Analyzing Application Data Security
on Android Devices

Pallavi Sivakumaran

Technical Report
RHUL–MA–2014– 12
01 September 2014
Analyzing Application Data Security on Android Devices

Pallavi Sivakumaran
Acknowledgements

I would like to thank Professor Keith Mayes for his immense support and encouragement during this project. I would also like to express my sincere appreciation to the rest of the staff at the Royal Holloway ISG, for putting together this MSc programme. Finally, I would like to thank my family and friends, who have been an invaluable source of inspiration to me throughout the years.
# Table of Contents

List of Figures .............................................................................................................. iv  
List of Tables ................................................................................................................ v 
Abbreviations and Acronyms ..................................................................................... vi 
Executive Summary .................................................................................................... vii 
1 Introduction .................................................................................................................. 1 
  1.1 Project Background ............................................................................................... 1 
  1.2 Project Objectives ............................................................................................... 2 
  1.3 Methodology ....................................................................................................... 2 
  1.4 Structure of Report ............................................................................................. 3 
2 The Android Architecture and Security Mechanisms .............................................. 4 
  2.1 Architecture of the Android Platform .................................................................. 4 
  2.2 The Android Security Model ............................................................................. 5 
    2.2.1 Access Permissions: Linux vs. Android ....................................................... 5 
    2.2.2 An In-depth Look at Application Sandboxing and Resource Access .......... 5 
    2.2.3 The Concept of a Root User ....................................................................... 7 
    2.2.4 Process Credentials and Prevention of Privilege Escalation ....................... 7 
    2.2.5 The ADB Shell and Root Privileges ............................................................ 8 
  2.3 Rooting an Android Device .................................................................................. 9 
    2.3.1 Possible Reasons for Rooting ....................................................................... 9 
    2.3.2 Gaining Root Privileges ............................................................................ 9 
    2.3.3 Implications on Security ........................................................................... 10 
  2.4 Chapter Summary ................................................................................................ 10 
3 Android Applications and Data Storage .................................................................. 12 
  3.1 The Structure of an Android Application ............................................................ 12 
  3.2 Android Application Data Storage ...................................................................... 13 
    3.2.1 The Need for Application Data Storage ...................................................... 13 
    3.2.2 Data Types in Android Applications ......................................................... 14 
    3.2.3 Android Storage Locations ....................................................................... 14 
  3.3 Protection for Android Application Data ............................................................. 15 
  3.4 Attack Vectors ..................................................................................................... 16 
  3.5 Chapter Summary ............................................................................................... 16 
4 Practical Approach to Data Security Analysis ......................................................... 18 
  4.1 Related Work ...................................................................................................... 18 
  4.2 Project Approach ................................................................................................ 19
List of Figures

Figure 1: The Android Architecture .......................................................................................... 4
Figure 2: Application Sandboxing and Resource Access on Android ........................................... 6
Figure 3: Process Credentials and Setuid(x) ............................................................................. 8
Figure 4: Contents of an Android APK File ............................................................................. 12
Figure 5: Practical Setup ........................................................................................................... 20
Figure 6: File Deletion and Data Residues on SD Card ............................................................... 26
Figure 7: Application Data Longevity on External Storage ......................................................... 29
Figure 8: Identifying Application Package Names .................................................................. 31
Figure 9: Attempting to Access Data via ADB Pull ................................................................. 33
Figure 10: Rooting the Android Device ..................................................................................... 35
Figure 11: Levels of Protection ................................................................................................. 43
Figure 12: Encryption Function in the Data Analyzer Application ............................................. 46
Figure 13: Structure of the Test Application .......................................................................... 57
Figure 14: Application Screen on Unrooted vs. Rooted Device .............................................. 60
Figure 15: Application Identifier within Hexdump ................................................................ 63
Figure 16: Application Code Retrieved from Memory Image ................................................... 63
Figure 17: Application File Names on External Storage ............................................................ 64
Figure 18: Application File Data on External Storage ............................................................... 64
Figure 19: Identifying Files Marked for Deletion .................................................................... 65
Figure 20: Recovery of Deleted File Data from External Storage ........................................... 65
Figure 21: “Hidden” Uninstalled Application ........................................................................... 66
Figure 22: Residual Data After Application Uninstallation ...................................................... 66
Figure 23: Residual Code After Application Uninstallation ..................................................... 66
Figure 24: Android Factory Reset Procedure ......................................................................... 67
Figure 25: Application Identifier Longevity on External Storage ........................................... 68
Figure 26: Application Code Longevity on External Storage .................................................. 69
Figure 27: Data Erasure from Internal Memory ..................................................................... 70
Figure 28: File Name Residues in Internal Storage ................................................................. 70
Figure 29: Deleted Code Segments .......................................................................................... 71
Figure 30: Code Residues after Application Uninstallation ..................................................... 71
Figure 31: Application Identifier Residues in Internal Memory ............................................... 71
Figure 32: Recovery of SD Card Encryption Key ................................................................... 72
List of Tables

Table 1: Security Expectations for Application Data on External Storage ........................................ 23
Table 2: Application Content and Description ..................................................................................... 27
Table 3: Percentage Residues of Application Data ............................................................................. 28
Table 4: Security Expectations for Application Data on Internal Storage ............................................. 30
Table 5: Possible Classification Scheme for Application Data ............................................................... 42
Table 6: The Different Flavours of Android ............................................................................................. 56
Table 7: Percentage Residues of Application Identifiers ................................................................. 68
Table 8: Percentage Residues of Application Code ............................................................................ 68
Table 9: Application Information for Data Longevity Experiment ....................................................... 69
## Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADB</td>
<td>Android Debug Bridge</td>
</tr>
<tr>
<td>ADT</td>
<td>Android Developer Tools</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APK</td>
<td>Android Application Package File</td>
</tr>
<tr>
<td>ARM</td>
<td>Acorn RISC Machine</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DDMS</td>
<td>Dalvik Debug Monitor Server</td>
</tr>
<tr>
<td>GID</td>
<td>Group Identifier</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Kit</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>MNO</td>
<td>Mobile Network Operator</td>
</tr>
<tr>
<td>NFC</td>
<td>Near Field Communication</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PBKDF</td>
<td>Password Based Key Derivation Function</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>SD</td>
<td>Secure Digital</td>
</tr>
<tr>
<td>SGL</td>
<td>Scalable Graphics Library</td>
</tr>
<tr>
<td>SIM</td>
<td>Subscriber Identity Module</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Sockets Layer</td>
</tr>
<tr>
<td>UID</td>
<td>User Identifier</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Executive Summary

Smartphone usage is on the rise and, of the numerous mobile operating systems that are available and in use today, the Android platform is one that is of particular significance. Its open nature has led to Android’s widespread adoption, and it is the operating system which is installed on the largest proportion of smart devices worldwide. Such being the case, the security of Android devices is a topic that garners much interest, since any vulnerability in the system will affect the greatest number of users.

If smartphone adoption is growing, then possibly more so is the use of smartphone applications. Such applications are performing increasingly complex functions, and are handling more and more data of sensitive nature. As such, we believe that application data security is an area which is likely to be of significant concern in future years.

This thesis analyzes the security afforded to third-party application data on Android devices. It first outlines the Android architecture design and identifies the security mechanisms in place for protecting application-specific data. We then move on to analyze the vulnerability of data on an actual device. To be able to achieve this objective, we develop an Android application, install it on a test device, and then conduct a number of practical experiments against the device.

The observations from our experimentation demonstrate that device-resident application data is at risk to a certain extent on Android phones, and that both the storage location and the root status of the device influence the protection afforded to such data. In particular, any data on external storage is always vulnerable to extraction, and is likely to be so for long after it is assumed to be deleted. When considering internal storage, we found that rooting a device provides full access to application data, but that the data deletion mechanisms are apparently more secure when compared with those for external storage.

Finally, we make use of our findings to propose an approach to safeguard application data even on vulnerable devices, via a data classification and controls mapping method.
1 Introduction

1.1 Project Background

Smartphones are rapidly rising in popularity and appear set to take over the global mobile phone market in a few years [1]. A “smartphone” is essentially a mobile phone that has an operating system, of which there are many in use today. Examples of smartphone operating systems include Android by Google, iOS by Apple, and the BlackBerry OS by BlackBerry Ltd. [2]. Dominating this market is the Android platform, making up 74.4% of worldwide smartphone sales as of 1Q13 [3]. The relatively open nature of the Android operating system enables it to be installed on a wider variety of devices than its primary competing platform: the closed-model iOS. This makes Android the more accessible operating system for most people, and is the reason for it to be selected as the subject for this study.

Alongside the growth in smartphones is the rise in the number of applications that are being developed for such devices. Where mobile applications were once limited to simple utility functions, such as calendars and calculators, they are now used to provide a wide variety of services, including mobile banking, mobile payments, and m-commerce.

In particular, it is becoming increasingly common for organizations to make available to their customers one or more mobile applications which provide the customers with a certain set of services, such as the ability to place retail orders via their mobile phones or to check their bank account balance [4]. It is also becoming common for such mobile applications to handle sensitive data pertaining to users, such as authentication credentials, preferences, etc. This gives rise to data security concerns, and any organization that develops and publishes a mobile application needs to ensure that the application stores and transmits user data securely, since information leakage by an application can have serious ramifications, including damage to the organization’s reputation.

Further, it is also becoming standard for enterprises to allow their staff to access business information, such as corporate emails and schedules, either on company-provided phones or on the employees’ personal mobile devices. In fact, companies often provide mobile applications for their workforce, to be used ‘in-house’, normally with the intention of improving efficiency and productivity. Such applications may connect to enterprise backend data systems and access critical or sensitive data, which may end up being stored on the user’s device. From an organization’s perspective, the protection of such data may be paramount.

The abovementioned concerns regarding data security are handled in different ways by different mobile operating systems, and the Android platform attempts to address these issues by utilizing application sandboxing and access permissions. It also disallows privilege-level access to normal users, in order to secure on-device programs and data. However, many of these mechanisms are not as effective as they should be. In particular, a process known as rooting, which grants privilege-level access to users, can bypass many of the security mechanisms employed by Android, and can enable malicious entities to access sensitive data [5]. On the other hand, it is sometimes possible for an attacker who has gained physical access to the device to retrieve application data even when the device is not rooted.
Given the increased focus on moving common consumer and business services to the mobile, the issue of application data security can only be more of a concern in the future. It is perhaps surprising then that, while a lot of effort has been expended in the past few years into analyzing the security of phone data and looking into data theft by malicious applications, not as much research has been conducted in the area of data security for legitimate applications. Again, where data security has been addressed, it is often more concerned with data in transit, rather than device-resident data.

This project analyzes the security of application data on Android devices, with focus on data that is stored on the device rather than that which is shared over a network. It does not concentrate on any particular application, but rather aims to analyze the security afforded by the Android platform to a generic application. It also determines the effectiveness of standard protection mechanisms, and proposes methods by which sensitive data can be safeguarded.

### 1.2 Project Objectives
The primary objectives of this project are as follows:

- Study the Android security model, and find out Android’s mechanisms for protecting device-resident application data.
- Identify whether protected application data can be retrieved from an Android device.
- Determine whether application data residues can be recovered in the discrete events of programmatic data deletion, application uninstallation, and device rooting.
- Examine the extent of application data that can be recovered when the phone is rooted when compared with an unrooted device.
- Propose methods by which application data can be secured.

Note that the project does not consider the recovery of phone data, such as SMS, call logs, phonebook entries, etc. Instead, it investigates application-specific data. In particular, it investigates application data that is not intended to be publicly viewable.

### 1.3 Methodology
The nature of the project results in a lot of the references being more readily available from the Internet, as opposed to printed materials, although books which address aspects of the subject have been referenced wherever possible. Most details regarding the Android architecture and security model have been obtained from technical documents or sites administrated by Google. The rooting process and application security are topics which are widely discussed on developer sites and technical blogs, and which have been the subject of several technical papers. These form the foundation of the literature review.

The project also has a significant practical component, in which we develop an Android application and use it to test certain aspects of application data security.

The results from the practical experiment, coupled with findings from the literature review, are used to assess the level of security that is actually afforded to application data on Android devices.
1.4 Structure of Report

The first chapter of this report discusses the motivation behind the project, and sets out the objectives that the project aims to achieve. It also provides a high-level overview of the methodology that was followed.

Chapter 2 contains the findings from the literature review, and is focused on the Android platform. It describes the architecture of the Android mobile operating system, and explores the protection mechanisms utilized by the platform to secure application data. In addition, it explains the concept of root users on Android, and briefly looks at the mechanisms behind the rooting process.

The third chapter deals with Android applications and application data. It outlines the structure of a generic Android application, and describes the various types of data that an application may create, store and handle during its lifetime. This chapter also specifies the default protection afforded to these data types by the Android platform.

Chapter 4 discusses the analysis approach followed in this project. It begins with an overview of similar work done in this arena, and then moves on to differentiate the project from existing research in this field and to justify the need for practical experimentation. It also details the set up that was used for the practical element of the project, including the tools that were used for development and analysis.

The next three chapters revolve around the practical experiment, and use the findings from the experiment to determine the actual security of application data on Android. Specifically, the fifth chapter focuses on data security on external storage, the sixth chapter analyzes the protection of data stored on internal storage, and the seventh chapter discusses the effect of rooting an Android device.

The practical experiment makes use of an older, albeit widely used, version of Android. Therefore, to address concerns of relevance, Chapter 8 discusses how security has been enhanced in newer versions of the platform. It also describes instances where features in newer operating systems may actually result in less protection when compared with older versions.

Chapter 9 proposes methods that can be utilized by application developers to safeguard any data their application may handle, and the final chapter states the project’s conclusions and suggests possible avenues for further research.
2 The Android Architecture and Security Mechanisms

When discussing Android security, the most logical starting point is with the design of the Android platform. This chapter summarizes the findings from the preliminary literature review, which examined the Android architecture and explored how data security and separation of concerns is handled by the Android operating system. It also looked at the concept of root users and the mechanics of the rooting process.

2.1 Architecture of the Android Platform

Figure 1 [6] illustrates the major components of the Android platform.

![Android Architecture Diagram](image)

At the core of the Android operating system is a stripped-down version of the Linux 2.6 kernel [7]. This provides the driver model necessary for interfacing with hardware components, as well as low-level functionality such as power management, memory management, user management and security. Above this layer is a set of native C/C++ libraries, which expose various capabilities (including but not limited to media, graphics and database services) to developers via an application framework. At this level is also the Android Runtime, which consists of core Java libraries and the Dalvik virtual machine (VM). The Dalvik VM is a mobile-optimized virtual machine of which every Android application runs a separate instance. [6], [5]

Further up in the architectural layers is the application framework, which makes available to Android developers several APIs that provide commonly-used functionality, for use in developing Java-based applications. Android applications themselves reside in the uppermost layer of the architecture. [5]
2.2 The Android Security Model

2.2.1 Access Permissions: Linux vs. Android

As mentioned previously, the Android platform is built upon the Linux kernel. This means that it follows the same basic access control principles as those used by Linux [7].

