Towards a General Collection Methodology for Android Devices

Timothy Vidas
Carnegie Mellon ECE/CyLab
tvidas@cmu.edu

Chengye Zhang
Carnegie Mellon INI/CyLab
chengyez@andrew.cmu.edu

Nicolas Christin
Carnegie Mellon INI/CyLab
nicolasc@andrew.cmu.edu

***Abstract*—The Android platform has been deployed across a wide range of devices, predominately mobile phones, bringing unprecedented common software features to a diverse set of devices independent of carrier and manufacturer. Modern digital forensics processes differentiate collection and analysis, with collection ideally only occurring once and the subsequent analysis relying upon proper collection. After exploring special device boot modes and Android's partitioning schema we detail the composition of an Android bootable image and discuss the creation of such an image designed for forensic collection.The major contribution of this paper is a general process for data collection of Android devices and related results of experiments carried out on several specific devices.**

***Index Terms*—Android Framework, Mobile Devices, Digital Forensics, Collection, Acquisition**

I. Introduction

Traditionally, mobile forensics requires procedures that are very specific to device manufacturer and/or model for both collection and analysis. Not only do mobile phones employ a diversity of cables, interfaces and form factors, but the devices have unique software, memory layouts and storage techniques. This amazing diversity has led to the digital forensics practitioner being assaulted with complex kits containing a plethora of cables and data collection techniques [37]. Mobile devices, mobile phones in particular, are ever present in today's society containing a wealth of information for the analyst.

At the end of 2010, 31% of U.S mobile consumers owned smartphones, with all the the top ten selling phones being smartphones [5]. Android, the Google-backed mobile software framework, is enjoying a larger market share and growth factor than Apple's iPhone [11], [26], quickly making Android a major player in the mobile market. Many existing manufactures produce Android based devices and all major carriers sell said devices. Indeed, Android can now be found in the living room powering GoogleTV, in upcoming Chevy and Ford vehicles and Sony's new "PlayStation Phone:" Xperia Play. Smartphone prices are declining, shrinking the market share of the less powerful "feature phones" which may also eventually be powered by the free Android software.

The ubiquity of devices utilizing the Android framework facilitates exploiting common properties to minimize the diversity required of digital forensics tools while simultaneously maximizing the potential for sound data collection. Manufactures and carriers tend to preserve competitive advantage by adding additional features to and providing additional services through mobile devices, yet Android based devices share a common framework that we utilize to perform collection (sometimes called "acquisition"). To our knowledge, this paper represents the first work done toward a general collection method for Android based devices.

II. Related Work

While the adoption of smartphones is rising, mobile phones themselves are not a particularly new technology. The array of mobile phones and pricing plans available today is indicative of the wide desires and needs of the user base, and must be coped with by digital forensics practitioners. In this section we present a progression of work related to digital forensics on mobile devices from what would today be considered a cheap "feature phone" to other work particularly targeting modern smartphones.

Willassen, in 2003, outlines many items of interest for GSM based mobile phone analysis: location, SMS, contacts, etc (refer to Appendix I for a brief overview of this data as it pertains to Android) [35]. In 2006 the same author explored collection methods for commodity mobile phones, in particular the use of physical access to the circuit board interface (e.g., JTAG port) or physically removing memory chips for later data collection via a chip programmer. Also in 2006, Casadei introduced a live collection technique, dubbed SIMbrush, facilitating entire file system collection (for unprotected files) particular to SIM devices [16].

In 2007, Al-Zarouni investigated the use of mobile phone flashing tools in regard to digital forensics [12]. The author concluded that while forensically sound, the primary use of the tool was to write to, not read from, the device. Read capability varied widely across device brands and models and consequently flashing tools may prove problematic as a means of collection.

Also in 2007, Mokhonoana and Oliver detailed a collection method for Symbian OSv7 devices [27], and Distenfano in 2008 explored a Symbian OSv8 collection method in [17]. Indeed, due to the vast diversity in the mobile space, it is very common for research to be scoped to particular operating systems or hardware platforms such as Blackberry [19], CDMA [28], iPhone 3Gs [14], Nokia [36], etc.

The demand for mobile forensics combined with the diversity of the mobile device market has led to a myriad of mobile forensics tools. In 2006, Ayers et al. compared existing

tools according to their acquisition, examination and reporting functions concluding that typical mobile phone information such as the IMEI and SMS/MMS could be discovered by existing tools [13]. The same year Williamson et al. studied the performance of mobile forensic tools particular to Nokia phones [36]. In 2007, Jansen and Ayers again compared existing tools on contemporary mobile phones and collected their work into a NIST report [24]. Later, in [37] Yates notes the diversity of the mobile device market and the associated complexity presented to a practitioner attempting to select the appropriate digital forensics tool. Just the comparison papers mentioned here cover: Cell Seizure, GSM.XRY, MOBILedit! Forensic, TULP 2G, Forensic Card Reader, ForensicSIM, SIMCon, SIMIS and Oxygen Phone Manager.

Specifically addressing Android devices, in 2009 Hoog discussed Android forensics including collection methods for Android mobile phones such as retrieving files on an active phone using an application debugging feature provided in the Android SDK, commercial tools such as Paraben's Device Seizure and "rooting" the HTC G1 [23]. Rooting is common vernacular for gaining administrative(root) access to a device where the user is intended to only have unprivileged access, such as a mobile phone. We explore the rooting further in section V-B.

