Faculty of Science and Technology
Department of Computer Science

Investigating the security issues surrounding usage of Ephemeral data within Android environments

-

Erlend Skog Høgset

INF-3981 Master's Thesis in Computer Science - August 2015

Abstract

With mobile devices managing more and more of our personal data, for many it has become a ubiquitous resource. This is also true in the workplace where they are instituting *Bring Your Own Device* practices. While this saves the enterprise money in terms of equipment, it also increases the diversity of devices brought to work. This presents security problems as corporate data received and transmitted by personal devices can be intercepted by other malicious apps on the mobile device or staying in memory for a long time. This thesis presents an ephemeral classification, where the mobile device only handles data in transit. This thesis further investigates limitations and possibilities surrounding the use of ephemeral data within Android mobile devices.

Acknowledgements

I would like to thank my advisor Professor Randi Karlsen and co-advisor Federico Mancini (FFI) for always being positive and being supportive of me.

Thanks also go out to the faculty adviser Jan Fuglesteg for always being available and helpful with his advice.

I would especially like to thank my family for all their support throughout the years.

Contents

List of Figures

List of abbreviations

ADB	Android Debug Bridge
AP	Access Point
BYOD	Bring Your Own Device
DMD	Droid Memory Dumper
GID	Group Identifier
IDE	Integrated development environment
IPC	Inter Process Communication
LKM	Loadable Kernel Module
NDK	Native Development Kit
SDK	Software Development Kit
TEE	Trusted Execution Environment
UID	Unique Identification Number
VM	Virtual Machine

1 Introduction

1.1 Background

As mobile devices continue to be an integral part of our daily lives, they are being more and more used as a mobile technological platform in the workplace. Many enterprises encourage their employees to utilize their own mobile devices as a part of their work environment, instituting a "Bring Your Own Device" (BYOD) practice. This allows the enterprise to cut costs while potentially improving productivity and improving client service. The downside of this is that the enterprise has to have support for a large variation of different mobile devices and the variation of different operating systems which may reside on these devices. This means that building enterprise specific applications becomes not only very expensive, but also very hard to keep secure.

Corporate data received and transmitted by personal devices can be intercepted by other malicious apps on the mobile device or the data can stay in memory for a long time, so that if the device is lost or stolen, this data may be compromised. Properly implemented and deployed encryption schemes and MDM solutions can help protecting data at rest and in transit, but it usually concerns only data written in persistent flash memory or ready to be sent off the device. Much sensitive data is also kept in RAM during processing, and may stay there until overwritten by new data. A recent cold boot attack for mobile devices has been demonstrated that could be used to extract such data in a targeted attack[1].

A possible solution to this is utilizing ephemeral data. Ephemeral data can be described as ‘short lived’ data where the data is erased immediately after use. This is interesting in a mobile environment because if achieved properly you can dismiss a lot of potential security challenges.

In the first quarter of 2015, the International Data Corporations published data from their Worldwide Quarterly Mobile Phone Tracker[2] showing that Android dominated with a 78% share of the market in front of iOS at 18.3% and Windows Phone at a mere 2.7%. While this is a slight dip from the previous year where Android had 81.2%, the overall market is still growing with android making new appearances on novel devices such as smart watches[3], TVs[4], and cars[5].

For this reason we chose to focus our efforts on analyzing and implementing a possible solution for Android OS, as it would affect the majority of mobile devices on the market today. Besides it is also easier to work with it due to its open and extensible nature.

1.2 Problem Definition

This thesis explores the limitations and possibilities surrounding the use of ephemeral data within an Android environment. This includes investigating how the Android platform functions as a whole, and the technical aspects which bound each layer of the Android stack. This research is intended to be a guiding point for developers that want to make user privileged applications with extra focus on security.
Therefore the working hypothesis is that:
“It is possible to create an Android framework that can give developers the option of creating secure applications that only use ephemeral data.”

1.3 Methodology

In order to design the component that will perform the wiping of the application memory, we will proceed by creating several test application that writes sensitive data in memory, and set up our development environment to collect live RAM dumps in order to analyze the effect of different wiping strategies.
An important objective will be to show the numerous ways how this secure deletion of data may fail as data retention may occur as a result of code in the entire Android software stack, rather than just the application dedicated memory.
We will then develop and test our wiping modules/functionalities at different layers of the Android system, and see how they function on the test applications.
An important point is that we limit the scope of this thesis only to the layers that can be accessed without needing root access to the device or a modified kernel. The reason is that we would like to understand how far we can go by using out-of-the-box devices, which would constitute the standard reference device for most application. The assumption that the OS integrity is also not compromised is also necessary to formulate any security guarantee of our proposed solution.
Finally we will analyze the results of our tests and conclude to what extent we believe data ephemerality can be achieved on out-of-the-box Android devices.

1.4 Motivation

This thesis can be considered to belong within a sub arch of digital forensics, and although digital forensics traditionally has been focused on non-volatile data such as hard drives and removable media, in recent years cold boot attacks have opened up research on a new field named live forensics which handles volatile data collected from running machines.
Further research has been done with regards to live forensics in the context of mobile devices, but never in the context of having an ephemeral classification on an application.
With Android being the largest mobile platform on the market, understanding possibilities and limitations within this platform is a requirement for future work.

1.5 Context

This thesis can is done in collaboration with the Norwegian Defence Research Establishment[6]. The Norwegian Defence Research Establishment is the chief adviser on defence-related science and technology to the Ministry of Defence and the Norwegian Armed Forces. This thesis aims to serve as a starting point for further research into employing ephemeral classifications to applications on mobile devices.

1.6 Outline

- Chapter 2 Presents some background information related to this thesis
- Chapter 3 Presents related work within Android Forensics
- Chapter 4 Describes assumptions, and requirements as well as an adversarial model
- Chapter 5 Shows a set-by-set guide on how these experiments were set up
- Chapter 6 Goes through implementation and experiments of our testing applications
- Chapter 7 Evaluates the threat assessment in relation to the adversarial model
- Chapter 8 Shows discussion around the implementation and results of our applications
- Chapter 9 Concludes the thesis

2 Background

2.1 Android

Android is a Linux based operating system intended for mobile devices.
The system was the product of Android Inc which was purchased by Google in 2005.
In 2007 the Open Handset Alliance consisting of device or micro electronics manufacturers such as HTC, Samsung, Sony, T-Mobile, Qualcomm and Texas instruments announced their plans for developing open standards for mobile devices. [7]
Android was their first product as a mobile device platform built on the Linux 2.6.25 Kernel. The Android platform is entirely open source and is created for a wide array of devices and different form factors

The Android software stack

The Android software stack consists of five layers: the Linux kernel, a hardware abstraction layer, libraries and the Android runtime, an application framework, and applications at the top.