In Linux-based operating systems, identification by the system of users and groups is accomplished via user IDs (UIDs) and group IDs (GIDs), and a process normally runs and accesses resources with the same permissions as the user who initiated the process [5].

Further, every resource on the system has associated with it a set of permissions, defined separately for three different ‘classes’ of user: the owner of the resource, a group (of users) associated with the resource, and all other users. For each user class, three bits are allocated to define the three different access rights (read, write, execute) to the file. This means that a single file will have nine binary flags associated with it to specify access rights. [8]. Note that the terms “file” and “resource” are used here interchangeably.

A file will also have three flags - suid, gid, and sticky - to define the mode of access. When considering executable files, the flags have the following connotations [8]:

- **suid or setuid (Set User ID)**: When an executable file is executed by a process, the UID of the process is usually the UID of the process owner. If, on the other hand, a file has the setuid bit set, then the process will take on the UID of the file owner.

- **sgid or setgid (Set Group ID)**: When a file is executed by a process, the GID of the process is usually the GID of the process group. But when a file has the getuid bit set, then the process will take on the GID of the file group.

- **sticky**: The sticky bit, when set, denotes an instruction to the operating system to keep the program in memory, even after it finishes executing.

The Android access control model follows much the same principles as Linux, with one major difference. Where in Linux systems, all applications would run with the permissions of the user and have the same UID (which effectively means that all applications initiated by the same user will have the same access rights to system resources), in Android devices, every application is treated as a separate user and has its own UID. Further, every resource stored by that application is assigned the same UID, and the application only has full permissions to its own resources. This is because Android applications can be installed from various different sources, and may not all be trustworthy. [7], [5]

An Android application will also be able to access system resources, but only in a restricted manner via API calls. It, however, will not be able to access the memory space, data or processes of any other application, since such resources will be associated with a different UID. In this manner, application sandboxing is achieved. [7], [5]

2.2.2 An In-depth Look at Application Sandboxing and Resource Access

As previously discussed, Android works as a multiuser operating system where every application is treated as an individual user. When an application package is installed onto an Android device, the kernel assigns a unique UID to the application and its associated files, databases and other resources. When an application is executed, it runs in its own instance of the Dalvik VM, isolated from other applications. [9]
A common mistake to make is to assume, because every application has its own instance of a virtual machine, that the Dalvik VM is the entity which enforces application sandboxing and security. This is not actually the case. Android enables developers to write application code in Java by making use of the provided APIs. This Java code is what eventually runs on the Dalvik VM (after compilation and conversion). [7]

However, it is also possible to compile and include native ARM architecture code. Native code runs outside the control of the Dalvik VM and is not restricted to the APIs made available to developers. However, even this native code has to follow the access limitations imposed by the kernel. Therefore, it is only the Linux kernel and not the Dalvik VM which ultimately enforces application sandboxing on Android platforms, and the sandbox includes both the Dalvik VM as well as any native code. [7]

Figure 2 [10] illustrates the different ways in which a sandboxed application could access various types of resources. Here, Browser, App, and AppAddOn are three separate applications, each in its own sandboxed environment.

(1) An application requiring access to (its own resources in) the filesystem: The kernel checks the UID of the application and the UID of the requested resource. If the required permission is available, access to the resource is granted [10].

(2) An application requiring access to system resources: System resources include the device camera, Bluetooth, GPS, etc. Any developer who requires these services within an application must explicitly specify all such services in a manifest file, and the end user of the application is requested permission for access to these services when the application is first installed [5]. Even when the user provides permission, the application is not allowed to make a direct call to such system resources. Instead, it uses a provided API from the application framework. The system server checks whether the application has permission for the requested resource and, if it does, makes a call to the relevant kernel driver on the application’s behalf. [10]
An application requires access to another application’s resources: This is usually only possible when both applications have been developed by the same entity and signed using the same certificate. Either the system server or the application itself will check for the relevant permissions. [10]

2.2.3 The Concept of a Root User
Every Linux-based operating system (and hence, the Android operating system) has a special type of user known as root or superuser. As the name suggests, this user has ‘super’ privileges over the entire system. The Android operating system does not apply the same restrictions to the root user as it does to all other users. This means that the root user is able to access all files and control every program on the system. [8]

The root user or superuser has a UID of 0 (similarly, there exists a concept of a root group, with GID = 0). When a process executes with UID = 0, the operating system will not go through the permission checking process, but will allow the process to perform a variety of privileged actions. [8]

Obviously, root privileges are dangerous, especially in the case of Android, where users (i.e., applications) are not necessarily trusted. Allowing all applications root access would effectively destroy the efforts of sandboxing, since every application would be able to access and interfere with the data of every other application. For this reason, only the kernel and certain core services run as root on Android, while all other applications run with limited privileges [6]. There are also mechanisms in place in the Android platform to prevent unprivileged users from gaining privileged access to the system. These have been described in Sections 2.2.4 and 2.2.5.

2.2.4 Process Credentials and Prevention of Privilege Escalation
Section 2.2.1 mentioned the file permissions that are present in Linux systems. In addition, every process on a Linux platform has certain credentials associated with it, which determine what the process can or cannot do. There are three such process credentials [8], [11]:

- uid or ruid: The real user ID. This is the ID of the owner of the process.
- euid: The effective user ID. This is what is used to make most access control decisions.
- suid: Saved user ID. This is a previously used ID.

A process, when first created, inherits the credentials of its parent. These credentials (and hence, the process’ privileges) can change either when a new program is executed or via certain system calls. [8]

When a process executes another program, its credentials may change, depending on the type of program being executed. If the setuid bit of the called program is set, it is known as a setuid program and the calling process’ euid is changed to the UID of the called executable file’s owner. Otherwise, the process credentials remain unchanged. [8], [11]

Given that access control depends in large part on the euid of a process, this euid value can also be changed by uid-setting system calls, such as setuid() [11]. If the process’ euid is zero (i.e., the process is already running as root), then executing setuid(x) will change all three of the process’ credentials to x [8].
If, on the other hand, the process' \textit{euid} is \textit{not} zero, then \textit{setuid} will only change the \textit{euid} to \textit{x}, but will not affect the \textit{ruid} or \textit{suid} values. Also \textit{x} has to already be present in the process credentials as the value of either \textit{ruid} or \textit{suid} in order for \textit{setuid} to be executable when \textit{euid} is not zero. [8]. We have created an illustration based on this decision flow, as depicted in Figure 3, where a process begins with credentials \textit{euid}=a, \textit{ruid}=b, \textit{suid}=c, and then executes the command \textit{setuid}(x).

![Figure 3: Process Credentials and Setuid(x)](image)

It has already been mentioned that most Android applications do not have root privileges. That is, most application processes do not have \textit{euid}=0. In order to escalate their privileges to root-level, applications must either execute a setuid program or make \textit{uid}-setting system calls [12].

Obviously, for security reasons, not every program should be a setuid program, nor should every process be allowed to make \textit{uid}-setting system calls. The \textit{su} and \textit{sudo} binaries in Linux are examples of programs that are able to make \textit{setuid}(0) calls to enable a currently-running unprivileged program to perform privileged functions. However, they are not normally available on a stock Android operating system. In fact, the Android platform has been designed in such a way that an unprivileged user application is not able to start another program as root, nor is it able to modify its own privileges to root via system calls. In this manner, privilege escalation is prevented on Android systems. [12]

\subsection*{2.2.5 The ADB Shell and Root Privileges}

The Android Debug Bridge (ADB) is a command-line utility that allows users to issue commands to an Android device or emulator instance. It allows for a variety of functions including installation of applications to the device, copying of files between a computer and the device, and (safe) killing of user processes, among others [13].
Connecting an Android device via USB to a computer and executing the `adb shell` command from the desktop command line tool will bring up an ADB shell from which the user can communicate with the mobile device. Users of command line tools in desktop environments (where the user often has administrative rights) may then expect that the ADB shell will enable them to execute privileged instructions on the device. However, this is most often not the case. [12]

The privileges that an ADB shell process will execute with depends on the value of a system property called `ro.secure` in Android. When this value is equal to 0, the shell process will execute with root privileges. Most often, however, the value is set to 1 on the versions of Android that are shipped with devices, which means that shell commands will be run as unprivileged. Further, the `ro.secure` property value is read-only and can only be changed by a privileged user. This prevents any non-root user from gaining root-level access to a device via the ADB shell. [12]

### 2.3 Rooting an Android Device

As mentioned in the previous sections, the versions of Android that ship with standard devices do not normally allow root privileges. *Rooting* is simply the process by which a user (an actual user or an application) gains root privileges for an Android system.

#### 2.3.1 Possible Reasons for Rooting

Rooting in itself is not necessarily a malicious act and is not illegal (although it is often believed to be). Legitimate users may root their devices for a number of reasons, some of which may be [14], [15]:

- Unlocking additional features on the device.
- Obtaining fine-grained control over device performance.
- Customizing the device beyond what is normally allowed.
- Gaining a ‘leaner’ operating system, without manufacturer or carrier bloatware.
- Getting access to the latest version of the Android operating system, without having to wait for carrier updates.
- Being able to install applications from various sources, not just official/carrier-approved marketplaces.

#### 2.3.2 Gaining Root Privileges

There are two main paths by which a user or application can gain root access to an Android device [6]:

1. **Unlocking the bootloader**

   The bootloader is the piece of code that runs immediately on power ON and which is responsible for loading the device operating system and the system recovery program [12] (which are both present on ROM). A mobile device is usually locked to the bootloader it ships with. To obtain root access to a device, the default recovery partition or ROM will have to be replaced with custom firmware which provides root access. However, this will not be possible for as long as the stock bootloader is present. Therefore, the first step in the rooting process is normally to ‘unlock’ the bootloader and then load custom recovery partitions or ROMs. [16]. During the unlocking process, any existing user data on the device is wiped [6], thereby reducing the likelihood of data compromise.
2. Exploiting a security vulnerability

In this method, root privileges are obtained by exploiting a security hole in the Android platform and performing privilege escalation [6]. As mentioned previously, Android limits the ability to run programs in privileged mode to processes which already have root privileges. If an Android device is found to contain any such privileged system process with a vulnerability that can be exploited to trick it into executing some user-provided code, also in privileged mode, then that device can be rooted. The user-provided code will make use of its root privileges to copy the `su` binary onto the system [12], after which any application can be run with elevated privileges. In contrast to the bootloader unlocking process, this method does not wipe user data. Therefore, existing user data becomes vulnerable to compromise [6].

Differences in the two rooting techniques

While the two techniques described above appear to achieve the same goal in that they both enable privileged access to the device OS, our understanding is that they differ slightly with respect to user control and data security. Bootloader unlocking is, for the most part, controlled by the user (as the unlocking process requires physical access to the device), unless the device has made its way into the hands of an attacker. Exploitation of security holes, on the other hand, may be accomplished by a malicious application, unknown to the user. Further, while data stored on the device after gaining root access is vulnerable in both scenarios, bootloader unlocking purportedly protects existing data from compromise.

2.3.3 Implications on Security

As mentioned previously, rooting is a process by which privileged access can be obtained to the Android platform. This clearly has serious implications on security. That is, by rooting a device, the security restrictions imposed by Android are circumvented [17].

Where once any malicious application that happened to get installed on an Android device was limited in what it could do, rooting now allows the same application to perform privileged functions and possibly take control of the system entirely. For example, malware on a rooted device might be able to send text messages, forward the user’s contact list to a remote location, and steal application data [17].

Further, an attacker who is able to obtain physical access to a rooted device will be able to retrieve a lot of sensitive information that would have been protected had the device not been rooted.

Therefore, any user who considers rooting an Android device should first be aware of the impact the process will have on the device’s security [17].

2.4 Chapter Summary

This chapter introduced the architecture of the Android platform, and outlined its components. It specified that Android is essentially a leaner version of the Linux 2.6 kernel, and that it handled access control in much the same way as Linux, using user IDs and access permissions for files. The main difference between Linux and Android was noted to be that, where all applications executed by a single user have the same UID in Linux systems, Android assigns a unique UID for each application.
Android applications are protected by a combination of techniques. Firstly, applications are *sandboxed* in that an application with a particular UID is prevented from accessing the code or data of other applications with different UIDs. To prevent this sandboxing from being circumvented by root or *super* user privileges, Android devices do not normally ship with root privileges enabled. In addition, privilege escalation is also prevented via different means.

Rooting is the process of getting privilege-level access to the Android platform, and can be achieved by either unlocking the device bootloader, or by exploiting a security vulnerability. While it is not an illegal operation, caution must be exercised if rooting is undertaken as it can open up the device to attack.

Now that the overall design of the Android system has been described, the next chapter will discuss the specifics of Android applications and application data storage.
3 Android Applications and Data Storage

Because this thesis deals specifically with Android application data security, it may be pertinent at this point to describe the overall structure of Android applications, as well as certain aspects of application data storage. In particular, this chapter specifies the need for data storage, as well as the data types that are typically handled by applications, the locations in which such data may be stored, and the protection provided to them by the Android system.

3.1 The Structure of an Android Application

Android applications are typically distributed as Android application package files. An application package file is an archive containing an Android manifest file, the compiled application code, data structures and other resources (such as images, for example). It has an .apk extension, and is often simply referred to as an APK file. [9], [18]

The components contained within an Android APK file have been shown in Figure 4.

![Figure 4: Contents of an Android APK File](image)

Those elements which are of significance with regard to this project have been described below.

*The Android Manifest file*

An application may have many components, all of which must be declared to the system before use. The AndroidManifest.xml file is used for this purpose. An Android developer must use this file to specify application components, permissions required by the application (such as write-access to the Secure Digital (SD) card, read-access to user contacts, etc.), and other information that is required in order for the application to run correctly [9]. One attribute within this file which is of particular interest is the application package name. This is a string, usually in the reverse domain name format (e.g. com.<domain>.<app_name>), which is used to uniquely identify an application [19].

*Application code*

An Android APK contains a .dex file named classes.dex. This is the collection of classes defined in an application, in a format that the Dalvik virtual machine can understand [18]. It is possible to decompile this file into a Java .jar file via several tools, to obtain and view the original application classes.
Application resources
An application typically makes use of image files, media files, and layout files, in addition to the application code itself. Some applications will also ship with data structures (such as databases) as part of the application’s resources. These additional files are stored in either a res or assets folder within the APK file [9].

In addition, most applications will create and access data structures such as files and databases during their operation. Application data can include sensitive information such as private communications, login credentials, company schedules and corporate emails, and banking or other financial information [20].

Since application data is the primary focus of this project, the following sections have been used to describe certain important aspects related to application data.

3.2 Android Application Data Storage

3.2.1 The Need for Application Data Storage
Given that there are a number of concerns surrounding application data security, the most logical method to ensure security might be to simply not store any data on the device. Indeed, this is the advice given by most security researchers [20], [21]. However, there are certain legitimate reasons why an application may require on-device data storage. Some such reasons have been discussed below.