In 2010, Thing et al. explore live memory forensics on Android to collect process memory [33]. Thing utilized the `PTRACE_ATTACH` method of invoking `ptrace` to trace existing processes. This technique is often used for debugging [31] but often repurposed for malware analysis [15].

III. Background

Built upon a Linux kernel, Android uses operating system primitives (such as processes and user IDs) as well as a Java Virtual Machine (Dalvik) to isolate apps providing a safety sandbox [32].

Android applications are typically written in JAVA and interact with the Android framework through a well defined API. For performance reasons, developers can create software that runs natively on the hardware avoiding overhead induced by the JAVA virtual machine. An SDK(Software Development Kit) and NDK(Native Development Kit) are made available for application development (not intended for lower level development such as kernel modifications). The SDK makes high level software development very approachable by providing features such as a full featured emulator, an Eclipse (the preferred Integrated Development Environment) add-on specific to Android and a special Android Debug Bridge (adb) to enable debugging information from an emulator instance or USB connected physical device [1].

The Android Debug Bridge consists of software components on the device (or emulator instance) and on the developer's machine connected via USB or TCP. Using the feature, the developer can not only observe debug information, but also perform an assortment of other actions such as installing software (bypassing the Market application) reboot the device, and even open an interactive remote shell. Note that adb is typically not enabled in production devices and must be enabled by the user[1]. For low level experimentation, Android is open source, the code can be easily obtained and compiled [10].

Some vendors, such the manufacturer of the G1 phone: HTC, embrace developers with a rich web site providing not only documentation but also pre-built tools and even phone images [6]. Contrarily, other manufactures never release such information to the public and protect phone images closely as coveted intellectual property.

For the rest of this paper we assume a more strict case when the device is "obstructed" via a screen lock mechanism as defined in [24]. While the technique described in the paper will work equally as well on an "unobstructed" device and is capable of a more comprehensive collection, it may be deemed simpler to simply enable adb on an unobstructed device and perform an unprivileged, logical collection. Such a collection would strictly contain less information due to the inaccessibility of unallocated storage locations and permissions enforced by the running Android system (the adb process does not execute with root privileges).

IV. Collection Objectives

Since our data collection is intended for forensics purposes, we must consider several constraints and desirable qualities on the technique. Ideally, an "exact copy" of the device and it's data can be obtained, even though by simply interacting with the device we alter it's state somewhat[2] we strive to obtain as close to an exact copy as possible [20]. Here we enumerate desirable criteria for the collection process.

- *Data Preservation.* Data storage locations that house user data are of primary importance, as this type of data is likely to be more valuable to investigation than system files common to many devices. This view does not completely discount the collection of storage less likely to house user data, it only places more value on user data. The ability to capture system information may, in fact, be very useful in tandem with user data. Though quite likely less common, particular cases may dictate that cache information, firmware and kernel crash information be critical. Again, ideally, everything can be collected as an "exact copy."
- *Atomic Collection.* Data should be collected as atomically as possible with respect to the device being collected. If a device is currently executing instructions and manipulating storage the collected state may not be a valid. Consider a disk that is defragmenting while an external process is copying the same disk. Given that the copy will take some amount of time, it's perfectly reasonable that a file is moved by defragmentation to a cluster that has already been copied by the external process. In this case, the copy will not contain the file! A simple solution is to

[1] via Settings - Applications - Development - USB Debugging, or similar

[2] In [20] this is dubbed the "Heisenberg principle of data gathering in systems" after the famous physicist and associated principle.

copy the disk while no other actions are being performed to the disk. Similar situations may arise relating to the validity of the file system when meta information such as allocation tables are altered during the copy process.
- *Correctness.* In relation to the atomicity criteria mentioned above, there is an obvious need for correctness. Even given the ability to copy atomically with respect to the device, the data must be copied correctly. Software must truly copy the data from source to destination and integrity must be preserved in transit.
- *Determinism.* The process must be repeatable so that the practitioner has a expectation that the process will indeed collect the data in question. Subsequent collections on the same device, in the same state, should ideally produce identical results.
- *Usability.* The process must be usable, and occur in a feasible amount of time.

Depending on the hardware characteristics of the device, other precautions may be required. For example, somehow jamming or otherwise preventing a mobile phone from receiving network communication may be desirable as incoming data will certainly change the state of the phone, potentially deleting valuable data.

V. Collection Process

Our technique repurposes the recovery partition and associated recovery mode of an Android device for collection purposes. For any device, collection is going to be a multi-step process that requires a *collection recovery image*. An outline on how to create such an image is outlined in section V-C.

Once a recovery image has been obtained, it is flashed to the device using the device specific instructions such as those outlined in section VI below. After a collection recovery image is loaded on the device, the device is rebooted into recovery mode and connected to a computer that has `adb` (from the Android SDK) installed. The `adb` program can then be used to verify that the device is connected (`./adb devices`), and remotely execute programs from the recovery image now executing on the device (`./adb shell`).