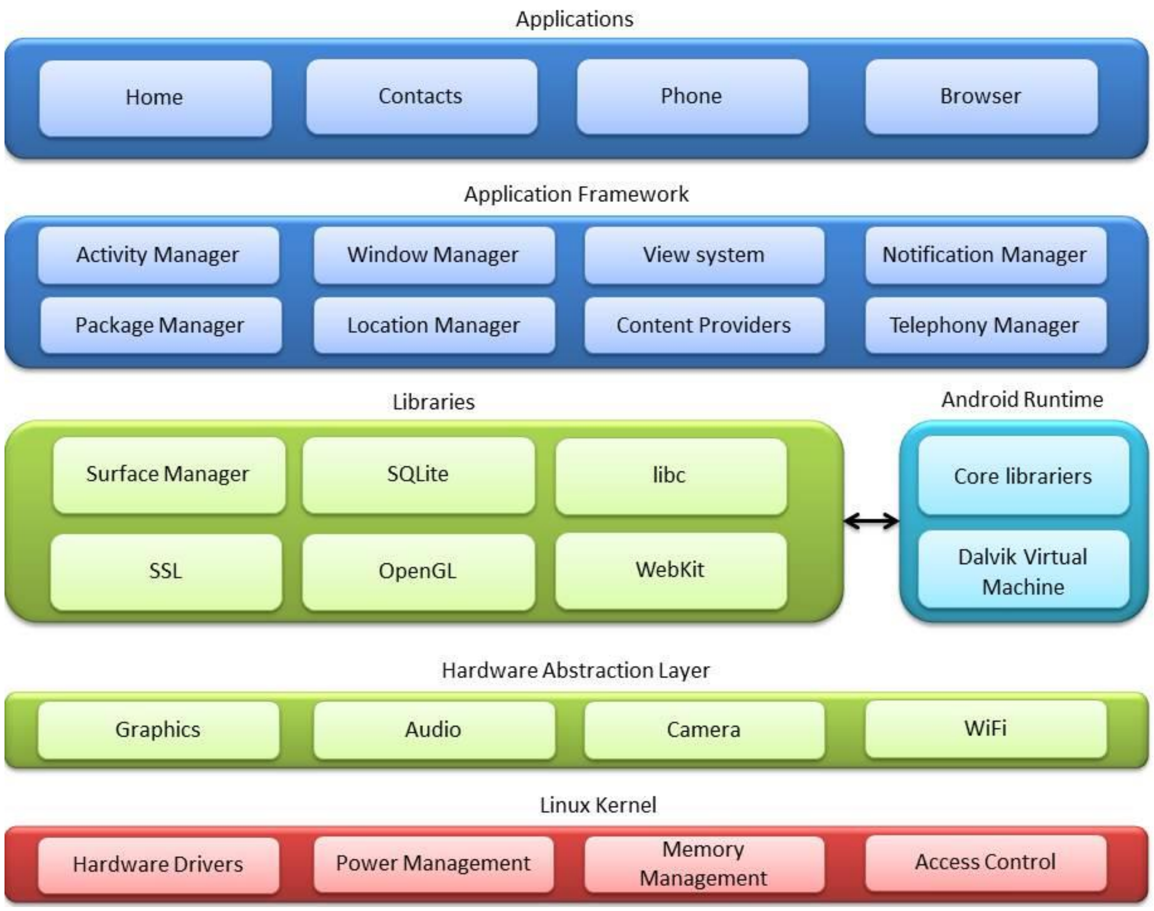

Figure 1: Android software stack

At the bottom of the stack we find a modified Linux kernel which is responsible for all of the basic services such as process scheduling, memory management, managing the network stack, and handling security such as access control and providing means for network security. The kernel also contains drivers for all the different components on the device such as the display, camera, Bluetooth etc.

The next layer is the hardware abstraction layer and it does exactly what the name suggests, it separates the android platform logic from the hardware as well as from the operating system. By doing this it provides the above layers with interfaces to access the underlying hardware.

At the layer above is the Android runtime and Android native libraries.
The Android runtime is the application runtime environment. Up until the Version 4.4 release (KitKat) the standard runtime environment was called Dalvik.
Dalvik is an open source process virtual machine that is a clean room implementation of a Java virtual machine. Programs written in Java are compiled to bytecode for the Java virtual machine, and then translated in to Dalvik bytecode which is stored in a Dalvik executable file (.dex). At Android version 4.4, google introduced an alternative runtime to Dalvik called ART (Android Runtime). In Android version 5.0 (Lollipop) ART replaced Dalvik entirely.
The Android native libraries are a selection of default libraries which are bundled with Android. Although Android applications are written using Java APIs, the implementations of these APIs are often written using C and C++ which are made available via the Java Native Interface.

The Android Application Framework provides services that are essential to the Android platform. Such services include the Window Manager which manages the top-level window's look and behavior, and the Telephony Manager which keeps track of the state of telephony services, amongst others.

At the top of the stack there are the native packaged applications that come with the Android Platform. This is also where the third-party applications reside.

Android's management of packaged and third party applications
There are four ways that an application can get introduced to an Android device.
An application can be pre-packaged with the device, installed through the official Play store, installed through an unofficial app store, or installed through the Android Debug Bridge. The official Play store is the default way to find and install new applications. It is run by Google and offers a centralized portal to find applications, music, books and more. The Play store uses an in-house automated anti-malware system called Google Bouncer which is there to remove malicious applications uploaded to the marketplace. It is also able to remotely uninstall applications off of devices if an application is found to have hidden malware which gets it by the Bouncer, or receives an update with malware.

Each application in an Android environment runs in its own process with a distinct user/group ID. This means that every application is sandboxed through tried and tested Linux process management which is based on the decades-old, well-understood UNIX security model.

In the Android environment there is a special process called "Zygote" whose job is to be a pre-loaded template for new applications that is to be launched. At boot Zygote launches the very first Dalvik VM and it is pre-loaded with all necessary Java classes and resources. When a new application is launched, Zygote is forked so that it creates a clone of itself. Once Zygote has forked, the new process is loaded with the code of the launching application and it is ready to start its activities. This is done to make the process of creating new VMs more resource efficient, but the real speedup is achieved by not copying the shared libraries. These libraries will only be copied if the new process tries to modify them, and this means that all of the core libraries can exist in one place because they are read only.

As part of the hardware abstraction model, when an application wants to make use of hardware modules such as Wi-Fi or the camera on the device it does not try to directly operate the hardware itself, instead it uses services or managers to handle such requests. These services and managers reside inside of the system server process and are user space shared libraries that have predefined APIs which creates an abstraction from the underlying driver. This allows applications to be somewhat hardware agnostic, allowing the same code to run on different devices.

Androids security architecture is based on privilege-separation, in which each application runs with a distinct system entity (Linux user ID and group ID). Privileged parts of the system are also separated into distinct identities and are thus isolated from applications. To allow applications to interact with device hardware modules Android applications has permissions associated with it. By default an application has no permissions, but in accordance with the developers manifest the application requests specific fine grained permissions from the user at install time. This happens at install time or when you update an application so that the user does not have to get prompted every time it is launched. In a normal user situation where a user is installing an application from the Google Play Store, the application cannot be installed if the user does not grant the application its requested permissions. In any other cases trying to use a feature that has not been granted permission to would result in a SecurityException.

When an application wants to communicate with the system server, it does so through a local manager which is a client side component that runs inside the VM, this manager bridges the communication with the system server through Binder, which is a lightweight procedure call mechanism that handles IPC. Binder pulls the UID and PID of the calling process and hands it to the system server which then uses these IDs to check permissions. Other Android permissions directly map to group IDs, which are then enforced by the kernel. An example of this is the permission to use the internet where the process ID is added to the *inet* group which is mapped to the group ID '3003'.

2.2 Ephemeral Data

Ephemerality is the concept of an object being transitory; existing only briefly.
In this context, ephemerality means that a specific data item should cease to exist within an environment and have no duplicates which can be retrieved past the specific data's expiry point.

2.3 Secure data deletion

Secure data deletion is a term which is often also called data erasure, data clearing, or data wiping. Secure data deletion in this thesis is referring to a software-based method of completely deleting data from physical media such that it is irrecoverable. In the digital world, the default protocol when deleting a file is not secure deletion, but rather to remove pointers to the object and directly or indirectly marking it as free or available storage space. This is largely because to the average user *deleting* a file is often done just to reclaim the storage space instead of ensuring that its contents does not get leaked or stolen. Another big factor is the fact that secure data deletion is costly in terms of resources as it requires additional operations compared to just removing pointers.
This mode of operation causes a lot "deleted" data to reside inside of physical mediums although they are not being referenced by anything anymore.

Reliable secure data deletion involves understanding the memory management of the concerning software and also host device to a degree that you know for a fact where all instances of the data resides in memory. Linux memory management is a complex system which makes reliable secure data deletion very hard to get right.
Some prominent figures have this to say on the matter:

"Note that memory usage on modern operating systems like Linux is an extremely complicated and difficult to understand area. In fact the chances of you actually correctly interpreting whatever numbers you get is extremely low. (Pretty much every time I look at memory usage numbers with other engineers, there is always a long discussion about what they actually mean that only results in a vague conclusion.)"
- Dianne Hackborn[8]

"Imagine you have an object inside a cardboard box, and you can't open the box. All you have are holes in the side which allows you to peek inside and get an approximate view of where things are."
- Karim Yaghmour[9]

2.4 Digital forensics

The term *digital forensics* was defined at the first Digital Forensics Research Workshop (DFRWS) in 2001 as: *the use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation, and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations*[10].
When collecting digital forensic evidence there are guidelines on what to do when and what to avoid. Some of the best current practice is listed in RFC 3227[11].
We can separate digital forensics in to two parts, traditional digital forensics which targets persistent storage mediums such as hard drives, NAND flash memory, SD cards, and other removable media.
The newer field of digital forensics is what is called *live forensics* where you target volatile data on a running machine. In other words, data that may be lost by powering down the machine.