Ease of use
An application that requires the user to log in before proceeding may allow the user credentials to be stored locally on the device, to simplify the sign-in process for the user. Otherwise, the user may become frustrated at having to enter his credentials each time he wishes to use the application.

Limited connectivity & offline access
If the application is installed on a device which operates in areas with limited connectivity, the application developer may wish to provide the user with a certain subset of offline functionality [20]. This way, the user is able to utilize the application even when he is out of the range of any network. This may require the storage of some data on the device.

Large volumes of data
It may be the case that the application in question downloads a significant volume of data from a backend server and displays it to the user. If the data was not persistent on the device, the next time the user wanted to view the data, the application would have to re-download it from the server.

Given that users are charged for data usage by volume, this could represent a significant cost to the user. It is also an unnecessary burden on the server. To reduce the costs associated with this method, developers may choose to store the data on the device instead.
3.2.2 Data Types in Android Applications

As seen in the Section 3.2.1, applications may need to store data on the user device for a variety of reasons. The data that is stored by an application can be of different types, and the choice of a data type normally depends on the nature of the information being stored.

Android specifies five data storage options for applications [22]:

- Shared preferences.
- Internal storage.
- External storage.
- SQLite databases.
- Network connection.

For the purposes of this project, the above storage options are reclassified slightly. We take databases and shared preferences to be examples of data structures, whereas internal and external storage, and even data stored over a network, are considered to be examples of storage locations.

Accordingly, this thesis classifies types of data used in Android applications as:

- **Files**: Files can be of many different types, such as images, text files, etc. Whether a file is private or public depends on its storage location [22]. The main storage location options are described in Section 3.2.3.
- **Databases**: Android supports the use of SQLite databases within applications, and these databases, when created using Android’s recommended methods, are private to the application by default [22].
- **Key-Value Pairs**: Android allows for storing primitive data types (such as Booleans, integers and strings) as key-value pairs. These are also known as shared preferences, and are suitable when small amounts of data need to be stored. This information is private to an application and remains persistent across sessions [22].

Something else that the developer may be concerned with, albeit not strictly application data, is the **application code** itself. APK files can be reverse engineered to obtain the source code of the application [23], which may be undesirable from the point of view of the developer. This may be due to intellectual property concerns, especially in the case of paid applications. It may also be because the application code specifies communication endpoints in the developer’s network, the details of which the developer would prefer not to expose.

3.2.3 Android Storage Locations

The two main location options for application data storage on Android are internal and external storage. While Android also specifies transferring data to a network (i.e., a server) as a storage option [22], data stored in this manner would not reside on the device, and would have a different range of controls applied to it when compared with data stored on-device. Further, data stored on a network would not normally be afforded any protection by the Android system. Instead, all protection mechanisms would have to be implemented by the developer.

Because this project primarily focuses on analyzing the security of device-resident application data, we do not consider network storage any further.
Internal storage
This refers to storing data in locations on the device’s internal memory. By default, files and databases stored by an application in internal storage are not made available outside the application [22].

All data belonging to a single application will be stored within the application data directory, which can be found within the Android system at /data/data/<package_name> [20]. The <package_name> is, as mentioned in Section 3.1, a unique identifier of the form com.<domain>.<app_name>.

The application data directory is configured such that the UID of the application is defined as the directory’s owner. This means that only processes with the same UID (and the root user) will be able to access data within this directory [5]. Therefore, internal storage is the most logical location to store private data. However, many phones (particularly older models) ship with only a small amount of internal memory, and this will be exhausted very quickly if it is used as the storage location for all applications.

External storage
This refers to storing data either on removable storage, such as an SD card, or on non-removable embedded storage. Data on external storage is world-readable, and is accessible when the SD card is read using a standard reader or when USB mass storage is enabled [22]. This would therefore normally not be considered a suitable option for storing sensitive data.

Android itself specifies that “there's no security enforced upon files you save to the external storage. All applications can read and write files placed on the external storage and the user can remove them” [22] and that “external storage is the best place for files that don't require access restrictions” [24]. However, developers may have no alternative but to store applications in external storage (especially if the application is large), given that internal memory may be limited.

To facilitate a certain degree of protection for applications and application data stored on external storage, Android encrypts these applications [20] and also provides methods by which private locations can be created for application data. According to Android, files stored in such locations are “private to the applications, and not typically visible to the user as media” [25].

Further, even though the application .apk file is stored on external storage (in encrypted form), all private user data and databases are stored in the internal device memory [26].

3.3 Protection for Android Application Data
Data security for Android applications needs to be considered from two angles:

- Secure data storage: Data stored by an application should, by default, be accessible to only that application, unless the data is specified to be public. As mentioned in Section 3.2.3, Android achieves this by sandboxing the application data directory to ensure that no application has access to resources belonging to another application. This protection is, for the most part, restricted to data that is stored within internal memory.
Secure data erasure: It is also important that application data, once deleted, should no longer be recoverable. Complete erasure of application data is necessary in multiple scenarios. For example, when data is deleted programmatically via the application, when the application is uninstalled, or when the device is reset. Secure data deletion is especially essential in the event that an Android device transfers ownership from one user to another, to ensure that no new user is able to access any residual application data belonging to the previous owner of the device.

3.4 Attack Vectors

As mentioned previously, applications often store a significant amount of data, some of which may be sensitive. Such data can be of interest to criminals, possibly for identity theft or corporate espionage. These criminals may employ many methods by which to steal device-resident application data, and most such methods can be broadly classified into two categories [20]:

Remote attacks:
This is where a malicious application manages to install itself on the device, possibly by masquerading as a legitimate application. It can then execute malicious code to transmit data to a remote location. The malware may be limited in the attacks it can carry out and the extent of data it can retrieve, but the data theft will usually go unnoticed by the device user, which is to the attacker's advantage.

Local attacks:
This is where an Android device gets into the hands of a malicious user. While it may seem unlikely, this is actually not an unimaginable scenario, since mobile phones are often mislaid or stolen [27], [28]. Further, the trend nowadays is to use a device for a short time before reselling it in favour of newer models [20], and it may be that the buyer of a used phone has malicious intentions (or even merely curiosity) and may try to recover data belonging to the previous owner.

An attacker who is able to gain physical access to a device is in a powerful position and can employ a number of methods to try and retrieve data from the device. With this type of attack, however, the victim (the legitimate user of the device) will be aware of the fact that the phone has been lost or stolen, and may be able to remotely lock the phone or wipe its data.

Note that this thesis focuses primarily on local attacks against Android devices.

3.5 Chapter Summary

This chapter explored the structure of an Android application package, or APK file. It briefly described the major components within an application, before moving on to describe aspects of Android data storage.

The data handled by an application can be of three main types: files, databases or key-value pairs. Of these, files can be private to an application or publicly visible, depending on where they are stored. Databases and key-value pairs are private by default.
This chapter also identified that data can be stored in two main locations on an Android device: internal or external storage. Data stored on internal storage is private to the application, whereas that which is on external storage is considered to be public and does not have the same level of protection. However, developers may be forced to save data in external storage due to internal memory limitations on devices.

Now that the concepts of Android security and application data have been introduced, the next chapter describes the approach followed during this project to analyze the security of application data on Android devices.
4 Practical Approach to Data Security Analysis

The objective of this project is to analyze how secure application data is when it resides on an Android device. That is, an Android application may store different types of data on its host device, along with the application code, either on internal or external storage. This data and code will be afforded a certain degree of protection by the Android platform. The primary goal of the project is then to identify exactly how effective such protection is.

When analyzing data security, a number of factors need to be taken into consideration. For example, we would like to know not only whether data can be retrieved, but also how much data can be retrieved. In addition, when application data is deleted or an application is uninstalled or device storage is formatted, we would like to find out whether the data has actually been removed completely, or whether residues can be recovered. Where residual data remains, it may also be useful to know for how long after an application is uninstalled its data will be recoverable.

4.1 Related Work

Before commencing the analysis of Android application data security, existing literature was reviewed to identify whether others have performed similar research in this area. We observed that several different avenues of study have been conducted in the field of Android security, and these are briefly outlined below.

*Studies regarding leakage of phone data by the Android platform*

These studies identify vulnerabilities in the Android platform and the way it handles user passwords and data, but usually focus on the theft of phone data, rather than application data. An example is the work done at Ulm University, Germany, regarding the insecure handling of authentication tokens by Android, which could potentially expose user credentials over public Wi-Fi networks [29], [30].

*Studies regarding data security in specific applications*

This type of research analyzes how widely-used applications handle user information (both in storage and in transit), and whether the applications protect such information adequately. Examples of such studies include the *Mobile App Security Study* by viaForensics [31], [32]. These reports shed light on development vulnerabilities, rather than platform weaknesses.

*Studies regarding the forensic analysis of Android devices*

This type of study analyzes how data can be recovered from a device. One of the most comprehensive texts for Android forensics is Hoog's *Android Forensics: Investigation, Analysis and Mobile Security* [20], which describes data theft targets, attack vectors, and forensic procedures for extricating data from an Android device. Lessard and Kessler [33] have built upon this work, and have analyzed the data that can be recovered from an Android phone using similar techniques as those described by Hoog.

However, forensic studies differ in perspective from this project in that forensic tests aim to retrieve as much data from the device as possible, and will usually have a relatively heavy focus on recovering phone data (such as call logs, contacts and emails) as opposed to application data. Having said this, the techniques employed in forensic analyses are probably the most relevant to this project.
Studies regarding application security

In recent years, there has been a slight increase in research conducted on application data security from the developer’s perspective, which has resulted in a number of publications on the subject. Possibly a forerunner in this area is the book Application Security for the Android Platform by Jeff Six [5]. This book outlines the Android architecture and security mechanisms, mentions the threats to application data, and describes methods by which to protect such data. However, the aim of the book is to aid developers in secure application development, rather than to verify Android protection mechanisms, which means that not as much focus is placed on proving data insecurity.

4.2 Project Approach

The review of existing research showed that, while a lot of work has been done with regard to analyzing Android security, there is still insufficient information to understand the actual protection afforded to generic Android application data (or even to validate claims of insecurity). For this reason, it was deemed necessary to conduct a practical experiment to obtain this information.

We developed a test Android application for this purpose, to be able to more accurately determine exactly how much application data can be retrieved from an Android device. The test application is used to store data in different device locations using the options mentioned in Sections 3.2.2 and 3.2.3. We then analyze the device for retrievable data, as well as residual application data in the discrete events of programmatic data deletion, application uninstallation, and device reset.

The practical experiment is split into two main sections, and the first section explores the security of data that is stored on unrooted Android devices (i.e., when the device is in the configuration as shipped from the manufacturer or operator). This first part is divided into two subsections, to analyze separately data security on external storage (discussed in Chapter 5) vs. internal storage (Chapter 6). The second part of the practical is concerned with extracting data from rooted devices (and has been documented in Chapter 7). Since the rooting process does not normally affect data on the external storage, this part of the analysis focuses mostly on internal storage.

4.3 Developing the Data Analyzer Application

The Data Analyzer is the test application that we created for this project. It was developed using the standard Java APIs provided by Android. The overall practical setup has been illustrated in Figure 5. Details regarding the development environment have been provided in Appendix A, while Section 4.4.3 describes the tools used for data extraction and analysis.

Test Application Functionality

The Data Analyzer application is quite straightforward, with all of its functionality being held in a few classes. At its very basic, upon execution, the application enables the user to write pre-defined data (as files, databases and key-value pairs) to default locations on the device, and then delete data using standard file deletion methods. As the experiment progressed, additional functionality was added to the application, such as root-checking mechanisms and encryption. The application was first written to be installed on external storage, and then modified for internal storage, to enable separate analyses.

~ 19 ~
The test application targeted an older version of Android (v2.3.6), and the rationale behind this decision has been provided in Appendix A. However, it could be argued that security in later versions of Android has likely been improved, and that shortcomings identified by testing on v2.3.6 may have since been overcome. To address this concern, Section 8 briefly describes the security enhancements in newer versions of Android.

4.4 Data Extraction and Analysis Methodologies

In order to be able to analyze Android application data, there first needed to be some method by which data could be extracted from the Android device.

Data acquisition can be performed against an Android device in two main ways: via logical techniques or physical techniques [20]. These two methods have been described in Sections 4.4.1 and 4.4.2.

4.4.1 Logical Data Acquisition Techniques

Logical acquisition methods normally access the file system and return data as allocated by the device operating system. This type of data extraction is usually easier to perform when compared with physical methods, and works to a certain extent even on unrooted devices. However, because the operating system controls the data that is returned, logical techniques will not provide access to all the data within a system. [20]

There are a number of tools that can be used for logical data extraction from an Android device, some more effective than others. Below are some such tools:

- **File Explorer:** This refers to the standard file explorer tool available with most operating systems. Data acquisition by this method simply involves connecting the Android device or SD card to a PC via USB or a card reader, and viewing the files via Windows File Explorer or similar software. This method does not allow for viewing the contents of internal memory.
- **ADB Shell:** The Android Debug Bridge is a very useful command-line tool that is packaged along with the Android Software Development Kit (SDK). It is capable of executing commands against a device that is connected to the computer via USB [13], to query files residing on the device. The `adb pull` command is particularly useful. When executed from a Terminal or Command Prompt, this command is capable of pulling files from the device, provided the required access permissions are available. The syntax for this command is `adb pull <path_to_file_on_device> <path_to_copy_to_on_local_computer>`. ADB typically allows more data to be extracted than when using a File Explorer, and provides access to some files on internal memory, but still respects the operating system control over data.

- **The Android Debug Monitor with the Dalvik Debug Monitor Server (DDMS):** This tool also comes bundled with the Android SDK, and displays most of the files that are extractable through ADB in a File Explorer view. This makes for easier navigation and exploration. However, hidden files are not accessible using this tool.

- **Android Backup:** Newer versions of Android (versions 4.0 upwards) allow a device to be backed up by executing the `backup` command via ADB [34]. This method could potentially allow access to a larger amount of application data.

- **AFLogical:** This is a free tool developed by viaForensics, which allows for the extraction of a certain subset of data residing on Android devices [35]. However, the data that is extracted is usually that which is exposed via content providers [36] (where a *content provider* is an interface in Android which enables code in one process to access data in another process [37]), and it appears that none of the AFLogical variants provide access to application data which is *not* made accessible through content providers.

### 4.4.2 Physical Data Acquisition Techniques

Physical extraction methods involve getting raw data from memory, and are independent of the file system. These methods are usually more complex, but are normally able to retrieve more data than with logical methods [20].

Examples of physical data acquisition techniques have been given below:

- **Memory Dumps (Memory Images):** This is a software-based technique in which the device is connected to a terminal and the visible memory is imaged, bypassing the file system [20].