At this point the recommended procedure for collection is to port forward TCP ports from the device using `adb`, start a receiving process on the computer and transfer data from the storage devices to the local device TCP port using a data dumping utility and a simple program that writes the output of the data dumping utility to a socket. The TCP transfer software written by the authors also calculates an integrity hash as data is written to the socket. The display of an integrity hash allows for the verification of correct transfer by checking the hash displayed in the adb shell with one calculated independently on the collected image on the computer.

The data dumping utility employed depends somewhat on the characteristics of the device. Many Android based devices utilize Memory Technology Devices (MTD). The MTD system is "an abstraction layer for raw flash devices" [8] that allows software to utilize a single interface to access a variety of flash technologies. For MTD devices, nanddump[3] can be used to collect NAND data independent of the higher level filesystem deployed on the memory. For devices that do not employ MTD other collection techniques must be employed. For example, the `dd` utility can be used to copy data. It is also important to note that not all data is necessarily stored in onboard memory. It is very common for Android devices to support one or more SDcards. Not only can the user elect to store some applications and data on such a device, some manufacturers may choose to store the entire user data partition on this media.

A. Android Partitioning

Android devices typically consist of several partitions typically mapped to MTD devices. Exact partitioning schema depend upon vendor implementation, but a typical scheme can be found in Table I. There are typically six partitions found on Android devices the most common being system, userdata, cache, boot, and recovery. As also seen in the table, many Android based devices utilize the YAFFS2 (Yet Another Flash File System 2) file system which was designed for use on flash memory. Newer devices may be found to utilize the EXT4 file system [29]. As one may expect, the "bootimg" denoted in Table I is a "bootable image" which is detailed further in section V-C. SDcard locations, sometimes marketed as "internal" or "external" are both typically identified by `/dev/block/mmcblkXpY` where X is the card ID and Y is the partition ID on the card. The SDcard device is typically mounted to `/sdcard` or `/mnt/sdcard`.

Those interested in exploring the forensic analysis on Android devices will likely be most interested in the userdata and system partitions. It is important to note that during normal operation, no user data is stored in the recovery partition, so corruption or overwriting of data in this partition is unlikely to change content on the device that may subsequently be relied upon in court. The recovery partition, and associated recovery boot mode, are critical to the collection technique explored in this paper.

B. To Root or not to Root

Some have suggested via presentations [23], blogs posts [3], or mobile phone "modding" forums [9] methods of data collection that require "rooting" a device. Rooting a device typically involves exploiting a security vulnerability (which is typically device and software version dependent) with the intention of installing unsupported software on the device. The motivation for rooting a device ranges from ideological desire to have control over the device that the user owns, to circumventing carrier specific controls preventing the use of particular software, to upgrading to a more recent version of Android than the carrier currently supports (some carriers have very long update cycles), and many other reasons not mentioned here. "Rooting" a device for the purpose of forensic collection is not exemplary for several reasons, among them:

[3] a NAND dump utility written in 2000 by David Woodhouse and Steven Hill

Path	Name	File System	Mount Point	Description
/dev/mtd/mtd0	pds	yaffs2	/config	Configuration data
/dev/mtd/mtd1	misc	-	N/A	Memory Partitioning data
/dev/mtd/mtd2	boot	bootimg	N/A	Bootable (typical boot)
/dev/mtd/mtd3	recovery	bootimg	N/A	Bootable (recovery mode)
/dev/mtd/mtd4	system	yaffs2	/system	System files, Applications, Vendor additions, Read-Only,
/dev/mtd/mtd5	cache	yaffs2	/cache	Cache Files
/dev/mtd/mtd6	userdata	yaffs2	/data	User data (Applications)
/dev/mtd/mtd7	kpanic	-	N/A	Crash Log

TABLE I
PARTITION INFORMATION TYPICAL OF AN ANDROID DEVICE

- Rooting a device typically leverages a software flaw often particular to specific model and software versioning on a device. If the device is locked, an investigator may not be able to verify software versions running on the device. The inability to verify software versioning decreases the changes of successfully rooting the device and increases the chances of inflicting damage upon the device and/or the data to be collected.
- Rooting the device alters portions of the device that may store user data. If avoidable, collection methods should not change the content on the device that "may subsequently be relied upon in court [24]." In some areas of digital forensics this is unavoidable, for example collection of memory from a running machine [34]. In instances where collection can be performed without modifying the data to be collected, collection should be done in such a manner [20].
- Rooting a device undermines Android's security model. A rooted device often permits easy escalation of privilege. With the device in a normal operating mode, the combination of easy privilege escalation with typical app execution and general network access can lead to malicious remote code execution as shown with the iPhone in [30].

C. Recovery Partition

Personal computers commonly allow users to configure BIOS boot time password passwords or even stronger security with Trusted Platform Modules (TPM) now available on an estimated 250 million systems [22]. The Android framework is commonly deployed on small devices such as mobile phones, tablets, and televisions which do not enjoy any kind of protected boot. Android devices ship with a partitioning scheme similar to those in Table I including a recovery partition. The recovery partition has special properties ostensibly used for recovery purposes. By booting a device into "recovery mode," the normal boot process is circumvented and the boot target is the bootimg currently loaded in the recovery partition. As seen Figure 1, common features found in a manufacturer installed recovery image include, wiping user data and updating the device. Similar to the normal operating mode, the factory recovery mode image commonly does not support adb, but also does not commonly enable the RF component of a

Fig. 1. **Recovery Mode** Motorola Droid booted to a typical recovery image.

device. Here we begin to craft a special collection oriented recovery bootimg. Such a bootimg will have unfettered access to memory not inhibited by access control as in [25] or varied memory access capabilities as in [12].