One of the most important things to consider when doing live forensics is remembering the order of volatility which says that when collecting evidence, one should proceed from the volatile to the less volatile. An example of order of volatility for a typical system could be:

1. Registers, cache
2. Routing table, ARP cache, process table, kernel statistics, memory
3. Temporary file systems
4. Hard drives
5. remote logging and monitoring data that is relevant to the system in question
6. Physical configuration, network topology
7. Archival media

2.5 Linux Memory Extractor (LiME)

LiME[12], previously known as Droid memory dumper (DMD), is a module created by Joe Sylve that allows for forensically secure acquisition of memory from Linux devices. Since its release in 2011, it has become the de facto standard of acquiring memory dumps with minimal memory perturbation.

2.6 Volatility

The Volatility framework[13] is a cohesive collection of modules written in python that parses and analyzes RAM dumps and has an extensible and scriptable API. It can analyze RAM dumps from Windows, Mac, Linux, and Android devices, and because of its modular design it allows for quick support as new operating systems and architectures are released. Continually adding support for new features, volatility is an open source framework that has become the market leader for forensic memory analysis.

3 Related work

This chapter is split in to two sections. First we present the some of the milestones within Android memory forensics and how they relate to the work being done in this thesis. In the second section we look at different methods of memory acquisition from Android devices, with their potential and limitations.

3.1 Android Memory Forensics

Traditionally the forensic research done on Android devices were focused on the acquisition and analysis of non-volatile storage media such as internal flash NAND memory or SD cards. This was largely due to the perception that the contents of volatile memory, as long as it was properly secured through software sandboxing and privilege enforcement, were unobtainable because it was erased immediately when the host machine lost its power. In 2008, Halderman et al. [14] showed the world how one could leverage temperature to exploit the *remanence effect* of DRAM[15]. By doing this, Halderman was able to recover encryption keys from DRAM with which he defeated full disk encryption schemes. The *remanence effect* states that the contents in RAM gradually fade over time, and Halderman et al. showed a correlation between temperature and degradation over time, meaning the colder you get the RAM chips, the longer it takes for data to fade. This allowed them to boot the machine from an external USB device which saved the contents of RAM in to a designated data partition on the device.

In 2011 Sylve[12] presented DMD(later LiME) a new method that is able obtain forensically secure complete memory captures from rooted Android devices.
Their module works by pushing a loadable kernel module(LKM) on to the device via the Android Debug Bridge. This kernel module acquires a copy of system RAM through parsing the kernel's *iomem_resource* structure to learn the physical memory address ranges of the system RAM, and then performs physical to virtual address translation for each page in memory whilst writing them to either a file on the device's SD card, or piping them through a TCP socket.

In 2012 Müller, et al[1] revealed that cold boot attacks could also be done to Android devices. To do this they cooled down the device to -10 degree Celsius before they rebooted it by disconnecting its battery very briefly. They then utilized a special key combination upon reboot that sends you in to a menu called fastboot instead of booting in to Android. From there their specially developed FROST recovery image was flashed on to the device via USB. Once the recovery image was transferred, all they had to do was boot up into the recovery partition where they placed the image. Once FROST has booted it can use the LiME module inside to dump the memory image via TCP over USB to a host computer.

In 2013 Stirparo, et al[16] wrote about how mobile applications manage users data when these are loaded into volatile memory. To do this they utilized LiME and volatility to analyze popular banking applications and find out how much sensitive information they could find from *data in use*. This paper was a part of the MobiLeak project.

3.2 Memory acquisition methods

There are different methods to acquiring memory which acquire different amounts of memory. On the smaller scale you have dumping the heap of a single process, going up a bit you can dump the entire memory space of a single process, or lastly you can dump the entire memory to a file.
To dump the heap of a single process you can use the DDMS-Tool which is located in the tools-directory of the Android SDK[17]. This gives you the heap of a specific process such that you can use a heap profiling tool (i.e. MAT) to analyze that process' memory profile.

To dump the memory of a single process you can utilize the NDK together with the '*dumpsys'* command executed over ADB. This is an approach that has been automated in to a library[18], but unless handled correctly it is prone to segmentation faults.

For capturing the memory of an entire system there are modules such as *fmem* which does memory dumps from Linux environments. Seeing as Android uses a modified Linux kernel one would think fmem would be a suitable candidate to do the job.
Sylve[11] however found that these modules relied on functions not found on the ARM architecture, and he also discovered that fmem only recovers 80% of the original memory of the device. His findings were that the high percentage of overwritten memory (20%) was likely due to the fact that fmem requires extensive interaction with userland which each time requires a context switch when swapping from kernelland.

Sylves solution was therefore to design his own module named DMD for Droid Memory Dumper, which was later renamed to Linux Memory Extrator (LiME).
One of the main advantages of LiME is that it runs almost all of its instructions in kernelland which minimizes memory perturbing due to context switches.
This is the module we use to obtain our memory dumps.

One problem faced with all LKMs is that the module is device-dependent, meaning that you cannot create a generic kernel module that is device-agnostic.
The kernel performs a number of sanity checks on the kernel module to ensure that the module was compiled for that specific kernel version.
The result is the need to cross-compile the kernel module up against the specific kernel version you wish to extract memory from.

4 Assumptions, requirements and adversarial model

For the remainder of this thesis, we conceptualize a system where we have a Non-rooted mobile device running Android 4.0.3 Ice Cream Sandwich, and a secured server at an unspecified location ready to receive requests. The application is only allowed to have user privileges. We assume the server is safe from physical attacks.
Communication between these two entities is assumed to be done across normal internet where the mobile device is attached to an access point such as a Wi-Fi router which is connected to the internet via landline, or directly communicating with a cellular network provider's cell tower over GSM, UMTS, or LTE frequencies.

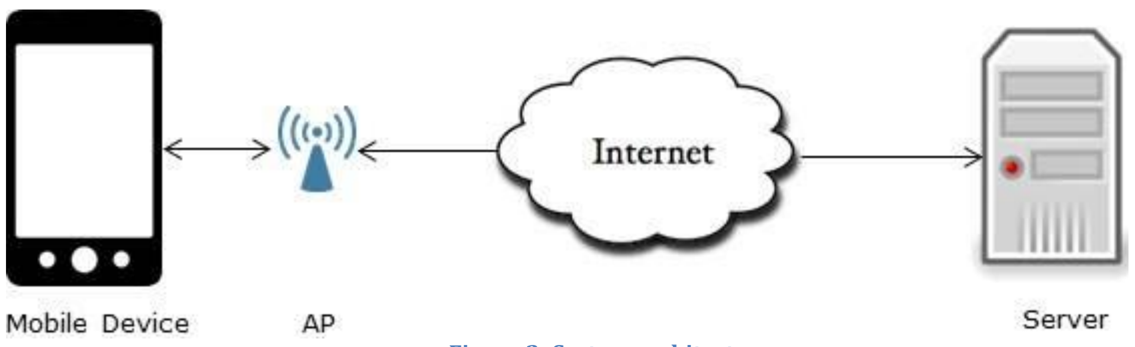

Figure 2: System architecture

4.1 Communication protocol

Focusing on the communication between the mobile device (MD) and the server, we assume the following protocol to be the communication standard when initializing a transaction.
We can section this into five parts:
1. Establishing a secure connection to the
2. Authenticate session with credentials
3. Request data
4. Deliver edited data
5. Terminate session