- **JTAG:** The Joint Test Action Group (JTAG) technique is a hardware-based method which enables imaging of the memory chip within a mobile device, and is capable of recovering all data stored on memory. JTAG was originally used for testing printed circuit boards (PCBs) [38] and involves soldering leads to JTAG pads on the PCB to obtain a direct connection to the device’s central processing unit (CPU) [20].

- **“Chip-off”:** Another hardware-based technique is ‘chip-off’. In this method, the memory chip is removed from the device and inserted into a special hardware reader to be able to extract its contents [20].
4.4.3 Data Acquisition and Analysis Tools Used during the Practical Experiment

From the options provided in Sections 4.4.1 and 4.4.2, we made use of the following tools to extract data from an Android device:

**Logical extraction tools**
- Windows File Explorer.
- The Dalvik Debug Monitor Server: for easier navigation and exploration of files.
- Windows Command Prompt and the ADB Shell: for issuing ADB commands.

Android backups could not be used in this project due to the fact that the test device was running an older version of Android, which did not support the backup feature. AFLogical tools were not used as it was understood that they could not extract private application data.

**Physical extraction tools**
Only memory dumps were used for physical acquisition in this project, since both hardware-based methods (JTAG and chip-off) would have required at least partial disassembly of the phone, and might have resulted in irreparable damage to the device [20], [39].

The following tools/platforms were used for obtaining memory images:
- Oracle VM VirtualBox v4.2.12 with virtual machine running Ubuntu 12.04 LTS: for imaging external storage.
- BusyBox: for imaging internal storage on rooted devices.

Note that because the memory images were obtained using the Linux command `hexdump` and are presented in a hexadecimal view, they are also referred to as hexdumps within this project report.

**Other software tools**
Once memory images had been acquired, they were formatted and analyzed using the following tools:
- Windows PowerShell v1.0: to split large hexdumps into smaller, manageable files.
- WinMerge v2.14.0.0: to compare hexdumps.
- Notepad++: to view and analyze hexdumps and other files.

4.5 Chapter Summary

This chapter introduced existing research that has been done around the area of Android security, and differentiated the focus of this thesis from other work. It then provided high-level details about the application that was developed as part of this project and how it was used to analyze application data security. The various tools and techniques that were used during this project for data extraction and analysis were also described.

The next chapter describes the first part of the practical experiment that was conducted, which was the analysis of application data security on external storage.
5 Assessing the Security of Data in External Storage

Because internal and external storage are very different, in that they have different controls applied to them by the Android platform, we analyze data security on these two storage locations separately. This chapter describes the practical experiment and findings pertaining to application data security on external storage.

5.1 Specifying Security Requirements for External Storage

Before evaluating external storage security, we first define a set of requirements against which data security can be assessed.

While it is true that SD cards are not intended to be secure storage locations, given that users can easily remove them and examine their contents, there can still be certain expectations regarding the protection afforded to data on external storage. These expectations are detailed in Table 1, and are either defined based on statements in the Android documentation regarding the way in which Android handles data in external storage, or are expectations that can be considered as reasonable from a developer’s perspective.

<table>
<thead>
<tr>
<th>Req_Ext_01</th>
<th>Application code that is stored on external storage should be protected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given that anyone can read the contents of an SD card, in the interest of protecting the developer’s intellectual property, it is desirable for the APK file on the SD card to be protected via encryption or some other means.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req_Ext_02</th>
<th>Private files stored on external storage should not be easily accessible as media</th>
</tr>
</thead>
<tbody>
<tr>
<td>This requirement stems from Android’s statement that files stored in private locations on SD cards are not usually visible as media [8]. This was taken to mean that such files would not be visible to the user via standard file explorers.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req_Ext_03</th>
<th>Files deleted using Android API delete methods should not be recoverable</th>
</tr>
</thead>
<tbody>
<tr>
<td>While it is uncertain as to how much control Android has over SD card content deletion, ideally any file that Android specifies as having been deleted should no longer be recoverable from the card.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req_Ext_04</th>
<th>Application uninstallation should remove all traces of the application</th>
</tr>
</thead>
<tbody>
<tr>
<td>When a device changes hands, an unwary user may simply uninstall all the applications he had previously installed, before transferring ownership of the device. The new owner of the device should not be able to recover any information regarding previously installed applications or their data.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req_Ext_05</th>
<th>SD formatting via Android reset should permanently remove all application data</th>
</tr>
</thead>
<tbody>
<tr>
<td>This expectation again follows from the trend for reduced device usage lifetimes as mentioned in Section 3.4. An Android owner who formats his SD card as part of the overall device reset procedure (prior to reselling the device), would reasonably expect no data to remain on the memory card.</td>
<td></td>
</tr>
</tbody>
</table>

Once the set of security expectations was identified, the test application was developed and used to check whether these expectations held true.
5.2 Testing the Security Requirements

We initially designed the Data Analyzer test application to perform only two functions: write predefined data strings to text files, databases and key-value pairs, and delete one of the text files using the Android file delete APIs.

The application was configured to install to external storage by setting android:installLocation="preferExternal" within the AndroidManifest.xml file, and then was installed onto the SD card from within the Eclipse IDE.

5.2.1 Retrieving Application Code

Immediately after the application was installed, but before any data was written by the application, the SD card was opened as media using Windows File Explorer. There were two folders that appeared to be relevant to the test application:

- **Android**: This folder contained a subfolder titled data, which in turn contained folders for each application on external storage. Because the Data Analyzer application had not yet written any data to file at this point, there was no corresponding folder for it.

- **.android_secure**: This was a hidden folder (made visible by choosing the relevant option in File Explorer), which contained a single file com.proj.dataanalyzer-1.asec.

The only file we retrieved that was relevant to the application was the .asec file. Some background research uncovered that an ASEC file is an encrypted application container, which contains an application APK file. Such containers are created for applications whose target install location is the memory card, to protect them from being directly copied from the card [40].

ASEC files on the SD card are afforded a certain degree of protection. These containers are bound to the device that first installed them to the SD card. That is, they are encrypted using a key that is available on the host device. For this reason, merely copying the .asec file from an SD card would be of no use to an attacker since the appropriate decryption key must also be known. [40]. In this manner, the application code is protected while it resides on external storage which is unmounted from the device. This indicates that Req_Ext_01 is satisfied.

It should be noted that, when the application is executed, the container is mounted to /mnt/asec/<package_name>-1 within the device and the system loads the application from the new mount point [40]. The actual (unencrypted) APK file can be found from the location /mnt/asec/<package_name>-1/pkg.apk, and could then be reverse engineered to obtain the source code.

To verify this, we connected the test device to a PC and executed the command adb pull /mnt/asec/com.proj.dataanalyzer-1 from a Command Prompt, to find that the APK was indeed recoverable in this manner. This shows that the application code is only protected while it resides on the unmounted SD card.

That is, an attacker who is only in possession of an SD card containing an (encrypted) application will probably not be able to obtain the application source. However, if the attacker also has the device which originally installed that application to the memory card, then the source code can be obtained.
5.2.2 Writing and Analyzing Application Data Files

For the next part of the practical, we used the Data Analyzer application to write text files to private locations on external storage (SD card). The application also wrote to a database and a key-value pair at their default locations, using Android-recommended methods. All data structures had easily recognizable names and contents.

The SD card was then unmounted from the device, inserted into a reader and opened using Windows File Explorer. We observed that, despite the text files being written to private storage, they were still visible via Explorer. This appears to be in conflict with Android’s claim that such files would not be accessible as media, and signifies that Req_Ext_02 is not satisfied.

Another observation was that neither the database nor the key-value pair was present on the SD card. To ensure that they were not present in some hidden location on external storage, we imaged the memory card and analyzed the memory dumps (the process for obtaining memory images has been provided in Appendix B, along with samples of the images that were used for analysis).

However, no trace of these data structures could be found, even when searching through memory dumps. This confirmed that, even though it was an application on external storage that created the database and key-value pair, the data structures were nevertheless stored in internal device memory instead.

5.2.3 Identifying the Effectiveness of API Delete Methods

Before beginning an analysis of how effective data erasure is on external storage, it is important to note that, when writing to and deleting from SD cards, the Android platform may not have control over exactly how data gets written or erased. It is more likely that the SD card itself controls the exact mechanism, and that the Android operating system can only issue certain commands to the card interface. It may therefore be useful to see how SD cards normally handle data deletion.

Data Storage on SD Cards

SD cards are an example of memory modules which use NAND flash technology to provide non-volatile storage. NAND flash memory allows for reading and writing data in units of one block at a time, and data erasure requires all bits within the block containing the data to be set to 1 [41].

Because repeated erase cycles can cause the memory to become worn out and unreliable, memory cards typically employ wear-levelling techniques to distribute writes and erasures evenly across all cells [42]. They also do not immediately erase (i.e., overwrite with 1’s) data that has been deleted. Instead, the deleted file will have its pointer removed from the file allocation table, such that it becomes ‘invisible’ when searched for from within the file system. Such deleted entries can be identified using special markers, e.g., in File Allocation Table (FAT) file systems, deleted entries are marked with $0xE5$. [43]. However, the actual data remains on the memory card, until it is overwritten by a new file [44].

This is true for SD cards used within Android devices as well, and there are several tools available for recovering photographs and other media files which have been deleted from phone memory cards [44]. The Data Analyzer application was used to prove that application data can also be recovered in a similar manner.
**Practical confirmation**

We used the test application to write data strings to the SD card, and subsequently imaged the card. The application was then used to programmatically delete one of the files using the file deletion API provided by Android, and the card was imaged again.

The SD card was accessed via the File Explorer, and the file was shown to have been deleted. However, when the two memory card images were compared, it was observed that, while the file had been marked for deletion such that it was hidden from file explorers, the file’s contents had *not* been removed from the card (see Figure 6). That is, Req_Ext_03 was not satisfied.

![Image showing file deletion and data residues on SD Card](image)

**Figure 6: File Deletion and Data Residues on SD Card**

This result has implications for application developers in that any data that an application writes to external storage will likely remain in the SD card memory even after the developer assumes it to be deleted.

### 5.2.4 Recovering Data after Application Uninstallation

To identify whether data remained on device after application uninstallation, the test application was uninstalled from the device (i.e., the SD card), and the memory card was then accessed via File Explorer, to note that the application was no longer present on the card. The test device was also queried via the ADB Shell, but this too did not return any application information.

Next, we imaged the card and analyzed the images, to find that the entire application code, as well as all its data, still remained in memory. All that had been modified were pointers to the application and its files. This indicates that Req_Ext_04 is not fulfilled. From a developer’s perspective, this means that any data that an application writes to external storage may be recoverable even if the developer has control over user devices and is able to remotely uninstall applications or wipe data.
5.2.5 Obtaining Data Residues after Factory Reset / SD Card Format

The test application was reinstalled onto the SD card, and the card was imaged as before. A factory reset of the device was then performed, and SD card formatting was included as an option during the reset procedure. The reset procedure on the Android device specified that selecting this option “Erases all data on device’s SD card”.

Once the reset and format had completed, the SD card was imaged and analyzed again. We found that the formatting performed by the device was of a high level, where the application data was essentially hidden, but not actually removed (similar to the case where the application had been uninstalled).

This result denotes that Req_Ext_05 is not satisfied, and has serious implications for users who resell their devices with the SD card intact. Such users get a false sense of security after performing a factory reset, as they assume that they are selling a ‘clean’ device, whereas actually the reset process does not really erase any of their application data from the memory card.

5.2.6 Analyzing Data Longevity

As observed via the previous tests, when an application is uninstalled from an SD card, the area it occupied in memory is marked for overwriting but its code and data remain. When a new application is installed, it may overwrite the previous application code. However, this is not certain. It may therefore be useful to estimate for how long old applications and data can reside on external memory before being overwritten.

We conducted an experiment for this purpose, where eight applications (which will be referred to as Old App #1, Old App #2, ..., Old App #8) were downloaded from the Google Play store (which is the official marketplace for third-party Android applications) and installed onto the SD card.

For every application, the card was imaged once immediately after the application was installed, and once after the application was executed. In this manner, the following types of application content were obtained:

- Immediately upon installation: Application identifier; Application code.
- After application execution: Application data (including user input).

A brief description of these content types, as well as reasons as to why they may be of interest (especially to an attacker), have been specified in Table 2.

<table>
<thead>
<tr>
<th>Content</th>
<th>Description</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application identifier</td>
<td>A value of the form com.&lt;domain&gt;.&lt;app_name&gt;.</td>
<td>Being able to recover this value will provide an indication as to what applications have/had been installed on the SD card.</td>
</tr>
<tr>
<td>Application code</td>
<td>An encrypted block (or blocks) of code. It was assumed that only the blocks of code written immediately upon installation were application code.</td>
<td>While it is possible to pull APKs for installed applications from a device without the need for hexdumps, imaging a card can enable an attacker to recover (encrypted) code for applications that used to be present on the device, but have since been uninstalled. However, it is unlikely that this will have much value for an attacker unless the associated decryption key is also available.</td>
</tr>
</tbody>
</table>
This is the data written by the application. It was assumed that anything written to the card after application execution was application data*. Again, only data that was intended to be private was considered.

This is what is generally considered as most important in terms of user data security (where the data is actually sensitive), and is what a developer would likely be most concerned with protecting. Data could include application-specific information, user files, credentials, etc.

* Note that, while the experiment assumed that everything written to the card upon application execution was application data, in reality this could have included external libraries, etc., downloaded only when the application was executed, and not at the time of installation. This fact may have affected the results of the experiment.

The eight applications were then uninstalled from the SD card, and eight new applications (referred to as New App #1, New App #2, ..., New App #8) were installed sequentially. After every new installation, the memory card was imaged to identify how much information pertaining to each old application remained (i.e., how much information was not overwritten by the new application). More specifically, the analysis focused on how much residual information of each of the above three types (application identifier, code, and data) remained.

Next, we assigned approximate percentage values for the data residues. That is, taking application code as an example, if the entire application code of Old App #1 was overwritten by New App #1 (i.e., no code remained), a value of 0% was assigned. Similarly, a value of 50% indicated that half the application code remained, while 100% denoted that the application code remained in its entirety. These percentage values were tabulated and then used to produce graphs, depicting the trend in data longevity on SD cards.

The percentage values for residual application data have been provided in Table 3.

<table>
<thead>
<tr>
<th>Old App</th>
<th>New App #1</th>
<th>New App #2</th>
<th>New App #3</th>
<th>New App #4</th>
<th>New App #5</th>
<th>New App #6</th>
<th>New App #7</th>
<th>New App #8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old App #1</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Old App #2</td>
<td>99%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #3</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #4</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #5</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #6</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #7</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #8</td>
<td>Application was run, but no user data was entered during the test. Application itself appears to have written no data.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the amount of data that is recoverable will likely depend on multiple factors, such as the capacity of external storage, the sizes of the applications, and the amount of data written by the applications.