An Android bootimg consists of header, a kernel, ramdisk (initrd) and optional secondary image each page aligned. The bootimg header, defined in bootimg.h found in the Android source [10], contains the magic signature "ANDROID!", an ID field [4] and meta information about the size and memory locations in which to load the kernel, ramdisk and secondary image.

The ramdisk portion of the bootimg is a compressed (gzip or lmza) cpio file containing an initial ram disk (initrd) directory structure for the kernel. This directory structure can be amended to include additional binary programs in order

[4]the ID field is typically a SHA1SUM of the kernel, kernel size, ramdisk, ramdisk size, secondary image, and secondary image size used to uniquely identify the bootimg based solely on the first page of the bootimg

to modify the behavior of the bootimg. In addition to adding the desired binary programs, some other modifications from a standard bootimg are necessary.

A practitioner, using an existing bootimg as a collection tool, may never intend to create a bootimg from scratch. In this case it is still useful to understand the file structure of a bootimg in order to verify the operation of the collection tool. Using common Linux commands a bootimg may be split into its respective pieces and the ramdisk may be ungzipped and "unCPIOed" for inspection. Once a custom, collection oriented recovery image has been created or obtained it must be "flashed" onto the device.

Proof of concept bootimgs created by the authors include modifying the ramdisk by modifying the `default.prop` properties file, `init.rc` file and adding `adbd`, `su`, `nanddump`, and custom transfer binaries. The primary modifications to the properties file include enabling `ro.debuggable` and `persist.service.adb.enable` to enable the use of adb. The `init.rc` file is used to start the adbd service (on `property:persist.service.adb.enable=1 start adbd`) and set permissive permissions on the added binaries and MTD devices to be collected.

When possible, we prefer the use of nanddump over other data duplication software since nanddump was designed specifically to dump the contents of flash devices. This allows for as close to a physical duplication as possible, containing more information than a file system level copy. There are currently no procedures for garnering valuable information from this extra data, but it is prudent to collect such data in anticipation of future techniques. Having this extra data causes no harm as subsequent analysis of the collected data will still provide at least as much information as a file system level copy would have provided.

It is desirable to create a small set of collection recovery images that together support a wide range of devices. To this end, the compilation process for a recovery image should target the older processor types so that the resulting applications will correctly execute on all backward compatible devices.

VI. Device Specifics

While the Android framework does provide some desirable common qualities, it is unlikely that a single all-encompassing bootimg that properly handles all devices can be created. Even though the Android framework provides a common interface at the application level and presents a familiar UI to the user, the devices are still unique at the hardware level employing different connectors, processors, etc. For this reason bootimg's that support several, similar devices can be created, but the creation of a universal bootimg is improbable.

Devices may be booted into different modes which are invoked via hardware key combinations during boot. These modes allow special functionality such as the ability to clean the phone of user data or flash new software onto the device. Each device has unique physical characteristics: number of physical keys, touchscreen, SD card slot, physical keyboard, etc. which in turn cause the key combinations required for special modes to not be uniform across devices. Table II shows a key combinations and their associated modes for a small set of devices.

The method of "flashing" the recovery partition will likely always be somewhat unique to different devices, though manufactures tend to re-use similar sequences across a product line. A lab that regularly encounters Android devices may wish to compile a comprehensive list of key combinations and modes similar to existing lists for computer BIOS' [2]. Flashing a device is a destructive process, the storage area that is "flashed to" will be irreversibly overwritten. The remainder of this section explores "flashing" methods for several devices, in each case only the recovery partition is overwritten with all other existing partition and data within remaining intact.

The low level bootloader and software that powers flashing modes is designed for specific uses typically reserved for the carrier or manufacturer, not the end user. These modes are not particularly robust in features and the user should exercise caution when accessing a device in this manner. Improper use can easily result in hindering data collection. Similarly, software features like USB negotiation are not as robust as other devices with which the user may be familiar. Using flashing software in VMWare, while possibly desirable, is not possible due to the device not properly negotiating with virtual machine's virtual USB controller.

A. Example: Motorola Droid

The Motorola Droid is a Verizon device with screen that slides to the right revealing a full QWERTY keyboard and D-Pad (Directional Pad). In addition to the keyboard the Droid has volume up/down, power and a camera button around the outside edge of the device. The Droid has a build in microSD card, a port for an additional microSD card, and has an ARM Cortex A8 550 mHz processor. It originally shipped with Android 2.0 [4].

Fig. 2. **Flash Mode** Motorola Droid booted to flash mode.