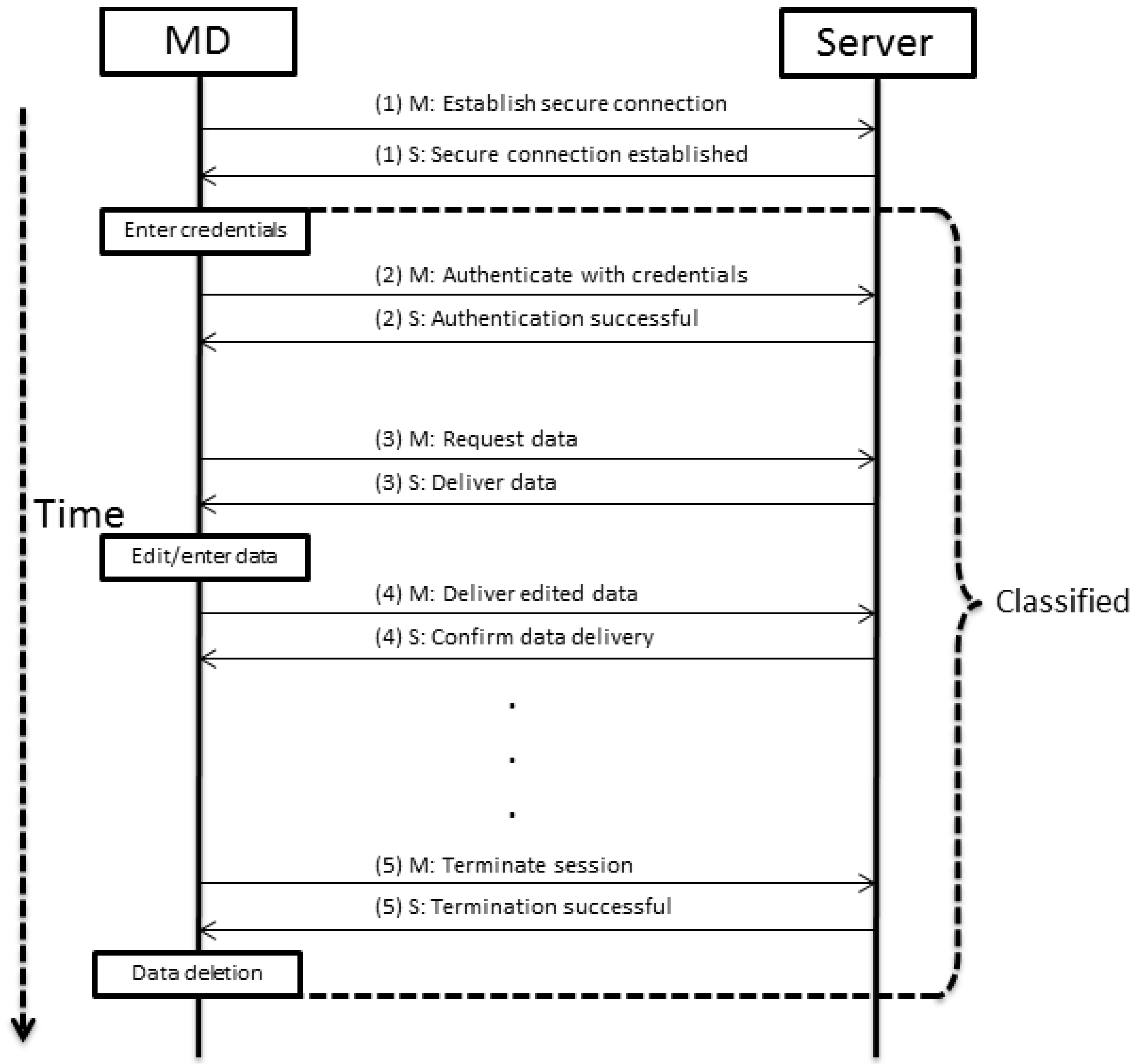
MD
Server
(1) M: Establish secure connection
(1) S: Secure connection established
Enter credentials
(2) M: Authenticate with credentials
(2) S: Authentication successful
(3) M: Request data
(3) S: Deliver data
Time
Edit/enter data
(4) M: Deliver edited data
(4) S: Confirm data delivery
(5) M: Terminate session
(5) S: Termination successful
Data deletion
Classified

Figure 3: Communication protocol between mobile device and server

4.2 Threat model

Based on this communication protocol we can establish a realistic adversarial model that takes into consideration security threats that affect mobile devices differently from other equipment.

Security objectives for this system can into the following categories:

- Confidentiality – Ensure that entered, transmitted, and temporarily stored data cannot be read by unauthorized parties.

- Integrity – Detect physical or software modifications to the mobile device, Detect changes to transmitted and stored data

- Availability – Ensuring that the resources of the system are available and that users can access them

We will now describe our perceived security threats based on this model

Loss of confidentiality:

1. The mobile device is stolen during a transaction and data present in the unit is compromised.
2. Unauthorized personnel physically sees authorized personnel enter in username/password or data.
3. Intruder uses software exploits to gain unauthorized access to device.
4. Another third party application with permissions on the device is allowed to spy on data entered on the mobile device.

Loss of Integrity:

1. The mobile device is physically attacked and has its software modified.
2. The mobile device is used with stolen credentials in order to enter false data to the server.
3. An attacker uses software exploits to enter malware into the mobile device.
4. Man in the middle attack where attacker is pretending to be the server.

Loss of availability:

1. Denial of service attack to the server.
2. The mobile device has its internet connection blocked.

4.3 Requirements

Security requirements
The system should strive to meet the following security requirements:

- No sensitive data should be stored outside of a transaction.
- The security measures of the device should not dependent on any external sources.
- Strong full disk encryption with a strong lock screen password should be used whenever possible.
- If the lock screen button is pushed whilst in an transaction, the device initiates emergency data erasure so the device can return to a non-classified state.
- Information that is to be erased on the device has to be overwritten in memory, even in the case of power failure or system restart.
- The device cannot be rooted.
- The bootloader has to be locked.

Functional requirements
The system should have the following functional requirements:

- The application should have text forms with normal editing functionality present. In addition a submit/cancel button pair should be apparent.
- The application should automatically connect to the correct server.
- The device should give feedback to the user if a transaction succeeds or fails.
- The application should have a sign out button where when pressed the session with the server is terminated and all data erased from memory.

4.4 Our approximation of this system

Creating this system with all of its aspects is outside the scope of this thesis as it is only a 30 point thesis running the course of one semester opposed to the standard 60 point thesis that run the course of two semesters.
We use this system as a reference to what kind of environment our implementations are meant to function in.

For our implementations we have the following functionality:
Have two buttons marked “Create” and “Delete”.
With pushing the “Create” button, display a known text on the screen for the user.
With pushing the “Delete” button, securely delete all known instances of said text.

This is done through locally creating the strings, and utilizing different methods of deletion to destroy the data.
Through using known unique data in the strings, we are able to effectively search through the ram dumps for the data.

5 Expermental setup

In this chapter we present in detail how we were able to successfully set up an environment where we are able to capture live ram dumps while using the emulator bundled with the SDK. These ram dumps are used in chapter 6 for analysis.

5.1 Setting up the environment

In this section we hope to give a step by step guide on how to set up a functional emulator running a 4.0.3 Ice Cream Sandwich image which we collected the memory dumps from.

Hardware: Mac mini (2011) with Dual core Intel i5 2.3 GHz, and 8GB 1333MHz DDR3 SDRAM.

Operating system: Linux Mint 17.1 Rebecca – Cinnamon 64bit.

The first thing we did was to install the Java JDK, which in our case was 1.8.0_45.

We had the openjdk java version prepackaged with mint which we needed to purge it from our system.

```
$ sudo apt-get purge openjdk-\*
##This command will completely remove OpenJDK/JRE from the system

$ sudo mkdir -p /usr/local/java
##This command will create a directory to hold the Oracle Java JDK binaries
```

Download the latest compressed(tar.gz) binaries for your system architecture from Oracles Java page[19].

```
$ cd /home/"your_user_name"/Downloads
$ sudo cp -r jdk-8u45-linux-x64.tar.gz /usr/local/java/
$ cd /usr/local/java
$ sudo tar xvzf jdk-8u45-linux-x64.tar.gz
## By this point we should have the binaries inside /usr/local/java


$ sudo nano /etc/profile
## Next we need to edit the system PATH file /etc/profile and add the
following system variables to the system path:

JAVA_HOME=/usr/local/java/jdk1.8.0_45
PATH=$PATH:$HOME/bin:$JAVA_HOME/bin
export JAVA_HOME
export PATH

##Save the /etc/profile file and exit.


$ sudo update-alternatives --install "/usr/bin/java" "java"
"/usr/local/java/jdk1.8.0_45/bin/java" 1

$sudo update-alternatives --install "/usr/bin/javac" "javac"
"/usr/local/java/jdk1.8.0_45/bin/javac" 1

$ sudo update-alternatives --install "/usr/bin/javaws" "javaws"
"/usr/local/java/jdk1.8.0_45/bin/javaws" 1

## These commands will tell the system that the new Oracle Java version is
available for use.

$ sudo update-alternatives --set java /usr/local/java/jdk1.8.0_45/bin/java

$ sudo update-alternatives --set javac
/usr/local/java/jdk1.8.0_45/bin/javac

$ sudo update-alternatives --set javaws
/usr/local/java/jdk1.8.0_45/bin/javaws

##These commands will inform your system that Oracle Java JDK/JRE must be
the default Java

$ source /etc/profile
## Reload the system wide PATH /etc/profile
```

```
$ java -version
## This command displays the version of java running on the system
## If done correctly, something like this should be displayed:



java version "1.8.0_45"

Java(TM) SE Runtime Environment (build 1.8.0_45-b26)

Java HotSpot(TM) Server VM (build 25.20-b23, mixed mode)

##After a reboot, the system will be fully configured for running java.
```