The resultant graph is depicted in Figure 7. The values and graphs corresponding to application identifier longevity and application code longevity have been made available in Appendix B.
It can be seen that, in most cases, all the data associated with an application remained on the device even though the application had been uninstalled and other applications had been installed afterwards. This shows that data written to an SD card may be vulnerable long after the application has been uninstalled.

5.3 Chapter Summary

This chapter focused on data security on external storage. It identified certain security expectations regarding external storage security, and then outlined the methods used to test these expectations. The practical experiments show that application files are easily accessible as media, even when marked as private, and that file contents are available in memory not only after the data is deleted, but even after the application is uninstalled or the memory card is formatted by the device. It was also observed that quite a lot of data remains on external storage even after a fair amount of new data has been written to memory.

The second part of the practical analyzes data security on internal storage, and has been described in the next chapter.
6 Investigating the Security of Data on Internal Storage

The Android operating system is far more stringent when it comes to internal storage security, since that is what the platform has the most control over. For this same reason, application developers will also have greater expectations regarding the protection given to their application data when stored in internal memory.

Possible expectations with respect to applications and application data stored on internal storage have been outlined in Table 4.

Table 4: Security Expectations for Application Data on Internal Storage

<table>
<thead>
<tr>
<th>Req_Int_01</th>
<th>Application package files should be protected on internal storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>It should not be possible to easily retrieve an application APK file from the device, since this may enable an attacker to reverse the file and obtain the application source.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req_Int_02</th>
<th>Application data should not be accessible outside the application that created them</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>It should not be possible to access an application’s private data (i.e., files, databases, and shared preferences) by any means, including file explorers, the ADB shell, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req_Int_03</th>
<th>Files that are deleted using Android API delete methods should not be recoverable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Similar to the requirement for external storage, it should not be possible to recover application data after it has been deleted.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req_Int_04</th>
<th>When an application is uninstalled, all traces of the application should be removed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This again is similar to the requirement for external storage, in that all application code and data should be completely removed from the device once the application is uninstalled.</td>
</tr>
</tbody>
</table>

While these requirements may be the same regardless of whether the target device is rooted or not, the degree to which they are satisfied will vary considerably depending on the root status of the phone. This chapter identifies the extent to which these requirements are satisfied on an unrooted device, while Chapter 7 explores how security may be affected once the device is rooted.

6.1 Assessing Application Package Security

We observed that it is possible to pull application packages from the internal storage of an Android device, even without root privileges. This observation was made in the following manner:

First, the Data Analyzer test application was modified to install to internal storage by specifying android:installLocation="internalOnly" within the AndroidManifest.xml file [19]. This attribute prevents the application from being installed to external storage, even should the user desire it.

Next, the exact application package location had to be identified. While it is generally known that user-installed applications reside somewhere within the /data/app/ folder in Android, the exact path to the package must be known in order to extract it.
An attempt was made to obtain this information by executing `ls -a` (which is a Linux command for listing all the contents of a directory) from within `/data/app` via the ADB Shell. However, this did not work on the unrooted test device due to lack of permissions (root permissions are required to be able to execute the `ls` command within the `/data/app` folder). Therefore, an alternate technique had to be found.

We ultimately identified the location to the path via the `packages.xml` file, which was found within the `/data/system` folder on the Android device. This file lists all the packages installed on the device, as well as the install path to the package [45], and is readable even without root privileges. We found that it was possible to pull `packages.xml` from the device using `adb pull`, and then view it using a standard text editor such as Notepad++. The file indicated that the test application was located at `/data/app/com.proj.dataanalyzer-1.apk` (as shown in Figure 8).

![Figure 8: Identifying Application Package Names](image)

The command `adb pull /data/app/com.proj.dataanalyzer-1.apk` was then executed to successfully obtain a copy of the application package from the Android device to the local machine, despite the fact that the test device did not allow root privileges.

This shows that extracting application files from the internal storage of an Android device is relatively straightforward, which is a violation of Req_Int_01.

**Protection Mechanisms against Application Extraction**

The experiment showed that it is possible to retrieve application APKs, even from an unrooted device, which may be cause for some concern. However, Elenkov [40] states that, in order to protect paid applications from being copied in such a manner, Google put forth a security mechanism known as ‘forward locking’ (or ‘copy protection’) for applications that were distributed through the official Google Play marketplace.

Developers publishing their application on Google Play had the option of marking it for Copy Protection. Older versions of Android then split the application into two, and installed a ‘public’ component (containing the `AndroidManifest.xml` and resource files) in `/data/app` and a ‘private’ component (containing the executable code) in `/data/app-private`. The part of the application that was held within `/data/app-private` was protected by filesystem permissions, and was therefore accessible only to the system user. [40]
Newer versions of Android do not use this method to protect application files [40]. The method employed from Android v4.1 for APK protection has been described in Section 8.1.1.

6.2 Accessing Application Data

Section 6.1 assessed the protection for application code on internal storage. This section deals with analyzing the protection afforded to application data.

For this purpose, we used the modified Data Analyzer application to write pre-defined data structures to default locations as before (but this time, only to internal locations). A number of different techniques were then employed to try and gain access to these data structures, as described in Sections 6.2.1, 6.2.2, and 6.2.3.

6.2.1 Attempting to Access Data via ADB Pull

Given that adb pull had been successful in pulling application packages from the internal memory of an unrooted device, the same method was used in an attempt to pull application data as well. As with the previous case, the absolute paths to files had to be identified first.

Unlike with application packages, information regarding application data is not made available on any public file within the system. However, it was possible to obtain information regarding the storage locations of application artifacts from various online sources [20], [46]. In particular, it was found that files belonging to an application are stored in /data/data/<package_name>/files/<file_name>, shared preferences (i.e., key-value pairs) are stored in /data/data/<package_name>/shared_prefs/MainActivity.xml, and databases are stored in /data/data/<package_name>/databases/<database_name> by default [46].

Because absolute paths require knowledge of the file (or database) names, the following attempts were made to try and obtain details about application file names (the attempts and outcomes have been depicted in Figure 9):

- Since it was known that files pertaining to the Data Analyzer application would be stored within the /data/data/com.proj.dataanalyzer/files folder, the ls -l command was executed from within that location in an attempt to list the names of all the files contained in the directory. However, the command did not succeed due to lack of permissions. That is, as with the /data/app folder, root privileges are required in order to list the contents of application data directories.

- An attempt was made to pull the entire application data directory via adb pull /data/data/com.proj.dataanalyzer. However, the command returned empty.

Assuming that an attacker is in possession of an unrooted device, and has no knowledge about the internals of the target application, he would probably not be able to determine the absolute paths to application files and databases. This is because the developer could name these artifacts in any manner of his choosing, and although the attacker could easily obtain the application package name, he would still not know the file names that were used. Therefore, in reality, an attacker would not be able to proceed any further.

However, for the purpose of the experiment, we used our knowledge of the Data Analyzer application to pinpoint exact file locations. But again, when the adb pull command was executed against these files on the device, the command failed due to lack of permissions.
6.2.2 Copying Data Using the `dd` Command

Since `adb pull` failed to return application data, the next step was to try and copy the files to some external location which has no access restrictions (such as the SD card), and then recover the files from there. The Unix `dd` command, which is built into Android, was used for this endeavor.

The Unix `dd` command is one that can be used to copy a file from one location to another [47]. However, it has a restriction that any file that is to be copied must be addressed by its absolute path name. For this, an attacker would have to know the locations and exact names of the files used within an application. As mentioned before, obtaining this information on a non-rooted device is difficult. Nevertheless, because we had knowledge of the files written by the Data Analyzer application, an attempt was made to copy them using `dd`. However, the command failed as the use of `dd` on Android is restricted to users with root privileges.

6.2.3 Imaging Internal Memory

The above methods (`adb pull` and `dd`) can be thought of as logical acquisition techniques, as they both attempt to gain access to files via the file system. Since neither method was successful, a software physical technique was utilized next to try and recover data directly from memory. That is, we attempted to mount the Android device’s internal storage onto the Ubuntu VM (as was previously done for SD cards), to be able to execute `hexdump` and obtain a memory image.

It was, however, not possible to mount the internal memory of the test device, due to a combination of two factors:

- The test device was running Android version 2.3: Prior to Android version 4.0, data transfer between a computer and an Android device was via the USB Mass Storage (UMS) option. This is a block-level protocol, which provides the host computer with direct access to physical storage blocks. This means that only one device can have the storage mounted at any given time. A dedicated storage partition is therefore required to be able to transfer data via UMS, since mounting the system storage would require that it be unmounted from Android first [48], which in turn would mean that the system would crash.
- The test device did not have separate partitions for data and system files: This meant that the data storage could not be mounted without also requiring that the system storage be unmounted from the device.
It is unclear as to whether it would be possible to mount and image an unrooted device even if it was running Android version 4.0+. If the device had a dedicated data storage partition, then it could be argued that such a partition would behave similar to an SD card, in which case it might be possible to image it. On the other hand, such a partition may be treated by the system as external rather than internal storage.

The results obtained from this part of the practical experiment gave rise to the conclusion that application data on an unrooted device cannot be recovered by the standard logical or software physical attacks. Therefore, security expectation Req_Int_02 is satisfied as far as unrooted Android devices are concerned.

It was not possible to test against expectations Req_Int_03 and Req_Int_04 as testing for these would require one of the above analysis methods to have been successful.

6.3 Chapter Summary
This chapter focused on testing the security of data that is stored in internal memory on an unrooted Android device. As with external storage, the chapter first specified a set of security expectations regarding application data security on internal storage. It then described the practical methods that were employed to test these expectations.

From the tests that were run against applications on internal memory, it was observed that application packages could be easily recovered from an unrooted device, but that application data remained protected by the access restrictions that were enforced by the Android platform.

The next step was to identify whether rooting an Android device would allow for access to greater amounts of data. The following chapter describes the part of the experiment that explored this option.
7 Examining Data Retrieval after Device Rooting

Chapter 6 identified that application data on an unrooted Android device is protected by kernel-enforced access restrictions. This chapter describes the practical experiment that consisted of rooting the test device and then analyzing whether such restrictions still hold.

It also describes the analyses that were performed on the rooted device to test against security requirements \textit{Req\_Int\_03} and \textit{Req\_Int\_04}, which could not be assessed on an unrooted device.

7.1 The Rooting Process

As mentioned in Section 2.3, rooting is simply the process by which root or ‘super’ privileges are enabled on an Android device.

The rooting process for the Samsung GT-S5363 test device was relatively straightforward, and consisted of the following steps [49]:

- An update archive was downloaded from [50] and loaded onto the SD card.
- The device was switched off and then booted up into recovery mode by pressing the Power, Volume Up and Home buttons simultaneously.
- The ‘Apply update from SD card’ option was selected (as shown in Figure 10).

![Figure 10: Rooting the Android Device](image)

The update file unlocked the bootloader and added the \texttt{su} binary to the system. To control which applications would have access to this \texttt{su} binary, the rooting process also installed an application known as \texttt{Superuser.apk} [51]. In addition, an executable known as BusyBox was installed. This is a toolkit containing common Unix utilities [52], which are not found on stock Android systems by default.

The device was then connected to the terminal and the ADB shell was loaded. We found that ADB Shell started up \textit{without} root privileges (this is identified by a $ prompt in the Shell). However, when \texttt{su} was typed into the shell, root privileges were obtained (indicated by a # prompt).
7.2 Application Data Analysis

7.2.1 Recovering Existing Application Data
Most sources specify that the part of the rooting process that involves unlocking the bootloader will wipe all applications and data from the device [6], [53], [54]. This would afford a certain degree of protection to applications, in that a device that is rooted by a malicious entity will at least not divulge details of previously installed applications.

However, we found that the Data Analyzer application and all its data remained on the device even after the bootloader was unlocked. Therefore, while it is uncertain as to whether this will hold true for all devices, it is still possible that an attacker who manages to gain access to an unrooted device might be able to root the device and recover old application data.

7.2.2 Accessing Private Application Files
Section 6.2 saw that private application data files could not be listed, viewed or pulled from the device on an unrooted Android platform. However, once root privileges were enabled, we found that it was possible to list all directory contents from within the application data directory via the ls -l command. This means that an attacker in possession of a rooted device will be able to learn the exact folders and file names created by an application.

Thereafter, we changed the permissions of data files using the chmod command, and then pulled them from the device. For example, chmod 666 shared_prefs/MainActivity.xml was executed from within /data/data/com.proj.dataanalyzer, to make the shared preferences file readable by all users, after which adb pull was used to successfully retrieve the file from the device. (Note that the chmod command cannot be executed on an unrooted device.)

The implication here is that all data (files, databases, and key-value pairs) belonging to currently installed applications are vulnerable on a rooted device, and that security expectation Req_Int_02 is no longer satisfied once the device is rooted.

7.2.3 Identifying the Effectiveness of API Deletion and Application Uninstallation
The test described in Section 7.2.2 showed that data belonging to all currently installed applications are always accessible on a rooted device. The next question was, therefore, whether data belonging to previously installed (and since uninstalled) applications could also be recovered from internal memory (as with SD cards). To be able to answer this question, a method was required, by which an image of the data directory within internal memory could be obtained.

Imaging internal memory using BusyBox
While it was not possible to mount and image the test device’s internal memory, as mentioned in Section 6.2.3, the installation of BusyBox during the rooting process provided an alternate method for obtaining a memory image. The hexdump command, which is not available in stock Android operating systems, is one of the utilities available in BusyBox’s suite of tools, and this was used for copying internal memory.
It was known that all data pertaining to non-system applications is available in the /data directory in internal memory. However, hexdump could not be used directly to copy the contents of this folder because the hexdump utility does not work on directories, but only on files (or on drives represented as files).

Therefore, to be able to image internal storage, first the memory partition containing application data had to be identified. In most cases, this information can be obtained by executing the command cat /proc/mtd [20]. However, this command returned a blank line when executed against the test device. This was due to the reasons mentioned in Section 6.2.3, in that the device in question did not have multiple partitions. Therefore, a different strategy had to be adopted.

When exploring the /proc directory, we uncovered a file titled mounts. The command cat /proc/mounts, when executed from within ADB shell, produced the following output:

```
# cat /proc/mounts
rootfs / rootfs ro,relatime 0 0
tmpfs /dev tmpfs rw,relatime,nodev,mode=755 0 0
devpts /dev/pts devpts rw,relatime,nodev,mode=600 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
tmpfs /mnt/asec tmpfs rw,relatime,nodev,mode=755,gid=1000 0 0
tmpfs /mnt/obb tmpfs rw,relatime,nodev,mode=755,gid=1000 0 0
/dev/stl9 /system rfs ro,relatime,vfat,log_off,check=no,gid/uid/rwx,iocharset=cp437 0 0
/dev/stl10 /cache rfs rw,nosuid,nodev,relatime,vfat,llw,gid/uid/rwx,iocharset=cp437 0 0
/dev/stl16 /mnt/.lfs j4fs rw,relatime 0 0
/dev/stl11 /data rfs rw,nosuid,nodev,relatime,vfat,llw,check=no,gid/uid/rwx,iocharset=cp437 0 0
/dev/block/vold/179:1 /mnt/sdcard vfat rw,dirsync,nosuid,nodev,noexec,relatime,uid=1000,gid=1015,fmask=702,dmask=702,allow_utime=0020,codepage=cp437,iocharset=iso8859-1,shortname=mixed,utf8,errors=remount-ro 0 0
```

From the output, we determined that the /data folder was mapped to /dev/stl11 within internal storage. The following command was then executed to copy this section of memory to a text file on the SD card for further analysis: busybox hexdump --C --v /dev/stl11 > /sdcard/<file_name>.txt. This method was used to obtain memory images and analyze the contents of internal storage in a manner similar to that for SD cards.

