	MTD device
Consists of sectors	Consists of eraseblocks
Sectors are small (512, 1024 bytes)	Eraseblocks are larger (typically 128KiB)
Maintains 2 main operations: read sector and write sector	Maintains 3 main operations: read from eraseblock, write to eraseblock, and erase eraseblock
Bad sectors are re-mapped and hidden by hardware (at least in modern LBA hard drives); in case of FTL devices it is the responsibility of FTL to provide this	Bad eraseblocks are not hidden and should be dealt with in software
Sectors are devoid of the wear-out property (in FTL devices it is the responsibility of FTL to provide this)	Eraseblocks wear-out and become bad and unusable after about 10^3 (for MLC NAND) - 10^5 (NOR, SLC NAND) erase cycles

In "traditional" flash devices like usb thumb drives and memory sticks (compact flash, SD, etc.) a microprocessor is present on the device along with the MTD and is dedicated in emulating a block device to the host computer that it's connected to by use of a simple software layer called the Flash Translation Layer (FTL). However, Android devices avoid this step by communicating directly with the MTD device instead of simply treating it as a block device and relying on an individual chip to provide simulation. This means that tasks that are usually handle by the included microprocessor like wear-leveling and garbage collection (better known as TRIM), now have to be taken care of by the filesystem; this is why YAFFS2 is used. YAFFS2 is designed specifically for the characteristics of NAND flash and is a robust log-structured file system that holds data integrity as a high priority, as well as providing wear-leveling and highly optimized garbage collection strategies. To do support some of these features YAFFS2 stores information about each page/block of data in what is called Out-Of-Bounds (OOB) area that contains meta-data, bad block bits, ECC, and tags that the filesystem uses. This is why it is essential to do a bit-by-bit extraction, because if data is obtained through the filesystem then information in the OOB isn't retained therefore losing very valuable information that can help in an investigation.

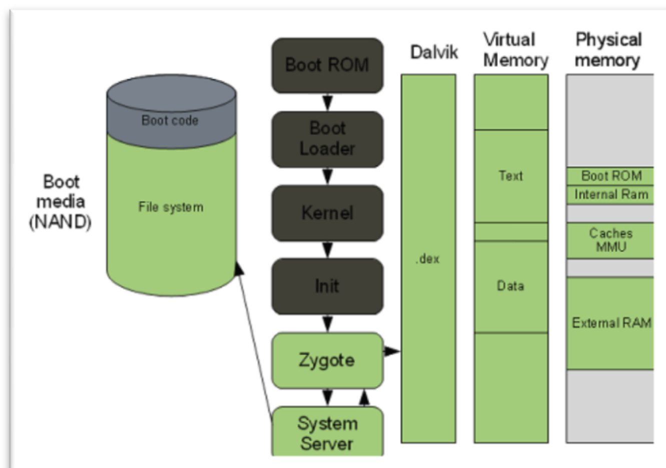




Boot Process

Understanding the Android boot process is essential in knowing how the operating system functions and what is performed at boot time. Looking into the source code we can see very interesting things in the startup scripts of what is being done (we will use this concept later). There are six fundamental steps in Android boot process which I'll briefly go over:

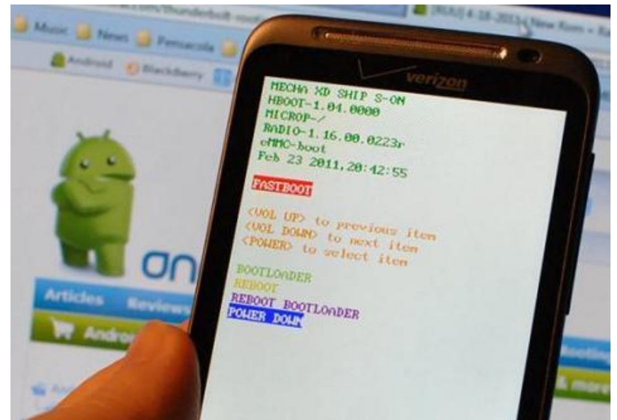
1. Power on and on-chip boot ROM code execution: This step is hardware specific in how the processor and other hardware units initialize on power-up, but the important part is that this is where the boot media is located and the bootloader is then copied to internal RAM and then executed.
2. The Boot Loader: Here the Initial Program Loader (IPL) detects and sets up the external RAM, copies the Second Program Loader (SPL) to internal RAM and transfers execution to it. The SPL is in charge of loading the OS and other possible mode such as the Recovery partition or the Fastboot protocol. SPL copies the Linux kernel that is located on the boot media to RAM and then transfers execution to the kernel.
3. The Linux Kernel: Now that the Linux kernel is loaded it begins its usual boot process including setting up more features as well as loading rootfs.
4. The init process: This is similar to the same init.d process that computer running Linux goes through when



booting up. Here startup scripts are ran to setup and do various functions, and start core services.

5. <u>Zygote and Dalvik:</u> This next process is called the Zygote process and is in charge of setting up the Java runtime environment (Dalvik), and then registers a socket with the system so it can communicate. Now applications can run by requesting new Dalvik virtual machines that each one runs in.
6. <u>The System Server:</u> Finally the system server begins running which takes care of core features like telephony, networking, and others.

Boot Loader



The Android bootloader is what is ran before any part of the actual operating system is loaded. It can be accessed on any Android device by holding down a device specific combination of buttons on power-up. This is important because we want to be able to access the disk that the data is stored on while bypassing the filesystem so that a bit-by-bit extraction can be performed. Here is where we look to somehow load a program to be able to do this. Luckily the bootloader has more options other than just booting into the operating system. One option is to load the fastboot protocol which allows you to modify the various partitions on the device. Unfortunately, this option isn't always present, and even if it is, is usually unable to modify partitions do to a security feature enabled by most manufacturers. The other option at the bootloader menu is to execute the Recovery partition, but it is simply preloaded onto the device by the manufacturer to recover or service the device if corruption occurs.

Data Acquisition

In acquiring data from an Android device there are a few tools that need to be discussed that will be used. This includes the Android Debug Bridge, the Recovery partition, and the Custom Recovery Partition.

Android Debug Bridge

The Android Debug Bridge (ADB) is included with the Android Software Development Kit which is free to download and use. ADB is a command line tool that allows you to communicate with an Android device through usb to transfer files, run a remote shell, and some other commands. Any Android can be used with the ADB by enabling debugging mode in the settings menu located at Settings→Applications→Developement→USB debugging. As mentioned before he ADB can be used to retrieve some file from the device but since this is through the filesystem, permissions are enforced so that not all files are accessible as well as the OOB information.

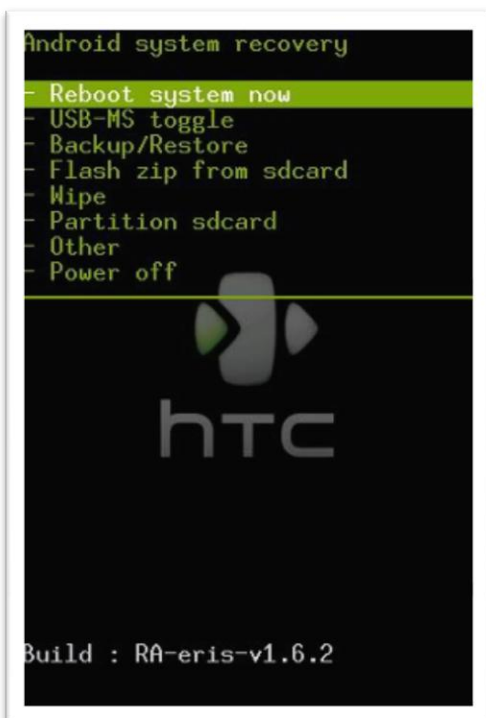
Recovery Partition

The Recovery partition is preloaded onto the device by the manufacturer in case the phone needs to be serviced or formatted from an irrecoverable state. The idea is to overwrite the Recovery partition with a custom image so that it can be ran from the bootloader, therefore giving us full access to the disk because the operating system and filesystem are never loaded. The problem is that since we can't use fastboot, the Recovery partition is only accessible through the filesystem by which it's is protected by.