Device	Mode	Key Combination	Description
Motorola Droid	flash	D-Pad UP + power	Mode that allows flashing via RSDLite
Motorola Droid	flash	camera + power	Mode that allows flashing via RSDLite
Motorola Droid	recovery	power + x	Boot to recovery partition (then camera + volup to display menu)
HTC G1	flash	power + back	Fastboot mode
HTC G1	flash	power + camera	Boot mode (switch to fastboot via 'back')
HTC G1	recovery	power + home	Boot to recovery partition
Samsung Captivate	flash	volup + voldn (then insert USB)	Boot to Samsung "force download" mode
Samsung Captivate	recovery	power + volup + voldn	Boot to recovery partition
Samsung Galaxy Tab	flash	power + voldn	Boot to Samsung "force download" mode
Samsung Galaxy Tab	recovery	power + volup	Boot to recovery partition

TABLE II
BOOT MODES FOR SELECT ANDROID DEVICES

The Droid has a special flash bootmode (shown in Figure 2) that can be entered by holding the camera button while powering on the device. This special boot mode allows flashing of the device's recovery partition. Motorola RSD Lite software (Windows only) can facilitate flashing of the recovery partition, but does not accept a bootimg file in it's native form: an RSD Lite compatible .sbf file containing the bootimg must be created. An .sdf file is comprised[5] of a header with file magic and a count of the parts in the file, each part also contains a header specifying the destination address, flash size, checksum and, of course, the image to flash, in this case the bootimg. Once an .sbf file containing the bootimg has been created the Droid, booted in flash mode, can be attached to a computer running RSD Lite and the device can be flashed with the .sbf file (and thus the contained bootimg). While not strictly required, re-booting the Droid into flash mode while connected the RSD Lite will allow RSD Lite to register a success message.

B. Example: HTC G1

The HTC G1 (shown in Figure 3) has a Qualcomm MSM7210A 528 mHz processor (ARMv6 instruction set), a full QWERTY keyboard, and an external microSD card port. In addition to the keyboard the G1 has a track ball, and physical volume up/down, camera, send, home, menu, back, and end/power buttons. One hardware feature germane to collection is that the G1's has a special HTC USB+Audio port (ExtUSB) in lieu of the more common microUSB port [7]. If the special ExtUSB cable that shipped with the device is not available, a standard miniUSB cable can be used for both recovery and fastboot modes.

The G1 employs a boot method called fastboot (shown in Figure 4). Fastboot requires a fastboot compatible boot loader and a `fastboot` program on a personal computer. The `fastboot` program can be compiled from Android source [10] or pre-compiled versions for Windows, Linux and OSX can conveniently be obtained via HTC's developer website [6]. After booting into fastboot mode

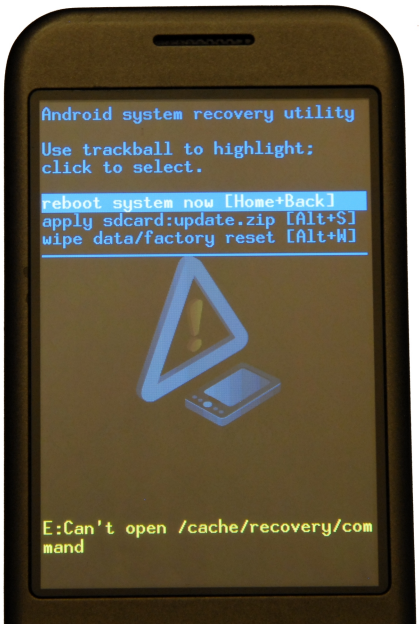

Fig. 3. **Recovery Mode** HTC G1 booted to a typical recovery image.

and connecting the device to the computer via the special HTC cable or miniUSB[6], the `fastboot` program can be used to enumerate devices attached (`./fastboot devices`) and to flash an image to the device (`./fastboot flash recovery` *`bootimg_filename`*). Fastboot also allows directly booting to a kernel and ramdisk located on a connected computer (`./fastboot boot` *`kernel_filename ramdisk_filename`*). Which may be slightly preferred over flashing the recovery image as the existing recovery image on the device remains intact. For consistency in the collection process we suggest maintaining a set of recovery images and flashing the recovery image for every collection.

[5]the exact structure of an sbf is not extremely important here, software that can create a well formed .sbf file when provided a bootimg can be found at http://www.ece.cmu.edu/~tvidas/

[6]The author observed that the G1 in fastboot mode was not recognized correctly when connected to a USB3 port. Readers may wish to specifically use USB2 ports.

Fig. 4. **Fastboot Mode** HTC G1 booted to fastboot mode.

Fig. 5. **Recovery Mode** Samsung Captivate booted to a typical recovery image.

C. Example: Samsung Captivate

The Captivate is part of Samsung's Galaxy S line of mobile phones, sold by AT&T (shown in Figure 5). The Captivate has a larger touch screen than the G1 and Droid, but it also has no QWERTY keyboard, in fact it only has 3 edge buttons: power, volume up and volume down. A standard microUSB port can be found at the top of the device behind a plastic sliding cover. In addition to typical internal hardware: 1GHz ARM Cortex A8, 512 MB of RAM, and 16GB internal SDcard, the Captivate also has a hardware graphics core.

Unlike the Droid and the G1 the Captivate employs Samsung's proprietary RFS (Robust FAT File System) and Samsung's OneNAND memory [21]. This requires Android to load kernel modules to support RFS and makes later analysis more difficult as there is no available software for parsing RFS related data. Instead of using MTD devices, the kernel modules create several STL (Sector Translation Layer) and BML[7] (Block Management Layer) block devices (`/dev/block/`). A partition table showing typical use of BML devices is shown in Table III. This type of device complicates collection slightly as it is not possible to read from some of the higher-layer STL devices, and collecting all of BML devices, while recommended, is not particularly useful as there is no way to analyze the resulting image.