Memory image analysis

A file was written to internal storage and then deleted using the standard Android APIs. The /dev/stl11 block was imaged after both actions, and the two hexdumps were compared. We found that, with internal storage, the data did not remain within memory after deletion, which means that security expectation Req_Int_03 is apparently satisfied. However, remnants of some file names were observed in the memory dumps. (All relevant hexdumps are available in Appendix B)
However, it is not entirely clear whether the removal of data was actually due to any inherent security of the platform. We believe it could also have been due to the fact that the test device had very limited internal memory, thereby necessitating immediate removal of deleted content.

The next part of the experiment consisted of uninstalling the application and analyzing the memory images again. We observed that while most of the application code was erased, not all of it was, and that some code segments remained in memory even after the application was uninstalled. Further, application identifiers also remained on internal storage. The conclusion here is that Req_Int_04 is not completely fulfilled.

An observation regarding the residual application code was that it was apparently the same sections of code (the parts which named the layout files that were used within the application) that always remained. In addition, it was noted that these sections were immediately erased with the installation of a new application.

7.2.4 Other Observations

**Decryption Keys for Applications on External Storage**

It was observed previously that applications that are installed on an SD card are placed within containers that are encrypted using a key which is available on the host device. This encryption key is not publicly visible on unrooted Android devices. However, when the test device was rooted, an exploration of the system files led to the discovery of the key, which we extracted from /data/misc/systemkeys/AppsOnSD.sks.

Theoretically, this means that application code on SD cards that was once protected can now be decrypted and reversed. This may not seem significant, since the APK files of applications installed on an SD card can always be pulled from the device itself by mounting and executing the application, as mentioned in Section 5.2.1. However, given that it is possible to recover from an SD card the code for previously-installed applications (as detailed in Section 5.2.6), which cannot be mounted onto the device, the availability of the encryption key now makes the decryption of even these applications theoretically possible.

7.3 Chapter Summary

This chapter dealt with two main elements: It described the rooting process and analyzed data security on rooted Android devices. It also made use of the newly-obtained root privileges to analyze generic application data security on the internal storage of an Android device, which could not be tested before due to lack of privileges.

From the experiments that were conducted, it was possible to conclude that API deletion methods and application uninstallation appear to involve complete erasure of data from internal memory, in contrast to external storage (this is regardless of whether or not the device is rooted). This could serve to reduce exposure to attack.

However, it was also observed that rooting is a relatively simple process which can immediately render an application’s data vulnerable. Rooting also exposes a lot of other sensitive information, such as SD card application encryption keys, which can expose application code and data to even more attack.
8 Newer Versions of Android

This project made use of an older, albeit widely used, version of Android for its analysis, and it could be argued that the results may not be as applicable for newer platforms. This chapter therefore analyzes the enhancements in later versions of Android and their impact on application data security.

8.1 Performance and Security Enhancements

8.1.1 Forward-Locked Applications

Sections 5.2.1 and 6.1 introduced the concept of ‘forward-locked’ or ‘copy-protected’ applications. This concept came about with Android version 2.2, when applications were first allowed to be installed on SD cards. Because external storage may contain paid applications, in order to protect such applications from simply being copied from the SD card, Android secured the APK files by creating encrypted containers on the memory card and placing the application files inside them. [40]

Applications on internal storage were protected by a different method in older versions of Android, where they were split into two parts and protected by different access permissions. However, because rooting could easily overcome access restrictions, this method was deprecated. The newest version of Android (v4.1 upwards) employs a technique similar to that which is used for external storage. That is, when an application is forward-locked, an encrypted container is created in /data/app-asec and the APK file is stored within the container. When the application is run, it is mounted to /mnt/asec and executed. [40]

This does not actually mean that an attacker will not be able to access an application on internal storage with Android 4.1+. A rooted device will enable an attacker to retrieve both the forward-locked APK [40], as well as the encryption key (as demonstrated in Section 7.2.4). However, it will make the process slightly more difficult than with older versions of Android.

8.1.2 Exclusive-Use Keys and Hardware Credential Storage

Android version 4.3 introduced a keystore provider, which is a means for securely storing cryptographic keys, as well as information regarding key owners [55]. Keys can be generated by an application and stored within the keystore, and such keys are available only to the owning application, but not to any others (not even to root). These keys are protected by being encrypted with a master key, which itself is encrypted with a system key. The system key is derived from the user password for the device. [56], [57].

This could therefore be considered a special case of password-based key derivation, where the device rather than the application is in charge of requesting the key from the user. However, there is no guarantee that all end users will use a key to lock their devices, which means there is a certain degree of uncertainty as to the security of the key store.

This version of Android also launched hardware-backed key storage. That is, if some form of hardware credential storage - such as a Secure Element, Trusted Platform Module or TrustZone - is used to house cryptographic keys, those keys will be bound to hardware and will no longer be exportable, even on a rooted device. [58], [59]
This is a significant enhancement which could greatly improve the security of application data, since developers can now encrypt on-device application information and then store the key in such a manner that it is not accessible even if root permissions are available on the device.

However, we mention here the caveat that not all smart devices have embedded secure hardware storage elements at present, which means that this feature may not be as relevant to current applications. Nevertheless, this feature will undoubtedly be useful in future.

### 8.1.3 Full-Device Encryption

Version 4.0 was the first Android platform targeting smartphones to feature on-device encryption. This means that users now have the option of password- or PIN-protecting all contents on their device. Any attacker getting hold of the device would only be able to recover scrambled data, unless he also had knowledge of the user secret [60].

While this mechanism can effectively protect device-resident data, we make the observation that developers do not have control over whether or not their users make use of this feature. Therefore, applications cannot be developed with any degree of certainty that they will be protected.

The only way that a developer could be sure of this protection being applied is in a corporate setting, where all target devices could potentially be managed using a centralized management solution.

### 8.1.4 Reduced Root Attack Surface

Android versions 4.2 and 4.3 have incorporated a number of enhancements to try and reduce the likelihood of root attacks [61], [62]. Some such enhancements are restricting Android applications from executing setuid programs, and removing all setuid/setgid programs [62].

Because application data on a rooted device is far more vulnerable than that on an unrooted device, any reduction in the likelihood of root attacks will serve to make data recovery more difficult for an attacker.

### 8.1.5 The Android Backup Feature

With version 4.0 upwards, Android has provided a feature which will enable users to backup application data on non-rooted devices. This means that anyone in possession of an Android phone with the latest firmware will be able to recover the entire application data, including databases, etc. which are meant to be private to the application. This data will be available in Android Backup (.ab) format [63], and apparently can unpacked using an open-source Java program [64].

We believe this suggests that some features in newer versions of Android may actually lead to reduced security. However, it should be noted that this feature can be disabled by a developer for his application.
8.1.6 Support for Multiple Users
Version 4.2 of Android announced support for multiple users on a single device. That is, an application published by a developer can be installed and used on a single device by multiple users. Android claims that the system will ensure that no user will have access to another user’s application data [65].

However, assuming that this protection is enabled through access permissions, it is our understanding that rooting an Android device could potentially make available the application data of all users to an attacker (where the attacker could very well be one of the users).

8.2 Chapter Summary
This chapter introduced multiple features that are available in newer versions of Android. Some features, such as device encryption and forward-locked applications, have the potential to make application data more secure, provided they are used correctly. On the other hand, it is our view that some enhancements that have been introduced to improve or augment functionality may in fact result in a decrease in the level of security that is afforded to application data.
9 Securing Application Data

The results from the practical experiment, as described in Chapters 5 to 7, confirm the fact that application data on an Android device is potentially vulnerable for a large part of the time, especially when the application is installed on external storage or when the host device is rooted.

Developers, for the most part, do not have control over their users’ phones (except possibly in the case of corporate-owned devices). This means that they have no way of knowing whether or not their application will end up being installed on a rooted device or be exposed to an insecure environment. We believe the safest approach is therefore to develop applications under the assumption that all target devices are insecure and that the resident data is always vulnerable.

To ensure the protection of device-resident data, security should be considered during the design stage of an application. We propose a two-phase approach for this purpose as follows:

Phase 1: Data classification

Data classification schemes are commonly used in defense organizations and even occasionally in the corporate sector, and can be extended to the mobile context as well. As mentioned by Gunasekera in [66], before incorporating mechanisms to protect application data, it is important to know exactly what type of data the application will be handling, and to determine the sensitivity or criticality of such data. Only then can appropriate protection techniques be decided upon.

We suggest a possible data classification scheme in Table 5:

Table 5: Possible Classification Scheme for Application Data

<table>
<thead>
<tr>
<th>Classification</th>
<th>Description</th>
<th>Possible Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highly Sensitive</td>
<td>Extremely critical information that must be protected using the highest level of security.</td>
<td>Financial information, corporate data, banking credentials, etc.</td>
</tr>
<tr>
<td>Sensitive</td>
<td>Data that is not critical, but which must be protected nevertheless.</td>
<td>User preferences, configuration information, application code.</td>
</tr>
<tr>
<td>Publicly Viewable</td>
<td>Data that can be viewed by anyone and which does not normally warrant any special protection.</td>
<td>Unrestricted-access articles, open source application code.</td>
</tr>
</tbody>
</table>

Obviously, the classification scheme and the contents within each category will differ between organizations and even from application to application, and will depend on many factors, such as the type of application, the target audience and the deployment environment.

Also, it is possible that not every application handles data at all classification levels. Indeed, it is probable that some applications deal solely with ‘Publicly Viewable’ data.

Phase 2: Selection of protection mechanisms

Once data classification has been performed, suitable security controls can be decided upon for implementation. Building up on the concept of Defense in Depth [5], we believe a multifaceted approach to security should be adopted, rather than to apply a single protection mechanism to secure all application data.
We categorize various protection mechanisms into levels as ‘Basic’, ‘Medium’ and ‘Strong’ depending on the ease of deployment and the security provided (Figure 11).

![Figure 11: Levels of Protection](image)

Some of the mechanisms specified depicted in Figure 11 may only protect certain types of data. For example, obfuscation of application code will have little to no effect on data that is generated or handled by the application during its execution. If both code and data need to be protected, then separate techniques will have to be utilized.

It should also be noted that the ‘security’ provided by many of these methods can be defeated or circumvented. However, they are useful in achieving the aim to add obstacles such that attacker will at least be forced to expend a considerable amount of time and/or resources in order to overcome the protection.

Note also that this does not purport to be an exhaustive catalogue of all possible controls. Nor is it suggested that all these mechanisms should to be applied to a single application. Instead, the recommendation is to select controls as appropriate based on the classification of the data handled by the application. For example, data that is classified as ‘Highly Sensitive’ should be protected by mechanisms that afford at least a ‘Medium’ level of security (although ‘Strong’ levels would be preferred, wherever possible). Similarly, data that is labelled as ‘Sensitive’ can be protected by a combination of ‘Basic’ and ‘Medium’ security techniques.

Where a single application handles multiple classifications of data, a decision may need to be made regarding whether to apply the highest level of protection to all data or whether to apply stronger controls to only the most sensitive data. This decision may be made on the basis of application performance considerations, the affect on end user experience, and the ease of deploying multiple controls.

Each of the protection mechanisms in Figure 11 has been briefly discussed in the subsequent sections.
9.1 Basic Protection Mechanisms

Obfuscate application code
Obfuscation is a mechanism by which application code is rendered apparently unintelligible, usually with the aim of hiding application logic or deterring reverse-engineering [67]. The Android build system contains a tool named ProGuard which optimizes code by removing unused code segments, and obfuscates code by renaming classes and methods with names that are more difficult to read and recognize [68].

This means that even if an attacker reverse engineers an application, the resultant code will be harder to understand. An added advantage is that obfuscation will not hinder the performance of the application in any way. However, it is important to note that obfuscation does not really secure code, and a determined attacker will be able to obtain an idea of how the application works with some effort.

Prevent installation on older versions of Android
As observed in Section 8, newer versions of Android have security capabilities which may afford increased protection for application data. A developer could therefore choose to specify a target API version for his application which enables installation on only the newest platforms. However, certain older versions of Android do have widespread usage even now, and the potential customer installation base of an application would reduce by a significant percentage if only new versions of the operating system were supported.

Prevent application installation and sensitive data storage on external storage
This project demonstrated that data on external storage is far more vulnerable than that which is stored in internal memory, especially in the case of unrooted devices. For this reason, where an application handles sensitive data, it may be advisable to prevent the application from being installed on external storage.

However, as mentioned before, limited internal storage is what prompts developers to allow applications to be installed on SD cards, and preventing this capability may serve to alienate a subset of the application’s users. If this is a legitimate concern, then the developer should at least store all sensitive application data within private internal storage and allow only the application code and publicly-viewable data to be installed on the SD card.

Disallow application data backup
As mentioned in Section 8.1.5, newer versions of Android enable users to backup their applications and application data via ADB. This data can then be unpacked and analyzed by an attacker. Using android:allowBackup="false" in the Manifest file can prevent an attacker from exploiting this feature to obtain application data. However, preventing backups may be viewed as an inconvenience by legitimate users.

Prevent application execution on rooted devices
The results from the practical attempts to retrieve application data from internal storage confirm that the likelihood of data recovery is much greater when the device has been rooted (see Chapters 6 and 7). Rooting essentially gives the holder of a device full control over and access to all data that is stored on the device. For this reason, application developers may desire to prevent their applications from being executed on rooted platforms.
To prevent applications from being installed on rooted Android devices and thereby exposing potentially sensitive data, one technique is to first run certain checks to find out whether the host device is rooted. Once this has been determined, a decision can be made as to whether or not the application is to be allowed to continue executing. We incorporated such checks into the test application (for which the code is available in Appendix A), and found that they were successful in detecting that the device had been rooted.

**Bypass (hiding root)**

While there are techniques for checking whether Android devices are rooted, there are also methods by which such checks can be fooled. There are, in fact, many applications in Google Play which effectively 'hide root'. We installed an application named *Hide Rooting Lite* [69] on the test device, and observed that this application was able to fool the root-checking mechanisms employed in the *Data Analyzer*. According to user reviews on Google Play, this application works with many (but not all) other applications that have root-prevention methods in place [69].