However, an exploit was discovered in the Android operating system that allows us to modify this partition's permissions so that we are freely able to modify it. The exploit is within the startup scripts executed by the kernel during the boot process. One script in particular modifies the permission of a file so that is can be modified by any user. Therefore, using the ADB, we can simply replace the file being modified by the startup script with a symbolic link (named the same as the file) pointing to the recovery partition. Now, once the device reboots, the startup script runs and modifies the permission bits on the recovery partition (thinking it is the usual file), giving us full access to it. Now we can use ADB to put our custom recovery image and a simple program on the device, and use the program to flash our custom image to the recovery partition. The commands to do this are as listed here:

```
adb push recovery.img /data/local/
adb push flash_image /data/local/
adb shell chmod 777 /data/local/recovery.img
adb shell chmod 777 /data/local/flash_image
adb shell rm /data/local/rights/mid.txt
adb shell ln -s /dev/mtd/mtd1 /data/local/rights/mid.txt
adb reboot
then once booted...
adb shell /data/local/flash_image recovery /data/local/recovery.img
```

Custom Recovery ROM



The custom recovery image ("ROM' and "image" can be used interchangeably) we flash onto the device recovery partition needs a few essential things to allow us to perform data extraction. First, a communication protocol needs to be employed so that we can transfer data to/from the device. For this, the ADB is from the Android SDK since it is open source. Next we need something to be able to execute commands and run the tools we need to use. Here, software called BusyBox is employed which is essentially stripped-down Unix tools that are combined into a single executable (it can be thought of as a tiny version of linux). Lastly, we need a tools that can actually extract the data from the device. We could simply use dd for this, but I utilized a tool called NANDroid which is specifically written for use on Android devices with flash disks. NANDroid simply creates bit-by-bit exact image copies for any of the partitions on the device. Instead of building my own recovery image I used a popular one that already existed called "Amon_RA" which contains all of these utilities. Now that we have our custom image flashed in the recovery partition, when we boot up the device to the bootloader and select the recovery partition we are greeted

with a custom menu that gives access to various functions that can be ran from the device itself, as well as ADB running in the background so that we may run a remote shell that gives us full access to the device.

Data Analysis

Once the data extraction is complete we must now analyze the obtained information to make sense of it and reconstruct events or actions to understand what the device was being used for. Because we have an exact copy of the data from the device we have many different options on how to analyze the information. Unfortunately, forensic programs have not yet caught up with the mobile device explosion and therefore do not support YAFFS2, making analysis using the file system very difficult. We can however still perform many of the usual analysis methods used in digital forensics.

Simple Methods

The most basic method to use would be simply using a hex editor to manually examine the images obtained. A hex editor allows us to view the raw bytes of data contained within the image, but is not very useful unless we know where and what we are looking for since we will view all data, regardless what file types it contains, in its hexadecimal form.

The next method would be to use the "string" program which simply prints out any printable characters found within the image. This way you could save all the strings output from the program and then search the file for the information needed. The advantage of this is that it is very quick, but there are still a lot of downfalls such as it only being able to find printable string, which leaves out compressed data, images, and other file formats that don't simply store information in raw strings. Also, like using a hex editor, you would have to be looking for something instead of being able to "browse" through the data; yet, it is better than the hex editor because you can employ regular expressions is searching so that you don't need to know exactly what you're looking for.