Device	Name	Mount Point	Description
bml1	boot	-	Primary boot loader
bml2	pit	-	Partition map data
bml3	efs	/efs	Unknown.
bml4	SBL	-	Secondary boot loader
bml5	download	-	Download Mode
bml6[7]	param	/mnt/.lfs	Unknown (lfs)
bml7	kernel	N/A	kernel + initramfs
bml8	recovery	N/A	kernel + initramfs
bml9[7]	system	/system	Typical /system data (RFS)
bml10[7]	dbdata	/dbdata	dbcache (RFS)
bml11[7]	cache	/cache	cache (RFS)
bml12	modem	-	Modem software

TABLE III
PARTITION INFORMATION TYPICAL OF A SAMSUNG DEVICE

Much like the Droid, the Captivate has a special flash mode (also called download mode, shown in Figure 6), that can be entered by holding both volume buttons and then connecting the device to a computer for flashing. In this mode the phone can be flashed using an open source tool called Heimdall [18] or the closed source software Odin (Windows only). When using Odin prior to flashing, source files must be placed in a `.tar`[8] archive. Heimdall does not require files to be packaged as a `.tar` and is compatible with Windows, Linux and Mac OSX.

In addition to the more complex partitioning and file system structure, Samsung devices do not employ the typical bootimg structure in the recovery partition. The recovery image is an initramfs image. Initramfs, available in 2.6.x Linux kernels, is a root file system that is actually embedded into the kernel. The details of creating an image suitable for flashing to a device are slightly more complex than described above, but the same theory applies.

The presence of an older Secondary Boot Loader (SBL)[9] will likely not utilize the BML8 recovery partition. Instead the normal boot mode and recovery mode share the same kernel. Unfortunately it is very difficult to tell the version of the SBL without interacting with the device. However the SBL can be

[7]In addition to the BML, the associated STL should also be collected because this is an RFS partition and at this time the STL data is easier to analyze than the BML.

[8]the .tar file *must* be POSIX 1003.1-1988 (ustar) format

[9]Prior to approx. Oct 2010

Fig. 6. **Download Mode** Samsung Captivate booted to a typical download mode for flashing.

flashed. So if after flashing the collection recovery image, if recovery mode is not working as expected, one may assume that an older SBL that does not boot to BML8 is on the device. The SBL can be flashed using Odin or Heimdall (but it requires having an exist SBL that is known to work with the device). Corrupting the SBL will make data collection very difficult because the device will no longer be able to reach download mode, as such flashing the SBL should be a last resort.

Note that some devices, such as the Samsung Galaxy Tab (Tablet), require a special cable. Unlike the HTC ExtUSB where a miniUSB cable can serve as a substitute, the 30 pin Galaxy Tab cable that ships with the tablet is the only method of connecting to the device.

VII. Discussion

We have demonstrated a general method for digital forensics collection on Android devices. Through special boot methods enabling the use of custom recovery bootimg, data on Android devices can be collected with very little probability of corrupting user data. Use of the recovery bootimg provides a consistent, repeatable method of collecting numerous Android devices without "rooting" the device in normal operating mode. We feel that this recovery bootimg method is both safer and had less impact to data likely to be useful for analysis.

Collection recovery images have been created for testing for the devices detailed in VI. The collection process involves calculating integrity hashes at the source and destination helping ensure the correctness of the collection. Data contained in the collected images was verified using standard a Linux distribution with MTD and yaffs2 support (see Appendix I). Where possible, we employed the use of a NAND dumping tool which collects more data than typical filesystem copy would collect. While current analysis techniques do not take advantage of this extra information, future techniques may.

Most devices transfer data at approximately 4.3 MB/s allowing for full collection to occur in a nominal amount of time. Even though the collection is not atomic, execution is restricted to the recovery partition and other partitions are not altered during collection resulting in an "exact copy" of original contents for all partitions other then the recovery partition.

Though no user studies have been performed, we feel that the solution is very approachable and could be adopted by practitioners. When thought of simply as a collection tool, the tool can easily be inspected for correctness.

VIII. Future Work

The software installed in collection bootimgs could easily be extended to further aide the practitioner. The menu presented on the screen when a bootimg is executed could have related menu options such as "transfer data" eliminating the need to run `./adb shell` on the collection computer. Similarly integrity hashes for collected partitions could be displayed on the device screen. By moving this functionality to the device there is less risk of user error, especially if a single computer is used to perform collection of several Android devices.

A comprehensive list of boot modes for Android devices, and associated flashing tools, should be created in order to have a reference in place prior to the need for collection on a particular device. Similarly, a comprehensive set of bootimgs supporting all Android devices should be created, maintained and tested.

Appendix I. Analysis of collected images

This appendix is not intended to be a comprehensive method of analysis, but only to serve as a brief means of verifying the collected images. Perhaps most important is the preparation of an analysis system, but even given a complete analysis system, locations and methods of data interpretation are left for future research. The methods detailed below are pertinent to a yaffs2 based device.