Therefore, even if a developer does utilize some means to prevent his application from being executed on a rooted device, he should nevertheless not assume that his application data is safe, and should instead incorporate additional controls to safeguard such data.

### 9.2 Medium-level Security

**Implement secure data deletion methods**

Section 5.2.3 identified that application data residing on external storage will still be recoverable from memory even after it is deleted using standard Android file deletion methods. For this reason, developers should implement their own secure data deletion methods within their applications. When application data or files are no longer needed, the file contents or data should be overwritten before deleting, such that even if the deletion leaves behind residues, the data residues will not provide an attacker with any useful information.

**Force timeout of stored data**

If an application does store sensitive data on an Android device, then implementing a validity period for the data (after which it is securely removed from the device) can reduce its exposure to attack. If the data needs to be re-downloaded when the application is used at a later stage, then the application user could be prompted for credentials first, to ensure that the device is still in the hands of a legitimate user).

It should be noted that this technique may hinder the smooth use of the application. Also, the data should be protected during its lifetime on the device (i.e., during its validity period), since it would be vulnerable to attack for that duration.

**Encrypt on-device data**

If there is a legitimate need to store sensitive data on the device, such data should always be encrypted using a strong, standardized algorithm. The issue then arises regarding the storage of the key. Some developers store the key within the application code and rely on obfuscation to hide it. However, this is in no way an appropriate mechanism for key storage, as an attacker *will* be able to recover the key with some effort.
Some possible options for data encryption have been outlined below:

Public-key cryptography
Asymmetric methods have been suggested as a potential solution for protecting device-resident data [5], by encrypting application data with the public key and then storing both the key and the encrypted data on-device. The nature of public-key cryptography would even allow for the public key to be embedded within the application code. The private key, on the other hand, would have to be stored either in a secure location on the device or on an external server.

However, to decrypt data, the private key would have to be downloaded from the server to the device. There would therefore be a small window of opportunity within which an attacker might be able to acquire the private key. Also, this method would result in a lot of processing and communication overheads.

System-generated keys
Allowing the system to generate a symmetric key and store it in internal memory is one possible strategy that could be adopted. To test the security of encryption using device-generated keys, we modified the Data Analyzer test application to include encryption, and then used it to encrypt some user-input string (the relevant code has been provided in Appendix A).

Figure 12 depicts the basic cryptographic function, which consists of two parts: key generation and encryption. Note that $m$ denotes the plaintext message, $[m]$ symbolizes the message in bytes, and $[c]$ denotes the encoded bytes.

![Figure 12: Encryption Function in the Data Analyzer Application](image)

A test string was encrypted, and the /data folder of the device was then searched, via several means (such as file exploration via ADB Shell, using the grep command to search for key strings, and searching through hexdumps).

We found that it was possible to recover the encrypted string, but could not identify the encryption key. However, this could simply have been because the key is well-hidden within the system, possibly in a directory other than /data.

Even though the experiment did not yield the location of the key, relying on such keys to protect application data may not be wise, since the key must be stored somewhere on the device. In fact, Android itself specifies that “encrypting data with a key stored on-device does not protect the application data from root users” [6].
Android keystore
Another option for symmetric key storage is the use of the Android key store, which was introduced in Android v4.3. However, as mentioned in Section 8.1.2, this method for key storage is only secure if there is some guarantee that the end user will password-protect his device.

Key derivation
The currently accepted method is to use key derivation techniques. That is, the encryption key for protecting application data is derived from some user input [70]. This protects against the possibility of encryption being used on a device which is not protected by a user password.

Key derivation has the negative side effect of requiring the user to remember and protect some sort of password/passphrase, and to input it each time the application is executed. Another consequence is that the performance of the application will suffer slightly [70].

9.3 Strong Security Mechanisms

Use hardware security
As seen in the preceding section, the biggest issue with encrypting device-resident data is that of finding a suitably secure location in which to store the decryption key. One way of securing data even on rooted devices is via a hardware-based solution [6]. Section 8.1.2 referred to the capability of Android 4.3 to support hardware-bound credential storage for applications, in such a manner that the keys would not be exportable even if root privileges are available on the device. However, it also mentioned that, at present, most Android devices do not have secure storage elements in place.

If, in future, NFC gains widespread use, or if some other form of embedded hardware storage is made available on mobile devices, then such hardware modules would be the ideal storage locations for cryptographic keys. In fact, if some form of security module management infrastructure was in place, then even sensitive application data could be stored within the module (rather than just keys).
10 Conclusions

This project dealt with the subject of application data security on Android devices. It utilized a combination of theoretical research and practical experimentation to assess the actual protection afforded to application data, and analyzed separately the security of data on internal and external storage, and on rooted and unrooted devices.

This final chapter assesses the outcomes of the project against its defined objectives, and outlines possible areas for future work.

10.1 Assessment against Objectives

*Study the Android security model, and find out Android’s mechanisms for protecting device-resident application data.*

The initial stage of the project involved analyzing the Android architecture design and studying the mechanisms by which the platform safeguarded application data. It identified that, because Android is built upon a version of the Linux kernel, it utilizes Linux-based methods for providing application data safety. In particular, the platform uses a permission-driven model and enforces application sandboxing to prevent data belonging to an application from being interfered with. This protection is further ensured by preventing privileged-level access to the system.

*Identify whether protected application data can be retrieved from an Android device.*

We identified via practical means that any application data that is placed on external storage can always be retrieved. This, however, was expected since the Android documentation itself indicates that all data stored in such locations will be considered to be world-readable.

In contrast, tests done on internal storage failed to recover application data from the test device when the phone was in its original unrooted state. Nevertheless, once the device was rooted, it was found to be possible to recover all existing and new application data.

*Determine whether application data residues can be recovered in the discrete events of programmatic data deletion, application uninstallation, and device rooting.*

Tests executed against data on an SD card revealed that application data remains in external storage for long after it is assumed to be deleted. Such data remains on the memory card after the owning application itself has been uninstalled and even after the card has been formatted by the device.

On the other hand, the Android platform appears to be more stringent when it comes to secure deletion of application data on internal memory. We found that application data appears to be completely erased from memory once the application is uninstalled. However, there is some uncertainty as to whether this was simply due to the limited internal memory available on the test device. The practical tests also found that application code was not completely removed upon uninstallation, with some instances of code residues being observed.

Further, we found that pre-existing application data could be recovered in its entirety after the test device was rooted, despite claims that data on internal memory is wiped during the rooting process.
Examine the extent of application data that can be recovered when the phone is rooted compared with an unrooted device.

The practical experiment suggested that data stored in internal memory is protected for as long as the Android device is not rooted. However, it was found that rooting is a fairly simple process, and that it may expose all existing application data to attack. Also, it was identified that all the data of any application that is installed after a device is rooted will be recoverable.

Propose methods by which application data can be secured.

This project proposed a two-phase approach to protect application data. A data classification process was recommended as a first phase, after which protection mechanisms could be selected in a manner which optimized security and performance. A lot of focus was placed on the use of encryption as a means for securing application data, with a recommendation that the encryption key itself be protected by being derived from a password or by being stored on a hardware security module.

10.2 Future Work

There are several avenues down which further research can be conducted in this arena. Some involve validating the test results for a broader range of devices, while others could analyze the extent to which the suggested security measures are actually successful.

For example, the experiment conducted in this thesis can be performed against different devices with higher capacity internal memory, to identify how many of the results still hold true. The tests could also be executed against devices running newer versions of the Android platform, to assess whether the security enhancements in newer operating systems do actually prevent the possibility of data recovery.

Research could also be done to see how feasible it is to recover a device-generated encryption key by combining device memory imaging techniques with password guessing or brute forcing attacks.
Bibliography

References


Additional Sources


Appendix A Test Application Development and Sample Code

This appendix describes details regarding the test application (the Data Analyzer) that was developed during the practical element of this project. In particular, it provides a rationale behind certain development decisions, specifies the environment and tools that were used during development, and describes the application structure.

This appendix also provides code segments which achieve some significant functionality, but does not contain the entire application code.

A.1. Selecting a Target Platform for Development

Android comes in different flavours, starting with the first commercial version which was released in 2008. All versions following v1.5 were assigned a codename in alphabetical order, and a single version may encompass multiple application programming interface (API) levels, as shown in Table 6. Each of these versions features bug fixes and optimizations, or even major design and/or performance improvements, over its predecessor [71].

<table>
<thead>
<tr>
<th>Version</th>
<th>Codename</th>
<th>API Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android 1.0</td>
<td>&lt;No codename&gt;</td>
<td>API level 1</td>
</tr>
<tr>
<td>Android 1.1</td>
<td>&lt;No codename&gt;</td>
<td>API level 2</td>
</tr>
<tr>
<td>Android 1.5</td>
<td>Cupcake</td>
<td>API level 3</td>
</tr>
<tr>
<td>Android 1.6</td>
<td>Donut</td>
<td>API level 4</td>
</tr>
<tr>
<td>Android 2.0</td>
<td>Éclair</td>
<td>API level 5</td>
</tr>
<tr>
<td>Android 2.0.1</td>
<td>Éclair</td>
<td>API level 6</td>
</tr>
<tr>
<td>Android 2.1</td>
<td></td>
<td>API level 7</td>
</tr>
<tr>
<td>Android 2.2–2.2.3</td>
<td>Froyo</td>
<td>API level 8</td>
</tr>
<tr>
<td>Android 2.3–2.3.2</td>
<td>Gingerbread</td>
<td>API level 9</td>
</tr>
<tr>
<td>Android 2.3.3–2.3.7</td>
<td></td>
<td>API level 10</td>
</tr>
<tr>
<td>Android 3.0</td>
<td>Honeycomb</td>
<td>API level 11</td>
</tr>
<tr>
<td>Android 3.1</td>
<td></td>
<td>API level 12</td>
</tr>
<tr>
<td>Android 3.2</td>
<td></td>
<td>API level 13</td>
</tr>
<tr>
<td>Android 4.0–4.0.2</td>
<td>Ice Cream</td>
<td>API level 14</td>
</tr>
<tr>
<td>Android 4.0.3–4.0.4</td>
<td></td>
<td>API level 15</td>
</tr>
<tr>
<td>Android 4.1</td>
<td></td>
<td>API level 16</td>
</tr>
<tr>
<td>Android 4.2</td>
<td>Jelly Bean</td>
<td>API level 17</td>
</tr>
<tr>
<td>Android 4.3</td>
<td></td>
<td>API level 18</td>
</tr>
</tbody>
</table>

It is important to note that, because many older devices do not support newer versions of the platform, and even users with newer devices may opt not to update their existing operating system, the Android market is highly fragmented. That is, almost every one of the above versions is likely deployed on at least a few devices worldwide. As a result, selecting any one platform for analysis can be quite difficult.
The test application was installed on a device running Android v2.3.6 (Gingerbread). The rationale behind selecting this particular platform for the experiment, despite the availability of several newer versions of Android, is that v2.3 is the most widely used operating system as of June 10, 2013 with a spread of approximately 40% [72]. That is, a large percentage of users are accessing applications on devices that run this version of the platform. This means that the experiment results will be relevant to the largest subset of users.

A.2. Tools Used for Development
The following environment, tools and devices were used during the development of the Data Analyzer application:

*Application development environment and testing tools*

- Operating System: Windows 7 SP1 64-bit.
- Java Development Kit v6u39 (version 6, update 39).
- Android Developer Tools (ADT) Bundle:
  - Eclipse Integrated Development Environment (IDE) with the ADT plug-in v22.0.1.
  - Android Software Development Kit (SDK) Tools.
  - Mobile Device Emulator (to test the application prior to deploying on device):
    - Android Virtual Device (AVD) Manager, with AVD running Google API level 10.

*Application deployment device*

- Device Model: Samsung GT-S5363.
- Android Version / Kernel Version: 2.3.6 / 2.6.35.7.
- Storage Capacity: 180MB internal RAM; 2GB external (removable SD card).

A.3. Application structure
The Data Analyzer application is fairly simple, with all functionality contained within a few Java classes. Figure 13 depicts the application structure, and is followed by a brief overview of the different classes.
**MainActivity.java**

This is the main class file, and contains the code that is executed on application startup. Further, it contains code for creating and writing predefined character strings to a number of data structures (text files, databases, and key-value pairs).

MainActivity.java also makes use of other classes to perform various functions such as manipulating databases, enforcing root-execution-prevention, and data encryption.

**CheckRoot.java, ExecShell.java**

These two classes were sourced from [73], and together perform the root checking function. In particular, they check for three elements which could be indicative of a rooted device:

- The presence of the su binary.
- The availability of the SuperUser.apk.
- Test keys within build tags.

The code for ExecShell.java has been given below:

```java
/* Code by Kevin Kowalewski on StackOverflow.  
* Ref: http://stackoverflow.com/questions/1101380/determine-if-running-on-a-rooted-device?rq=1 */
public class ExecShell {

    public static enum SHELL_CMD {
        check_su_binary(new String[] {"/system/xbin/which","su"}),
        
        String[] command;

        SHELL_CMD(String[] command){
            this.command = command;
        }
    }

    public ArrayList<String> executeCommand(SHELL_CMD shellCmd){
        String line = null;
        ArrayList<String> fullResponse = new ArrayList<String>();
        Process localProcess = null;

        try {
            localProcess = Runtime.getRuntime().exec(shellCmd.command);
        } catch (Exception e) {
            return null;
        }

        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(localProcess.getOutputStream()));
        BufferedReader in = new BufferedReader(new InputStreamReader(localProcess.getInputStream()));

        try {
            while ((line = in.readLine()) != null) {
                fullResponse.add(line);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        return fullResponse;
    }
}
```
The code for `CheckRoot.java` has been given below:

```java
public class CheckRoot {
    public static boolean isRooted() { // Method checks for root in three different ways
        if (suBinaryPresent()) { return true; }
        if (superuserApk()) { return true; }
        if (testKeys()) { return true; }
        return false;
    }

    public static boolean suBinaryPresent() { // Checks for the presence of the su binary
        if (new ExecShell().executeCommand(SHELL_CMD.check_su_binary) != null) { return true; }
        return false;
    }

    public static boolean superuserApk() { // Checks for the Superuser.apk
        try {
            File file = new File("/system/app/Superuser.apk");
            if (file.exists()) { return true; }
        } catch (Exception e) { }
        return false;
    }

    public static boolean testKeys() { // Checks whether build tags contain test keys
        String buildTags = android.os.Build.TAGS;
        if (buildTags != null && buildTags.contains("test-keys")) { return true; }
        return false;
    }
}
```

This method returns true if evidence of rooting is found and false otherwise. However, given that different rooting mechanisms employ different techniques, these three tests alone may not detect every instance of rooting.

**RootPreventActivity.java**

This class contains the code that is to be executed on discovering that the target device is, in fact, rooted. It contains no other functionality apart from drawing the application layout and specifying to the user that the application cannot proceed further.