After setting up java, we need to set up Android studio[20] and the NDK packages[21].
First we install the full Android studio package which includes an IDE.
Once you have installed Android studio, make sure to run it once to complete the installation.
Next we download the NDK package. To execute this package we need to make the file executable.

```
$ chmod a+x android-ndk-r10e-darwin-x86_64.bin

$ ./android-ndk-r10e-darwin-x86_64.bin
##The folder containing the NDK extracts itself.
```

Next, we need to update some packages

```
$ sudo apt-get install bison g++-multilib git gperf libxml2-utils make
zlib1g-dev:i386 zip
```

The Android source tree is located in a Git repository hosted by Google.
We'll refer you to the Android source site[22] as we did exactly as instructed there.
It should be noted that we checked out the master branch which is somewhere around 35GB of data to download. It should be possible to get away with specifying the 4.0.3 branch alone to avoid downloading the source code of branches that might never be used.

After downloading the source code we need to initialize the environment with the *envsetup.sh* script. And then build the code

```
$ source build/envsetup.sh
$ launch full-eng
```

At this point we should have our developer environment fully set up.

Next, we download the Android Kernel Source Code.
Because we use the Android Emulator, we are downloading the Goldfish Kernel as shown below:

```
$ git clone https://android.googlesource.com/kernel/goldfish.git ~/source
Cloning into '/Users/erlend/source'...
remote: Total 2442118 (delta 2048282), reused 2442118 (delta 2048282)
Receiving objects: 100% (2442118/2442118), 501.84 MiB | 465 KiB/s, done.
Resolving deltas: 100% (2048284/2048284), done.

$ cd ~/source/

$ git branch –a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/android-goldfish-2.6.29
  remotes/origin/android-goldfish-3.4
  remotes/origin/linux-goldfish-3.0-wip
  remotes/origin/master

$ git checkout -t remotes/origin/android-goldfish-2.6.29 -b goldfish
Checking out files: 100% (26821/26821), done.
Branch goldfish set up to track remote branch android-goldfish-2.6.29 from
origin.
Switched to a new branch 'goldfish'
```

At this point we create an AVD inside of Android Studios AVD manager.
For our experiments we found that we had to run it on a Nexus 4 – 4.7" 768x 1280 xhdpi avd. When we tried creating an AVD running on nexus 5 phone with 1080 x 1920 xxhdpi the avd would not boot with our image. Make sure to download and set the system image to API 15 for 4.0.3 Ice Cream Sandwich using armeabi-v7a.

At this point we start the new AVD. Before we can build the Goldfish Kernel we need to extract a working config from the running AVD.
Once the AVD is up and running:

```
$ cd ~/android-sdk/platform-tools
$ ./adb pull /proc/config.gz
```

Once we have the config.gz file, we decompress it, and copy it into the Goldfish source directory as ".config".

```
$ tar -xvzf config.tar.gz
$ cp config ~/source/.config
```

Next we need to enable loadable kernel modules in the .config file.
We do this simply by adding the following lines:

```
CONFIG_MODULES=y
CONFIG_MODULES_UNLOAD=y
CONFIG_MODULES_FORCE_UNLOAD=y
```

At this point we need to make sure that all the tools and the cross-compilation tool chain is in our path. Below is an excerpt of our .bashrc file.

```
export USE_CCACHE=1
export PATH=$PATH:~/Android/Sdk/tools/
export PATH=$PATH:~/Android/Sdk/platform-tools/
export ANDROID_SWT=~/Android/Sdk/tools/lib/x86_64/
export CCOMPILER=~/android-ndk-r10d/toolchains/arm-linux-androideabi-
4.6/prebuilt/linux-x86_64/bin/arm-linux-androideabi-
```

At this point, we should be able to switch to the Goldfish source folder and compile the kernel.
A little word of warning on this: This Goldfish Makefile is recursive. We had a bit of trouble compiling this without error and sometimes it would compile but would ignore the file writeout so that we were left without an Image. What we found was that it was easiest to back up the untouched Goldfish source folder to have a clean slate to work with in case the compilation did not work.

Inside the Goldfish source folder type:

```
$ make ARCH=arm SUBARCH=arm CROSS_COMPILE=$CCOMPILER EXTRA_CFLAGS=-fno-pic
modules_prepare
```

If the code compiled successfully, we will have a new image at

```
~/goldfish-source/arch/arm/boot/zImage
```

If that does not compile, we would suggest removing the extra flags of *–fno-pic* and *modules_prepare*
We had to play around with the them to make it compile correctly.

At this point we should be able to emulate the newly compiled kernel with the AVD we created earlier.

```
$ ~/Android/Sdk/tools/emulator -avd myAvd -kernel ~/android-
source/arch/arm/boot/zImage -show-kernel -verbose
```

5.2 Setting up LiME

The next step in our journey is downloading and cross compiling LiME with our kernel.
We use SVN to check out the LiME trunk:

```
$ svn checkout http://lime-forensics.googlecode.com/svn/trunk/ ~/lime-
forensics
```

Inside of this folder we will find a sample Makefile (Makefile.sample) which we can use as a template to create our own. Our Makefile is shown below.

```
obj-m := lime.o
lime-objs := tcp.o disk.o main.o

KDIR_GOLD := ~/source
KVER := $(shell uname -r)
PWD := $(shell pwd)

CCPATH := ~/android-ndk-r10d/toolchains/arm-linux-androideabi-
4.6/prebuilt/linux-x86_64/bin/

default:
	# cross-compile for Android emulator
	$(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-linux-androideabi- -C
$(KDIR_GOLD) EXTRA_CFLAGS=-fno-pic M=$(PWD) modules

	$(CCPATH)/arm-linux-androideabi-strip --strip-unneeded lime.ko
	mv lime.ko lime-goldfish.ko

	# compile for local system
	$(MAKE) -C /lib/modules/$(KVER)/build M=$(PWD) modules
	strip --strip-unneeded lime.ko
	mv lime.ko lime-$(KVER).ko

	$(MAKE) tidy

tidy:
        rm -f *.o *.mod.c Module.symvers Module.markers modules.order
\.*.o.cmd \.*.ko.cmd \.*.o.d
        rm -rf \.tmp_versions

clean:
        $(MAKE) tidy
        rm -f *.ko
```

Once we have compiled the code we get two output files. The one we are interested in should be called “lime-goldfish.ko”.
To load the LiME LKM on to the Android device we use ADB.

```
$ ~/Android/Sdk/platform-tools/
$ ./adb push ~/lime-forensics/src/lime-goldfish.ko /sdcard/lime.ko
```

To extract the ram dump from the device we can either dump it to the SD-card then copy it over, or pipe it directly to the pc via forwarded TCP.
We choose to pipe the memory dump via TCP.

```
$ adb forward tcp:4444 tcp:4444
$ adb shell
$ su
# insmod /sdcard/lime.ko "path=tcp:4444 format=lime"
```

Now on the PC, we can establish the connection and acquire the memory using netcat.

```
$ nc localhost 4444 > ram.lime
```

If all went well, we should have a good memory dump named “ram.lime” to analyze.