A. Analysis system preparation

System preparation steps must only be performed once per system. Mostly this consists of installing several packages if they are not already installed. In Fedora these packages are titled `mtd-utils`, `sqlite` and `kernel-devel`[10] (only if you want to build the yaffs2 kernel module yourself).

The most difficult part is the compilation of the yaffs2 kernel module, which is needed to mount yaffs2 partitions for earlier Android devices. To compile your own kernel module copy the kernel sources to a working directory, apply the patch supplied with the yaffs2 source[11], ensure MTD devices are built as modules, and building the modules. A shell log would look something like:

```
cp /usr/lib/<kernel> <somedir>
../yaffs2/patch-ker.sh  <somedir>
cd <somedir>
menuconfig
```

[10] make sure you obtain the source for the kernel you intend to run, in Fedora you can look in the /boot directory for config* files for all installed kernels

[11] available at http://www.aleph1.co.uk/yaffs2

```
make modules
make M=fs/yaffs2 modules
depmod
modprobe yaffs
```

In menuconfig (or by editing the .config file) you need to ensure that MTD,MTD_CHAR,MTD_BLOCK,MTD_NAND, and MTD_NAND_NANDSIM are all set to "m."

Installing the module may require slightly different syntax depending on your kernel configuration, such as `make M=fs/yaffs2 install_modules` and it may be needed to copy the compiled module to the kernel modules directory `cp fs/yaffs2/yaffs.ko /lib/modules/kernel/fs/yaffs/yaffs.ko` (check permissions to match other directories). Note that you don't need to recompile, and use the entire kernel.

Perhaps the easiest method may be to download a prepared VM from: http://www.ece.cmu.edu/~tvidas/.

B. Analysis Initialization

Analysis initialization must occur each time a device analyzed. Modules must be loaded:

```
modprobe mtd
modprobe mtdchar
modprobe mtdblock
modprobe nandsim first_id_byte=0x20 \
second_id_byte=0xac third_id_byte=0x00 \
fourth_id_byte=0x15
modprobe yaffs
```

The nandsim (NAND simulator) options determine the characteristics of the simulator such as 512 MB device with 2048 byte pages, if you need to specify other sizes see Table IV for more options[12]. Nandsim creates a pair of related devices `/dev/mtd0` and `/dev/mdtblock0`.

For each collected image, write the data to the virtual MTD device:

```
nandwrite -a -o /dev/mtd0 [dumpfile]
```

Finally mount the mtd device:

```
mount -t yaffs2 -o ro /dev/mtdblock0 \
/some/mount/point
```

Now the partitions can be browsed and easily inspected from `/some/mount/point`.

C. Common mobile phone data locations

It is important to note that Android "smart phones" are more complex than, and typically contain much more information than, a standard mobile phone, and the data locations described here are by not means comprehensive or universal. Recall that you will need to determine the partition associated with the original mount point for `/data` similar to Table I.

[12]NAND size and page size vary widely among manufactures, see http://www.linux-mtd.infradead.org/nand-data/nanddata.html for more information

Desired Size		Byte Specifier			
NAND	**Page**	**1**	**2**	**3**	**4**
16 MB	512 b	0x20	0x33	-	-
32 MB	512 b	0x20	0x35	-	-
64 MB	512 b	0x20	0x36	-	-
128 MB	512 b	0x20	0x78	-	-
256 MB	512 b	0x20	0x71	-	-
64 MB	2048 b	0x20	0xa2	0x00	0x15
128 MB	2048 b	0xec	0xa1	0x00	0x15
256 MB	2048 b	0x20	0xaa	0x00	0x15
512 MB	2048 b	0x20	0xac	0x00	0x15
1024 MB	2048 b	0xec	0xd3	0x51	0x95

TABLE IV
NANDSIM OPTIONS

Phone contacts, and call log data can be found at:

```
/data/data/com.android.providers.\
contacts/databases/contacts2.db
```

Calendar information:

```
/data/data/com.android.providers.\
calendar/databases/calendar.db
```

SMS and MMS messages:

```
/data/data/com.android.providers.\
telephony/databases/mmssms.db
```

Gmail and gtalk data:

```
/data/data/com.google.android.providers.\
gmail/databases/mailstore.cmu.android.\
<GMAILADDRESS>.db
```

Each of these databases are SQLite3 databases, the easiest way to 'export' data is via the `sqlite3`:

```
sqlite3 /data/data/com.android.providers.\
calendar/databases/calendar.db .dump \
> calender_raw.txt
```

But in many cases it may be more useful to actually leverage SQL:

```
sqlite3 com.android.providers.telephony \
/databases/mmssms.db 'select address, \
person, body, protocol, subject, body \
from sms'
```

A wealth of information is available in SQLite3 databases (Android's method of structured storage) both for packaged and user installed apps, but analysts should be aware that apps may elect to store information using proprietary methods.

ACKNOWLEDGMENTS

This work is supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and by the National Science Foundation under ITR award CCF-0424422 (Team for Research in Ubiquitous Secure Technology) and IGERT award DGE-0903659 (Usable Privacy and Security), as well as a hardware donation by Google Inc.

REFERENCES

[1] Android developers. http://developer.android.com/.