The last method is to just mount the image and manually look through the filesystem. It is very important to remember that we should never modify the extracted information, so when the image is mounted we must make sure it is mounted as read-only. Also, to be able to mount the file system the image we obtained must also contain the OOB information or else the file system will not be able to load properly. An issue with this last method is that since YAFFS2 is constructed specifically for flash devices, we cannot simply mount the image as we would any other. To mount a YAFFS2 image we must load special kernel modules and utilities that will enable us to emulate a flash device that we will then be able to mount the image to. This is possible to do but sometimes difficult to execute because the process is not yet standard and can be hard to work with sometimes.

File Carving

File carving is an extremely useful method in digital forensics because it allows for data that has been "deleted" or hidden is some way to be recovered for analysis. The tool I used for this is called "scalpel". Scalpel is based off of another file carving program called Foremost which was originally written by the United States Air Force Office of Special Investigations. Scalpel read from a database of specified header and footers of certain file types, reads the supplied data, and when it finds a sequence of bytes that matches the header and then footer, it

“carves” that piece out of the data and creates a new file containing the information found matching the header and footer. This way files are found regardless if they were hidden or deleted from the file system because the entire raw image is searched. One things to keep in mind is since the OOB data is used by the file system and no data from the user is actually kept within it, it must be removed before file carving is done or else the carver will not work properly since each block of data will have the extra OOB information not pertaining to the actual data. I did this with a simple python script that strips the OOB data for each block, which is simple because we know the blocks’ and OOBs’ fixed lengths. There are some caveats to this though, like false-positives and fragmented data.

Results

Something that I realized quickly was that most of the user’s data was stored on the SD card inserted in the device. This seems to be a trend for Android devices because internal memory isn’t very large, so data and even applications are installed/stored on the SD card. Since the SD card is removable and does not use YAFFS2, we are able to use traditional forensics techniques for extraction and analysis. However, it is very important to realize that Android can use space on the SD card as cache for certain things, therefore it is important to obtain as much data as possible from the SD card before removing it or powering down the device. For this I simply used dd through ADB while the device was running to obtain a an image of the SD card of the device.

Once I had my data extracted from the device using the exploit, custom recovery image, and NANDroid I first tried using some simple methods for analysis. I quickly found that a hex editor is pointless unless you really know what and where data is you want, but I founds that using “strings” was surprisingly useful. Since “strings” is so fast from its simplicity, I was able to look for specific things like Google Map queries from which I was easily able to obtain recently visited addresses and even the user’s home address. One thing to note however is that you lose the context of the results since you don’t know when and how the queries are made. I then found that mounting the obtained images was useful for browsing through the files, and since everything obtains its structure we can open file types such as images and PDFs to see their contents.

The most interesting results however were from file carving. Performing the process on the obtained images showed images and files recovered from various parts of the system and cache. Files such as images, PDFs, zip files, documents, and databases were carved but there were many false-positives and incomplete files because of fragmentation. Also, I’m not sure how much “deleted” data was recovered because Android devices’ storage doesn’t act like typically block device, YAFFS2 performs garbage collection meaning that “deleted” files may actually become erased making file carving useless on flash disks. But, the SD card doesn’t use YAFFS2 and since it contains most of the user data, carving can successfully be used to recover “deleted” files from it.

Future Work

There is a lot of more work that can be done is this area, especially in improving file carving techniques to improve false-positive results and incorrect carvings from data fragmentation. They key thing is that since we can obtain an exact image from the device, we have the file system intact which contains meta-data (like the OOB) about the data within it. Using all this information we can potentially reconstruct any information from the device regardless if it is fragmented or not. Also, since YAFFS2 is a log-structured file system we can

perform special timeline analysis using its information. More research can also be performed on the specific areas of the device that are used to store certain information, and the format of that information. Inspecting commonly used applications like Facebook or Maps, we can define special header/footers for the file carver to use in inspecting the data. Also, services for the GPS, cellular, and WiFi systems can be looked at to determine if/where the data they collect is cached/stored so that the devices physical and logical locations can be reconstructed for an investigation.