5.3 Setting up Volatility

In order to do memory analysis we utilize the Volatility framework.
To install and set it up correctly we first install Dwarfdump[23] which is an application used to print DWARF debug information in a human readable form.

```
$ sudo apt-get install dwarfdump
```

Once we have Dwarfdump installed we can install the Volatility framework.

```
$ svn checkout https://volatility.googlecode.com/svn/trunk/ ~/android-
volatility
$ cd ~android-volatility/tools/linux
```

To start using the Volatility framework we need to correctly build a volatility profile.
We must edit the Makefile inside of the tools/linux directory to correspond with the paths on our system. We have added the Makefile we used to compile our profile below:

```
obj-m += module.o
KDIR ?= ~/source
KVER ?= $(shell uname -r)

CCPATH := ~/android-ndk-r10d/toolchains/arm-linux-androideabi-
4.6/prebuilt/linux-x86_64/bin/

-include version.mk
all: dwarf

dwarf: module.c

        $(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-linux-androideabi- -C
$(KDIR) CONFIG_DEBUG_INFO=y M=$(PWD) modules

        dwarfdump -di module.ko > module.dwarf
clean:
        $(MAKE) -C $(KDIR)/lib/modules/$(KVER)/build M="$(PWD)" clean
        rm -f module.dwarf
```

With this Makefile we can compile the *module.dwarf* driver.
The final step is to combine that module.dwarf file and the *System.map* file from the Goldfish source directory into a zip file. Put that zip file in the volatility/plugins/overlays/linux directory.

```
$ zip ~/android-volatility/volatility/plugins/overlays/linux/Golfish-
2.6.29.zip module.dwarf ~/source/System.map
```

```
  adding: module.dwarf (deflated 90%)
  adding: Users/erlend/source/System.map (deflated 73%)
```

We now have a fully functioning Volatility profile, and with that our testing environment is nearly complete.

In order to install applications on the device we need to compile the code in Android Studio, and then we can install an .apk file via ADB like this:

```
$./adb install ~/AndroidStudioProjects/MemoryTest/app/build/outputs/apk/
app-debug.apk
```

Volatility uses a modular plugin structure where each plugin should be usable as a standalone tool, but they are also able to call on each other so that the results of one plugin can be used for further processing.

A list of plugins can be generated using the –info command.
Below you can see an excerpt from this list with some interesting plugins.

```
$ python vol.py –info

Plugins
-------
linux_dump_map           - Writes selected memory mappings to disk
linux_find_file          - Recovers tmpfs filesystems from memory
linux_lsof               - Lists open files
linux_memmap             - Dumps the memory map for linux tasks
linux_netstat            - Lists open sockets
linux_proc_maps          - Gathers process maps for linux
linux_psaux              - Gathers processes along with full command line
and start time
linux_pslist             - Gather active tasks by walking the task_struct-
>task list
linux_pslist_cache       - Gather tasks from the kmem_cache
linux_pstree             - Shows the parent/child relationship between
processes
```

First we can look at linux_psaux which lists up all the processes along with the full command line from when they were launched. Below is an excerpt from the resulting list.
Let’s make a note the PID of the application bundled com.erlend.memorytest.

```
$ python vol.py --profile=LinuxGoldfish-2_6_29ARM -f ~/ram.lime linux_psaux
Volatility Foundation Volatility Framework 2.3.1
Pid    Uid    Gid    Arguments
1      0      0      /init
339    10011  10011  com.android.providers.calendar
353    10007  10007  android.process.media
373    10017  10017  com.android.exchange
386    10027  10027  com.android.email
407    10028  10028  com.android.mms
488    10003  10003  com.android.defcontainer
504    10023  10023  com.svox.pico
517    10033  10033  com.android.quicksearchbox
536    10040  10040  com.erlend.memorytest
549    0      0      /system/bin/sh -
551    0      0      sh
553    0      0      insmod /sdcard/lime.ko path=tcp:4444 format=lime
```

Let’s take a look at linux_proc_maps. This plugin prints details of process memory, including heaps, stacks and shared libraries. While using this plugin we provide it with a specific process id 536, which corresponds to the *memorytest* application.
Below is a small excerpt detailing some of the data elements that show for each file in the memory there is a start memory address, and an end.

```
$./python vol.py --profile=LinuxGoldfish-2_6_29ARM -f ~/ram.lime
linux_proc_maps -p 536
Volatility Foundation Volatility Framework 2.3.1

Pid      Start              End                File Path

-------- ------------------ ------------------ ------------------------
536 0x000000004ab18000 0x000000004ab20000 /data/app/com.erlend.memorytest-
1.apk

536 0x000000004ab20000 0x000000004ad86000 /data/dalvik-
cache/data@app@com.erlend.memorytest-1.apk@classes.dex


536 0x000000004ad86000 0x000000004ad8c000 /dev/ashmem/InputChannel 419369b8
com.e...est/com.erlend.memorytest.MainActivity
```

Using the linux_dump_map plugin we can extract the memory of every process into distinct .vma files which cover the different memory ranges that each processes’ virtual memory address space consists of. Using this plugin we can also specify a single process to dump from or from a specific memory address within that process.

In order to test the data localization of the strings in the application we needed to do a search of the entire memory dump. Strings[24] is a good candidate as it is useful for determining the contents of non-text files. For each file given, Strings will look through the file and print the printable character sequences that are at least 4 characters long and are followed by an unprintable character.

When we find one or more instances of the data item through using Strings we must attempt to localize them. The first thing we do is to is to run linux_dump_map for only the targeted application, as it’s a good bet that one or more hits will be in there. We then need to run the Strings command on all of the resulting .vma files to know which one, if any, contains the data. Since there are so many files in memory for each process it seemed prudent to automate the process a bit. We therefore created a small bash script to help with the acquisition of strings.

```
#!/bin/bash
        for i in $( ls ); do
            $(strings $i > ./strings/$i.txt)
        done
```

This script executes the strings command on every file in the current directory, creating text document with the results inside for each pass through the loop.
Correspondingly, for each file in memory we get one .vma file and one .txt file.
Now that we have successfully extrapolated the strings from the binary files we can do a grep lookup to see if we have a match in any of the files.

```
$ grep -rnw . -e “ce656850400574e9f9cffb285ee8abc0”
```

If this search is successful it should show us which files, and therefore correspondingly memory area, the data is in.

If we have not accounted for all of the data instances found by our initial Strings scan, we can scope out further and run linux_dump_map without specifying a process to dump from. With this command we get the virtual address space of every process in the linux_psaux list.
If we do the same procedure with the script and then grep, we should now be able to see if the data is contained in any of the other processes virtual memory address space.

6 MemoryTest Implementations

In chapter 4 we detailed our approximation of the conceptualized system.
This chapter describes the design and implementation of the three testing applications we built to investigate the possibilities and limitations surrounding the memory management in Android. These applications are not meant to be thought of as a testing ground for experimentations around data in memory.

6.1 Overview

We have sectioned these tests into three applications:
The first application named “MemoryTest” is fully written in Java and acts as a baseline for the test of the tests.
The second application named “MemoryTestNative” utilizes the Android Native Development Kit (NDK) so that it has one part written in java, and another part written in C.
The third application named “MemoryTestNativeBuffer” also utilizes the NDK with one part written in java, and the other written in C, but we use different means of communication between the two parts.

The purpose of these applications is to present the different aspects within the Android platform that hinders the concept of data ephemerality through secure deletion.

As detailed in chapter 4, the function of these applications are very simple.

1. Through pressing a button named "Create", display a string on screen.
2. By pressing the button named "Delete", delete all known instances of said string.

In order to verify the destruction of the data it is important that the string is both known and unique. It is important that the string is known because we will search through extracted ram dumps for its signature. It is also important that it is unique because we want the results from these searches to be as accurate as possible, and if the content of the string is not unique, it may lead to false positive results. For these reasons we landed on using a the hash value "ce656850400574e9f9cffb285ee8abc0", which is the md5 hash of "secretKey".

The reason we use strings as the selected form of data is because it is both easier to verify the results of our tests, and it is easier to create a unique piece of data that is searchable.

The procedure for collecting the memory dumps of these applications were as described in chapter 5. For the "MemoryTest" application that was fully written in Java, we performed only one memory dump. This is because we wanted this application to serve as a baseline for the other tests and it does not have a "Destroy" function. As such the memory dump was performed while data was intact and on screen. For the two other applications we performed two tests. The first test involved dumping memory while data was intact and on screen, and the other test after we had pressed the "Delete" button.

Step	Test 1	Test 2
1	Reboot emulator	Reboot emulator
2	Launch the app	Launch the app
3	Press Create button	Press Create button
4	Memory Dump	Press Delete button
5	Finish	Memory Dump
6		Finish

Table 1: Step-by-step procedure for the tests

To ensure we got proper results we deleted the Android Virtual Device (avd) folder in *~/.android/avd* between each test, and replaced it with a clean backup.

6.2 The difference between Java and C in the context of Android

The common approach to writing applications for the Android platform is to write them in Java, and execute them within the virtual machine environment, called Dalvik or ART. Because the Android platform is running on a Linux kernel written in C, you get the option of writing C or C++ code for your application. The way this works is that you have a toolset called the Native Development Kit which allows you to call C or C++ code from the Java application through a Java Native Interface (JNI). The reasons for using this feature is usually performance related, but we use it for enhanced security.