The `CheckRoot` method is called from within `MainActivity`, to determine which execution path the application should take.
If the device has not been rooted, the normal application screen is displayed. Otherwise, a message is displayed indicating that the application cannot be executed further (Figure 14).

**Figure 14: Application Screen on Unrooted vs. Rooted Device**

DatabaseHandler.java
This class contains the code for handling SQLite databases, and is adapted from [74]. When called (from within MainActivity.java), this class creates a database and a database table, and immediately inserts a row of predefined data into the table columns.

Encrypt.java
This class performs certain cryptographic functions and was adapted from [75] and [76]. It first generates a 128-bit key, which is automatically seeded using system entropy. It then converts the plaintext string to bytes and encrypts the bytes using the Advanced Encryption Standard (AES) algorithm and the key that was previously generated.

Decryption functionality was not implemented, since the purpose of the code was merely to analyze whether the encryption key was recoverable.
The code that was used to achieve the encryption functionality is as follows.

```java
public class Encrypt {
    static final String LoggingTAG = "EncryptionLogs";
    // Set up secret key spec for 128-bit AES encryption and decryption
    public static SecretKey appKey = null;

    public static String encryptString(String stringToEncrypt)
    {
        // First convert string to byte array
        byte[] inputBytes = stringToEncrypt.getBytes();
        Log.d(LoggingTAG, "Byte array = " + inputBytes);
        // Encode the original data with AES
        byte[] encodedBytes = null;

        /* Generate cryptographic key
        * Code adapted from Android Developer Blog (original author: Trevor Johns)
        * Ref: http://android-developers.blogspot.co.uk/2013/02/using-cryptography-to-store-credentials.html */
        final int outputKeyLength = 128;
        SecureRandom secureRandom = null;
        try {
            keyGenerator = KeyGenerator.getInstance("AES");
        } catch (NoSuchAlgorithmException e1) {
            Log.e(LoggingTAG, "no such algo = " + e1);
        }
        keyGenerator.init(outputKeyLength, secureRandom);
        appKey = keyGenerator.generateKey();

        try {
            Cipher c = Cipher.getInstance("AES");
            c.init(Cipher.ENCRYPT_MODE, appKey);
            encodedBytes = c.doFinal(inputBytes);
            // Log.d(LoggingTAG, "encodedBytes = " + encodedBytes);
        } catch (Exception e) {
            Log.e(LoggingTAG, "AES encryption error");
        }

        String encryptedString = Base64.encodeToString(encodedBytes, Base64.DEFAULT);
        // Log.d(LoggingTAG, "encrypted string = " + encryptedString);
        return encryptedString;
    }
}
```

Note that here, every time the `Encrypt` method is called, the key is regenerated. Obviously, this would not be practical for a real application, since decryption would require the same key to be persistent. However, for the purpose of this experiment, the code above sufficed.
Appendix B Memory Imaging and Analyses

Memory imaging played a large part in the practical experiment for this project. This appendix details the methodology that was followed in order to image the external memory of the test device. It then catalogues the results and observations of the imaging process.

The internal memory imaging process has been detailed within the main text, and has therefore not been replicated here. However, relevant results that were obtained through the internal imaging process have been discussed.

B.1. Obtaining Memory Images - External Storage

The following process was utilized for imaging a memory card:

- A VirtualBox virtual machine (VM) was set up on the host PC with Ubuntu as the guest operating system. The VM had the following configuration:
  - Base Memory: 1024 MB.
  - Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX.
  - Storage: 20.00 GB fixed SATA.
  - Network Adapter: Intel PRO/1000 MT Desktop (NAT).
- The USB port corresponding to the SD card reader on the host machine was configured to mount to the VM whenever the VM was available.
- A shared folder was configured to transfer files between the Windows host and the Ubuntu guest systems.
- The SD card was inserted into the reader and the partition table was listed for the system by executing `sudo fdisk -l` from the Ubuntu Terminal. This was used to identify that the card had been mapped onto the `/dev/sdb1` device file within the (virtual) system.
- The command `hexdump -C -v /dev/sdb1 > <output_file_path>/<output_file_name>.txt` was used to obtain a memory dump of the SD card to a text file at a location specified by `output_file_path`. The option `-c` specified that the output should be in Canonical hex+ASCII display (for easier analysis), while `-v` indicated that all data from `/dev/sdb1` should be written out [77].
- At ~10GB, the resultant hexdump file was too large to be analyzed as a single unit. It was therefore split into several (125) files, using a PowerShell script from [78].
- These individual files were then opened in Notepad++ and analyzed.

B.2. Analysis of Memory Images - External Storage

To analyze data residues of an application, it is first necessary to identify the data that was written by the application. To do this, the SD card was low-level formatted and then imaged using the `hexdump` tool, to obtain an image of a clean card.

The Data Analyzer application was then installed onto the card and the storage was imaged again. The difference between the two images (which would correspond to application code, and any other ‘administrative’ data) was compared using the WinMerge tool.

When an application is first installed, some identification information is included on storage. This is of the form `com.<domain>.<package>`. Figure 15 shows the identifier corresponding to the Data Analyzer application (`com.proj.dataanalyzer`).
The application code can also be identified (see Figure 16), although it is not in recognizable patterns, due to the fact that it is held within an encrypted container [40].

The application was then executed and some data was written to file. In particular, some text files, a database, and a key-value pair (all with fixed data content) were created. The SD card was imaged again and compared with the previous images to identify the location to which the data had been written. It was observed that the file name (Figure 17) and the actual contents of the files (Figure 18) were in different areas of memory.
Note that the hex dumps allow us to identify the file name, which in this instance was extNormalFileDelete.txt. The file contained the string “EXT_STANDARD_ANDROID_FILE_DELETE”, which again is easily identifiable within the memory dump (Figure 18).

Effectiveness of API Delete Methods
Once data files had been written to memory, the Data Analyzer application was used to programmatically delete one of the files (it deleted the extNormalFileDelete.txt file) as follows:

```java
/* Delete file on external storage */
try{
    // Get the app's private directory.
    File privateDir = this.getExternalFilesDir(null);

    // Delete the text file using standard Android file deletion APIs
    File extNormDeleteFile = new File(privateDir, extNormalFILENAME);
    extNormDeleteFile.delete();
}
catch(Exception e){...}
```
The memory card was imaged and compared with a hex dump that had been obtained prior to the file deletion. It was observed that the file is marked for overwriting, but that the file contents still remains in memory.

An entry which is deleted in FAT file systems is marked with 0xE5. This can be observed in Figure 19, where the file that was 'deleted' has the bytes e5 at the beginning of its information.

![Figure 19: Identifying Files Marked for Deletion](image1)

Figure 20 depicts hex dumps which confirm that the file's contents are still present and recoverable from memory.

![Figure 20: Recovery of Deleted File Data from External Storage](image2)

**Application Uninstallation and Residual Data**

The next part of the experiment dealt with analyzing data residues after the application was uninstalled from the device (or, more precisely, from the SD card). The analysis considered both residual data as well as remaining code.

For this, the application was uninstalled and the memory card was imaged, as before. The resultant hex dumps showed that applications themselves are marked in a similar manner as data files when they are uninstalled, as shown in Figure 21. That is, the pointer to the application is modified.
However, all application data (Figure 22) and application code (Figure 23) still remains on the SD card, even after the application has been uninstalled.
Factory Reset and SD Card Format

When the device was reset to factory conditions, an option was presented to format the SD card. The options screens that were presented to the user as part of the reset procedure specified that erased data would not be recoverable (Figure 24).

![Android Factory Reset Procedure](image)

However, when the memory card was imaged subsequent to the reset and formatting process, all data and code still remained on external storage, albeit ‘invisible’ to the file system, as before.

Data Longevity

To identify the length of time for which deleted application data can remain on an SD card, a practical experiment was conducted, the details of which have been provided in Section 5.2.6 and briefly outlined below:

- Eight Android applications were downloaded and installed onto the SD card.
- For every application, the card was imaged once after the application was installed, and once after the application was executed. The application identifier, code, and data were identified.
- The applications were uninstalled from the SD card, and eight new applications were installed.
- After every new installation, the memory card was imaged to identify how much information pertaining to each old application remained.
- The values were tabulated and then used to produce three graphs, depicting the trend in data longevity on SD cards.

Section 5.2.6 within the main text has already specified the results obtained in terms of application data longevity. Therefore, only the data and graphs corresponding to application identification and application code have been provided here.

Application Identifier Residue

An application identifier is a value of the form `com.<domain>.<app_name>`. Being able to recover this value will provide an indication as to what applications have/had been installed on the SD card.

The value in row \textit{m}, column \textit{n} in Table 7 refers to the proportion of the application identifier for old app \textit{m} that is recoverable after new app \textit{n} has been installed.
Table 7: Percentage Residues of Application Identifiers

<table>
<thead>
<tr>
<th>Application Identifier</th>
<th>New App #1</th>
<th>New App #2</th>
<th>New App #3</th>
<th>New App #4</th>
<th>New App #5</th>
<th>New App #6</th>
<th>New App #7</th>
<th>New App #8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old App #1</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #2</td>
<td>75%</td>
<td>13%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #3</td>
<td>100%</td>
<td>100%</td>
<td>25%</td>
<td>25%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #4</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #5</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #6</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #7</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #8</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>66%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

The data from Table 7 has been graphed to produce the diagram in Figure 25.

![Application Identifier Longevity](image)

**Figure 25: Application Identifier Longevity on External Storage**

With application identifiers, the normal trend seems to have been that when new application $i$ is installed onto the SD card, the identifier for old application $i$ is overwritten. Although this pattern was not observed uniformly across all applications, it appeared to be the most common behaviour. Occasionally, partial overwrites were also observed.

**App Code Residue**

Application code refers to an encrypted block (or blocks) of code. It was assumed that only the blocks of code written immediately upon installation were application code.

The value in row $m$, column $n$ in Table 8 refers to the proportion of the application code for old app $m$ that is recoverable after new app $n$ has been installed.

Table 8: Percentage Residues of Application Code

<table>
<thead>
<tr>
<th>Application Identifier</th>
<th>New App #1</th>
<th>New App #2</th>
<th>New App #3</th>
<th>New App #4</th>
<th>New App #5</th>
<th>New App #6</th>
<th>New App #7</th>
<th>New App #8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old App #1</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #2</td>
<td>75%</td>
<td>2%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #3</td>
<td>98%</td>
<td>98%</td>
<td>33%</td>
<td>33%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>Old App #4</td>
<td>100%</td>
<td>96%</td>
<td>94%</td>
<td>94%</td>
<td>80%</td>
<td>8%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #5</td>
<td>98%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>45%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #6</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #7</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Old App #8</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>95%</td>
</tr>
</tbody>
</table>
The data in Table 8 was used to produce the graph in Figure 26.

![Application Code Longevity on External Storage](image)

**Figure 26: Application Code Longevity on External Storage**

It can be seen that application code showed a lot of variability between applications in terms of how much was overwritten by new applications.

**Notes:**
- Old App #1 did not get executed (the application crashed), and the “data” consisted of '?'s and '*'s. Therefore, the fact that all this “data” remained may be an exception.

**Application Information**
It should be noted that the amount of data that is recoverable will depend on the capacity of external storage, the sizes of the applications and the amount of data written by the applications. For this reason, details about the applications (or more specifically, the application sizes) that were used for this experiment have been provided in Table 9.

**Table 9: Application Information for Data Longevity Experiment**

<table>
<thead>
<tr>
<th>App ID</th>
<th>App Size on SD</th>
<th>App Data on SD</th>
<th>App ID</th>
<th>App Size on SD</th>
<th>App Data on SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old App #1</td>
<td>3.62MB</td>
<td>1.74MB</td>
<td>New App #1</td>
<td>4.09MB</td>
<td>11.73MB</td>
</tr>
<tr>
<td>Old App #2</td>
<td>1.88MB</td>
<td>1.35MB</td>
<td>New App #2</td>
<td>2.00MB</td>
<td>188KB</td>
</tr>
<tr>
<td>Old App #3</td>
<td>1.09MB</td>
<td>56.00KB</td>
<td>New App #3</td>
<td>4.25MB</td>
<td>140KB</td>
</tr>
<tr>
<td>Old App #4</td>
<td>4.18MB</td>
<td>76.00KB</td>
<td>New App #4</td>
<td>1.69MB</td>
<td>88.00KB</td>
</tr>
<tr>
<td>Old App #5</td>
<td>2.52MB</td>
<td>240KB</td>
<td>New App #5</td>
<td>3.60MB</td>
<td>92.00KB</td>
</tr>
<tr>
<td>Old App #6</td>
<td>3.31MB</td>
<td>9.24MB</td>
<td>New App #6</td>
<td>2.41MB</td>
<td>10.08KB</td>
</tr>
<tr>
<td>Old App #7</td>
<td>3.48MB</td>
<td>92.00KB</td>
<td>New App #7</td>
<td>2.48MB</td>
<td>84.00KB</td>
</tr>
<tr>
<td>Old App #8</td>
<td>4.14MB</td>
<td>724KB</td>
<td>New App #8</td>
<td>3.35MB</td>
<td>320KB</td>
</tr>
</tbody>
</table>
B.3. Analysis of Memory Images - Internal Storage

Although the steps for obtaining internal memory images vary from those used for external memory, the resulting hex dumps can be analyzed in the same manner. Note that all the results in this section were obtained from a rooted device.

API file deletion methods

To assess the effectiveness of API file deletion methods within internal storage, the *Data Analyzer* application was used to write files to internal memory and the /data directory was imaged. A file was then programmatically deleted using the standard Android APIs, and the directory was imaged again. The two hexdumps were compared using the WinMerge tool, to find that the file contents had been removed from memory (Figure 27).

![Figure 27: Data Erasure from Internal Memory](image)

However, similar to the case with SD cards, the file names were still present in memory, albeit “hidden” (see Figure 28).

![Figure 28: File Name Residues in Internal Storage](image)

Application uninstallation

The application was then uninstalled from the device, and /data was imaged and compared with previous hex dumps. It was observed that application code was only partially erased. That is, only certain blocks of code were removed (as shown in Figure 29).
However, there were some application code segments which remained in memory. It was possible to identify that these code segments did indeed belong to the test application by means of application-specific files, such as `activity_root_prevent.xml`, which was a layout files within the Data Analyzer application (see Figure 30).

It was also observed that application identifiers and folder structures remained, although marked as deleted (Figure 31).
Extraction of external storage encryption keys

An exploration of files on the rooted Android test device revealed a file named AppsOnSD.sks within /data/misc/systemkeys. Executing busybox od -t x1 AppsOnSD.sks returned the value of the key in hexadecimal [40], as shown in Figure 32.

![Command Prompt - adb shell]

Figure 32: Recovery of SD Card Encryption Key

According to [40], this corresponds to a 128-bit key used with the Twofish algorithm, and is the same key which is used to encrypt copy-protected applications on internal storage in newer versions of Android.