[2] Backdoor and default passwords. http://www.freelabs.com/~whitis/-security/backdoor.html.

[3] Computer forensics - forums - general discussion - mobile phone forensics - android / root access. http://www.forensicfocus.com/index.php?name=Forums&-file=viewtopic&t=5779.

[4] DROID by motorola - android phone - tech specs - motorola mobility, inc. USA. http://www.motorola.com/Consumers/US-EN/Consumer-Product-and-Services/Mobile-Phones/ci.Motorola-DROID-US-EN.alt.

[5] Factsheet: The U.S. media universe | nielsen wire. http://blog.nielsen.com/nielsenwire/online_mobile/factsheet-the-u-s-media-universe/.

[6] HTC - developer center. http://developer.htc.com/adp.html.

[7] HTC - products - T-Mobile g1 - specification. http://www.htc.com/www/product/g1/specification.html.

[8] Memory technology device (mtd) subsystem for linux. http://www.linux-mtd.infradead.org/faq/nand.html.

[9] [Q] android rooting [Archive] - xda-developers. http://forum.xda-developers.com/archive/index.php/t-866622.html.

[10] Welcome to android | android open source. http://source.android.com/.

[11] Android most popular operating system in U.S. among recent smartphone buyers | nielsen wire. http://blog.nielsen.com/nielsenwire/online_mobile/android-most-popular-operating-system-in-u-s-among-recent-smartphone-buyers/, Oct. 2010.

[12] M. Al-Zarouni. Introduction to mobile phone flasher devices and considerations for their use in mobile phone forensics. In *Proceedings of the 5th Australian digital forensics conference*, Dec. 2007.

[13] R. Ayers, W. Jansen, L. Moenner, and A. Delaitre. Cell phone forensic tools: An overview and analysis update, NIST interagency report (IR) 7387, march 2007.

[14] M. Bader and I. Baggili. iPhone 3GS forensics: Logical analysis using apple iTunes backup utility. *SMALL SCALE DIGITAL DEVICE FORENSICS JOURNAL*, 4(1), Sept. 2010.

[15] M. Burdach. Finding digital evidence in physical memory. *Black Hat, Las Vegas, NV*, 2006.

[16] F. Casadei, A. Savoldi, and P. Gubian. Forensics and SIM cards: an overview. *International Journal of Digital Evidence*, 5(1), 2006.

[17] A. Distefano and G. Me. An overall assessment of mobile internal acquisition tool. *digital investigation*, 5:S121–S127, 2008.

[18] B. Dobell. Heimdall website. https://github.com/Benjamin-Dobell/Heimdall.

[19] K. Fairbanks, K. Atreya, and H. Owen. BlackBerry IPD parsing for open source forensics. pages 195–199. IEEE, 2009.

[20] D. Farmer and W. Venema. *Forensic discovery*. Addison-Wesley, 2005.

[21] L. Flash Software Group, Samsung Electronics Co. Linustoreii samsung onenand flash linux device driver - gpl compliance, July 2008.

[22] T. C. Group. Trusted computing group (tcg) timeline. Apr. 2009.

[23] A. Hoog. Android forensics. *Mobile Forensics World*, 29, 2009.

[24] W. Jansen and R. Ayers. Guidelines on cell phone forensics. *NIST Special Publication*, 800:101, 2007.

[25] K. Kim, D. Hong, K. Chung, and J. Ryou. Data acquisition from cell phone using logical approach. *Proceedings of the World Academy of Science, Engineering and Technology*, 26, 2007.

[26] M. Meeker, S. Devitt, and L. Wu. Ten questions internet execs should ask & answer. San Fransisco, CA, Nov. 2010.

[27] P. Mokhonoana and M. Olivier. Acquisition of a symbian smart phones content with an on-phone forensic tool. 2007.

[28] C. Murphy. The fraternal clone method for CDMA cell phones. *SMALL SCALE DIGITAL DEVICE FORENSICS JOURNAL*, 3(1), June 2009.

[29] R. Paul. Ext4 filesystem hits android, no need to fear data loss. http://arstechnica.com/open-source/news/2010/12/ext4-filesystem-hits-android-no-need-to-fear-data-loss.ars, 2010.

[30] D. Reisinger. Another iPhone worm, but this one is serious | the digital home - CNET news. *cnet news*, Nov. 2009.

[31] D. Sarma and S. Vaddagiri. Debugging multiple threads or processes, Mar. 2002.

[32] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. Towards formal analysis of the Permission-Based security model for android. pages 87–92. IEEE, 2009.

[33] V. Thing, K. Ng, and E. Chang. Live memory forensics of mobile phones. *digital investigation*, 7:S74–S82, 2010.

[34] T. Vidas. The acquisition and analysis of random access memory. *Journal of Digital Forensic Practice*, 1(4):315–323, 2006.

[35] S. Willassen. Forensics and the GSM mobile telephone system. *International Journal of Digital Evidence*, 2(1):1–17, 2003.

[36] B. Williamson, P. Apeldoorn, B. Cheam, and M. McDonald. Forensic analysis of the contents of nokia mobile phones. page 36, 2006.

[37] I. Yates. Practical investigations of digital forensics tools for mobile devices. pages 156–162. ACM, 2010.