Java is a high-level object oriented programming language; this means that data is abstractly referenced by objects of different types. Being high-level, Java abstracts away from the notion of the developer handling his own memory allocations. This means that you cannot easily reference direct memory addresses.
In stark contrast, C is a low-level procedural programming language that allows for dynamic memory allocation. This dynamic memory allocation allows for blocks of memory of arbitrary size to be requested at run-time from the heap.

Because the Java developer is not tasked with handling is own memory allocation he is not tasked with deleting data either. Java therefore has a garbage collector that attempts to reclaim “garbage”, which is memory occupied by objects that are no longer in use by the program. Android through Dalvik and ART also has this. The traditional garbage collection scheme for Android has been mark-and-sweep, where algorithm consists of two phases: In the first phase, it finds and marks all accessible objects. The first phase is called the mark phase. In the second phase, the garbage collection algorithm scans through the heap and reclaims all the unmarked objects. The second phase is called the sweep phase. If this garbage collection scheme runs multiple times over the course of a long-running program you may end up with what is called fragmentation. This problem occurs when live objects end up being separated by many, small unused memory regions and can cause the application to crash because it could not allocate a big enough memory segment.

In 2014 google announced that they have a compacting garbage collector under development[25], and this will be very influential for developers that want to try the path of secure deletion using Java within Android. Opposed to the traditional mark-and-sweep techniques used by Androids garbage collectors, a compacting garbage collector moves(copies) objects in use in order to avoid fragmentation. This can very quickly lead to unwanted duplicates of sensitive data in memory.

6.3 Implementing the applications

MemoryTest

This application is built to be as simple as possible. We allocate a String object on the heap with the known string value, and pass it on to our TextView via the setText method.

```
String key = new String("ce656850400574e9f9cffb285ee8abc0");
TextView secretKeyTextView = (TextView)findViewById(R.id.secretKey);
secretKeyTextView.setText(key);
```

This displays the hash value on the screen.

MemoryTestNative

This application is written using the NDK to link C code together with Java code.
As such we have created our on C-library called “nativeLib” which we use to provide low-level data allocation and secure deletion of data.

As the application is launched, the java segment starts by creating the linkage to the C segment. The Java segment controls the graphical parts of the application and is responsible for the two buttons “Create” and “Delete” as well as the TextViews.

The methods that triggers when either of the “Create” and “destroy” buttons are pressed further calls down to different native JNI methods.

```
public void createObjects(View v) {

        TextView fullNameTextView = (TextView)findViewById(R.id.fullName);
        TextView secretKeyTextView =
(TextView)findViewById(R.id.secretKey);

        fullNameTextView.setText(getfullName());
        secretKeyTextView.setText(getsecretKey());

}
```

This Java method calls upon two different JNI methods, *getfullName()*, and *getsecretKey()*. The strings we search for in the memory dump now gets declared in C as volatile char arrays. The volatile keyword acts as a data type qualifier and alters the way the compiler handles the variable so that it does not attempt to optimize the storage of it.

```
volatile char fullName[] = "SECRET_NAME";
volatile char key[] = "ce656850400574e9f9cffb285ee8abc0";
```

The two C functions that correspond to the two JNI methods that gets invoked when the “Create” button is called are very simple. They each return a built in JNI function called NewStringUTF.

```
JNIEXPORT jstring JNICALL
Java_com_erlend_memorytestnative_MemtestActivity_getsecretKey (JNIEnv *env,
jobject obj){
    return (*env)->NewStringUTF(env, key);
}
```

What this JNI function does is to convert the contents of the C char array into a jstring, which corresponds to a normal Java String. The resulting string is then displayed in the TextView. When the “Delete” button is pressed, the C function of *destroyData* is invoked.

```
JNIEXPORT void JNICALL
Java_com_erlend_memorytestnative_MemtestActivity_destroyData (JNIEnv *env,
jobject obj){

    int fullNameSize = sizeof(fullName);
    int keySize = sizeof(key);

    secure_memset(fullName, 'a', fullNameSize);
    secure_memset(key, 'a', keySize);

    fullNameSize = 0;
    keySize = 0;
    return;

}
```

This function is responsible to call the *secure_memset* function which handles the overwriting of the char arrays.

```
void *secure_memset(void *v, int c, unsigned int n) {
    volatile char *p = v;
    while (n--) *p++ = c;
    return v;
}
```

This is an elegant little function similar to the regular memset which is used to overwrite a memory address range. The reason for using this function instead of memset is that because the data is not used after it is overwritten, there is a chance that certain compilers might optimize the entire overwriting process out. By using the volatile keyword as well as our own memset function, we can be assured that the secure deletion takes place.

MemoryTestNativeBuffer

This application is similar to the other Native application, but relies on a different approach to sending the data through JNI. Within this application we utilize a Direct ByteBuffer to send the data from C to java. A ByteBuffer functions like a view of some underlying storage of bytes. A byte buffer is either direct of non-direct, meaning that with a direct buffer the overlaying virtual machine will make a best effort to perform native I/O operations on it. This means that it will avoid copying the buffers content into an intermediate buffer when handled through underlying operating system native I/O operations. Using this buffer technique should allow the data to be allocated in C, and allow Java methods to fetch the contents of the same memory address with only passing the buffer reference through JNI. This buffer, when allocated in C will be stored on the Native heap, meaning that it is outside of the scope of any garbage collection activities.

```
ByteBuffer b = (ByteBuffer) allocNative(32);}
```

In Java, this fetches the reference to the ByteBuffer.

```
JNIEXPORT jobject JNICALL
Java_com_erlend_memorytestnativeBuffer_MemtestActivity_allocNative(JNIEnv*
env, jlong size){
    byte* buffer = malloc(keySize);
    CharToByte(key, buffer, keySize);
    jobject directBuffer = (*env)->NewDirectByteBuffer(env, buffer,
keySize);
    jobject globalRef = (*env)->NewGlobalRef(env, directBuffer);
    return globalRef;
}
```

This is the corresponding C function which handles the memory allocation and copies the contents of the char array into the buffer. To handle the copy we added a simple *ByteToChar* Function.

```
void ByteToChar(byte* bytes, char* chars, unsigned int count){
    unsigned int i = 0;
    while(i < count)
    {
        chars[i] = (char)bytes[i];
        i++;
    }}
```

Once we have received the buffer reference on the java side, because the char array in C is ASCII encoded we need to decode this to UTF in order to send it on to the TextView.

```
Charset charset = Charset.forName("US-ASCII");
CharsetDecoder decoder = charset.newDecoder();
String str = null;
str = decoder.decode(b).toString();
fullNameTextView.setText(str);
```

Note that this is a suboptimal method for extracting the data from the buffer because like in the “MemoryTest” application we are creating a Java string with the contents of the buffer. Otherwise we should be able to delete the sensitive data by overwriting the char arrays and the contents of the bytebuffer. We were not able to find a successful way of transferring the string data of the bytebuffer to the TextView without storing it in an intermediate buffer or string which effectively creates a duplicate or the data.

6.4 Results

Using the memory acquisition methods discussed in chapter 5, we now present our results after running tests on the built applications.

MemoryTest
As previously stated we did no attempts to destroy the data in this application so the memory acquisition was taken while the data was on screen.
Our findings were that we only found one instance of the given data, and it was located inside the Dalvik cache of our application process.

```
/data/dalvik-cache/data@app@com.erlend.memorytest-1.apk@classes.dex
```

MemoryTestNative
With this application we did two tests: First one where the data was intact and in the view, and the second where the data was deleted both from the original char array and the view.

On the first test we got four hits in the memory dump

Upon further investigation we discovered that two of these hits were located within the System_server process, and one was in the applications own virtual address space.

The system_server process is responsible for starting all system services and managers. Although we were able to find that both hits were in the same memory address range of 0x4ad26000, there was no listing of description for that address.

The hit in the application's own virtual address space was located in the compiled library file for our nativeLib.

```
/data/data/com.erlend.memorytestnative/lib/libnativeLib.so
```

That leaves one instance of the data somewhere in the memory dump that is outside of any processes virtual memory address space.

On the second test where we attempted to securely delete the data we got three hits in the memory dump.

Two of the hits were still in the system_server, but the hit from within the application was not found! We still had one hit outside of known virtual address space.

MemoryTestBuffer
Unfortunately we were unable to test this application due to some bugs within the code. While we were able to run it on the stock emulator that runs with android studio, we were sadly not able to make it run on our custom image.

6.4 Conclusion

In the MemoryTest application we saw a hit point to the dalvik cache .dex file, and in the first test of the MemoryTestNative application we saw a hit point to the compiled custom added native library libnativeLib.so. In neither of the tests had we done any attempted deletion of sensitive data, so it is quite natural that we would get hits in both those places. However, in the second test of MemoryTestNative we did proceed with secure deletion of the local char array in the native library and the tests showed that the data hit was gone. We take this to mean that secure deletion work for that purpose. We only need to figure out how to send the data up to the screen properly.

7 Evaluation of threats

In chapter 4 we defined an adversarial model, and listed potential threats.
In this chapter we describe solutions that may handle the given threats to an acceptable degree.

Loss of confidentiality:

1. **The mobile device is stolen during a transaction and data present in the unit is compromised.** This is the worst case scenario. Without highly advanced methods of detection by the mobile device, data will be compromised. The only thing one can do is to limit the damage as much as possible. To that end we propose doing data deletion at the end of every transaction instead of log-on session; so that we erase the applications memory as soon as it gets the confirmation that the data has been delivered. This way, the amount of potential sensitive data is reduced to its absolute minimum.
2. **Unauthorized personnel physically see authorized personnel enter in username/password or data.** This is an issue that can be prevented if the proper steps are taken. First and foremost the authorized personnel that is to handle the device should be very aware of their surroundings when they start a classified session. Functional measures should be taken to prevent this such as not displaying key presses on the screen for more than a second. Physical measures could be taken by adding a privacy filter to the screen that reduces viewing angle of the display.

3. **Intruder uses software exploits to gain unauthorized access to device.** The chance of this can be lowered by being strict on what applications one allows in the mobile environment as well as always keeping the software up to date. In the case that someone targets a device specifically and has knowledge of software vulnerabilities on it there is not much you can do.
4. **Another third party application with permissions on the device is allowed to spy on data entered on the mobile device.** This threat can be greatly diminished by being critical to what applications you allow on the device. For extreme security measures verified boot and trusted execution environment would be the next step.

Loss of Integrity:

1. **The mobile device is physically attacked and has its software modified.** The only way to prevent this is to have verified boot within a trusted execution environment. For this to work you would need to have a root of trust based in hardware, as software gets easily manipulated.
2. **The mobile device is used with stolen credentials in order to enter false data to the server.** There is not much to do to mitigate this other than look at users access patterns to closely monitor and try to catch irregular behavior.

3. **An attacker uses software exploits to enter malware into the mobile device.** This threat can be diminished by reducing the amount of attack vectors available on the mobile device through strict application installation approval, and frequently updating the software that is on the device. Verified boot and a TEE could help detect such modifications.
4. **Man in the middle attack where attacker is pretending to be the server.** Can be prevented by keeping a copy of the server's public key certificate on the mobile device. If such an attack were to happen, the only data the attacker would receive would be an encrypted challenge.

Loss of availability:

1. **Denial of service attack to the server.** Can be partially mitigated through having good server infrastructure with load balancers and being aware of the load on the network.
2. **The mobile device has its internet connection blocked.** No mitigation available.

8 Discussion and evaluation of results

8.1 OpenGL

Most of our problems with the results involve getting the data to the screen. At some point they have to be temporarily stored as Java strings. An alternative to this is avoiding using the java view all together by opting to use the OpenGL library to draw on the screen directly from C. Using third party libraries like FreeType[26] you can render fonts into a bitmap. We have not explored the full potential of this path but it seems promising.

8.2 Application crash

In the event that the application crashes before a secure erasure of data can be done the memory segment dedicated to the application will still reside in memory but will be marked freed by the garbage collector. This means that sensitive data can still reside within the memory segment until that area of the memory is allocated to another process and the specific memory addresses are overwritten. We see no way to avoid this apart from modifying the kernel to invoke some special procedure should the application crash.

8.3 Data findings outside of virtual address ranges

In our MemoryTestNative application we found a result that pointed to the fact that one of the instances of the sensitive string must lie outside of the virtual address range of any of the running processes. What this means is that we have one instance in memory of the string that are not accounted for by any process. It could be that these additional string reside in physical pages that are no longer allocated so they wouldn't be mapped to any process' virtual address space. This could be a side effect of the kernel's memory manager trying to relocate the pages. Another possibility is that the strings are in pages that are allocated, but in kernel memory. This could happen if we were seeing some of the dex files in the kernel's file system cache.

8.3 Lessons learned by the author

Coming from a background of very little Java experience, having done no Android development, and having very little knowledge about mobile forensics or secure data deletion; this thesis has forced the author to learn a lot in a very short amount of time.
There was an awful lot of debugging involved with setting up LiME and volatility correctly. Java is still a programming language the author is learning to handle.

9 Conclusion

The main goal of this thesis was to illustrate and elaborate on the use of ephemeral data in an Android environment. To do this we illustrated an architecture where the ephemeral data was a lynchpin and exposed potential weaknesses in relation to the mobile platform and how to best overcome them. In addition we showed in great detail how to set up a virtual android device so that memory forensics experiments could be conducted. We also demonstrated how the volatility framework functions, and demonstrated that Linux memory management is a highly complex system.

Bibliography

1. Tilo Muller, M.S., and Felix C. Freiling. . *Frost: Forensic Recovery of Scrambled Telephones*. 2012; http://www1.cs.fau.de/filepool/projects/frost/frost.pdf.
2. International Data Corporations. *Worldwide Quarterly Mobile Phone Tracker*. 2015; http://www.idc.com/tracker/.
3. Google. *Android Wear*. 2015; http://www.android.com/wear/.
4. Google. *Google TV*. 2015; http://www.android.com/tv/.
5. Google. *Google Auto*. 2015; http://www.android.com/auto/.
6. *The Norwegian Defence Research Establishment (FFI)*. [cited 2015; http://www.ffi.no/en/Sider/default.aspx.
7. Open Handset Alliance. *"Industry Leaders Announce Open Platform for Mobile Devices"*. 2007.
8. Hackborn, D. 2013; http://stackoverflow.com/a/2299813/5252645.
9. Yaghmour, K. *Opersys - Memory Management Internals*. 2015; https://www.youtube.com/watch?v=0BLLt_U5pus.
10. DFRWS, *A Road Map for Digital Forensic Research.* 2001: p. 16.
11. The Internet Society, *Guidelines for Evidence Collection and Archiving.* 2002.
12. Sylve, J., *Android Memory Capture and Applications for Security and Privacy.* 2011.
13. Volatility. *The Volatility Framework*. [cited 2015; https://code.google.com/p/volatility/.
14. Halderman, e.a., *We Remember: Cold Boot Attacks on Encryptions Keys*. 2008.
15. Gutmann, P., *Data Remanence in Semiconductor Devices.* 2001.
16. Pasquale Stirparo, I.N.F., Ioannis Kounelis, *Data-in-Use leakages from Android Memory - Test and Analysis.* 2013.
17. Google. *Capturing a Heap Dump*. 2015; https://developer.android.com/tools/debugging/debugging-memory.html#HeapDump.
18. Teoh, P. *How to dump memory of any running processes in Android (rooted)*. 2011; https://tthtlc.wordpress.com/2011/12/10/how-to-dump-memory-of-any-running-processes-in-android-2/.
19. Oracle. *Java SE Downloads*. 2015; http://www.oracle.com/technetwork/java/javase/downloads/index.html.
20. Google. *Android Studio*. 2015; http://developer.android.com/sdk/index.html.
21. Google. *NDK Downloads*. 2015; http://developer.android.com/ndk/downloads/index.html.
22. Google. *Downloading the Source*. 2015; http://source.android.com/source/downloading.html.
23. Dwarf Debugging Format. *Libdwarf And Dwarfdump*. 2014; http://wiki.dwarfstd.org/index.php?title=Libdwarf_And_Dwarfdump.
24. About.com. 2015; http://linux.about.com/library/cmd/blcmdl1_strings.htm.
25. Google. *Compacting garbage collector under development*. 2015; http://developer.android.com/guide/practices/verifying-apps-art.html.
26. *FreeType software library to render fonts*. 2015; http://www.freetype.org/.