On Implementing Deniable Storage Encryption for Mobile Devices

Adam Skillen and Mohammad Mannan
Concordia Institute for Information Systems Engineering
Concordia University, Montreal, Canada
{*a_skil, mmannan*}@*ciise.concordia.ca*

Abstract

Data confidentiality can be effectively preserved through encryption. In certain situations, this is inadequate, as users may be coerced into disclosing their decryption keys. In this case, the data must be hidden so that its very existence can be denied. Steganographic techniques and deniable encryption algorithms have been devised to address this specific problem. Given the recent proliferation of smartphones and tablets, we examine the feasibility and efficacy of deniable storage encryption for mobile devices. We evaluate existing, and discover new, challenges that can compromise plausibly deniable encryption (PDE) in a mobile environment. To address these obstacles, we design a system called Mobiflage *that enables PDE on mobile devices by hiding encrypted volumes within random data on a device's external storage. We leverage lessons learned from known issues in deniable encryption in the desktop environment, and design new countermeasures for threats specific to mobile systems. Key features of Mobiflage include: deniable file systems with limited impact on throughput; efficient storage use with no data expansion; and restriction/prevention of known sources of leakage and disclosure. We provide a proof-of-concept implementation for the Android OS to assess the feasibility and performance of Mobiflage. We also compile a list of best practices users should follow to restrict other known forms of leakage and collusion that may compromise deniability.*

1 Introduction and Motivation

Smartphones and other mobile computing devices are being widely adopted globally. For instance, according to a comScore report [7], there are more than 119 million smartphone users in the USA alone, as of Nov. 2012. With this increased use, the amount of personal/corporate data stored in mobile devices has also increased. Due to the sensitive nature of (some of) this data, all major mobile OS manufacturers now include some level of storage encryption. Some vendors use file based encryption, such as Apple's iOS, while others implement "full disk encryption" (FDE). Google introduced FDE in Android 3.0 (for tablets only); FDE is now available for all Android 4.x devices, including tablets and smartphones.

While Android FDE is a step forward, it lacks deniable encryption—a critical feature in some situations, e.g., when users want to provide a decoy key in a plausible manner, if they are coerced to give up decryption keys. Plausibly deniable encryption (PDE) was first explored by Canetti et al. [6] for parties communicating over a network. As it applies to storage encryption, PDE can be simplified as follows: different reasonable and innocuous plaintexts may be output from a given ciphertext, when decrypted under different decoy keys. The original plaintext can be recovered by decrypting with the *true* key. In the event that a ciphertext is intercepted, and the user is coerced into revealing the key, she may instead provide a decoy key to reveal a plausible and benign decoy message. The Rubberhose filesystem for Linux (developed by Assange et al. [3]) is the first known instance of a PDE-enabled storage system.

Some real-world scenarios may mandate the use of PDE-enabled storage—e.g., a professional/citizen journalist, or human rights worker operating in a region of conflict or oppression. In a recent incident [45], an individual risked his life to smuggle his phone's micro SD card, containing evidence of atrocities, across international borders by stitching the card beneath his skin. Mobile phones have been extensively used to capture and publish many images and videos of recent popular revolutions and civil disobedience. When a repressive regime disables network connectivity in its jurisdiction, PDE-enabled storage on mobile devices can provide a viable alternative for data exfiltration. With the ubiquity of smartphones, we postulate that PDE would be an attractive or even a necessary feature for mobile devices. Note, however, that PDE is only a technical measure to prevent a user from being punished if caught with contentious material; an adversary can always wipe/confiscate the device itself if such material is suspected to exist.

Several existing solutions support full disk encryption with plausible deniability in regular desktop operating systems. Possibly the most widely used such tool is TrueCrypt [46]. To our knowledge, no such solutions exist for any mainstream mobile OSes, although PDE support is apparently more important for these systems, as mobile devices are more widely used and portable than laptops or desktops. Also, porting desktop PDE solutions to mobile devices is not straightforward due to the tight coupling between hardware and software components, and intricacies of the system boot procedure. For example, in Android, the framework must be partially loaded to use the soft keyboard for collecting decoy/true passwords; and the TrueCrypt bootloader is only designed to chainload Windows.

We introduce *Mobiflage*, a PDE-enabled storage encryption system for the Android OS. It includes countermeasures for known attacks against desktop PDE implementations (e.g., [10]). We also explore challenges more specific to using PDE systems in a mobile environment, including: collusion of cellphone carriers with an adversary; the use of flash-based storage as opposed to traditional magnetic disks; and file systems such as Ext4 (as used in Android) that are not so favorable to PDE. Mobiflage addresses several of these challenges. However, to effectively offer deniability, Mobiflage must be widely deployed, e.g., adopted in the mainstream Android OS. As such, we implement our Mobiflage prototype to be compatible with Android 4.x.

Our contributions include:

1. We explore sources of leakage inherent to mobile devices that may compromise deniable storage encryption. Several of these leakage vectors have not been analyzed for existing desktop PDE solutions.
2. We present the Mobiflage PDE scheme based on hidden encrypted volumes—the first such scheme for mobile systems to the best of our knowledge.
3. We provide a proof-of-concept implementation of Mobiflage for Android 4.x (Ice Cream Sandwich and Jelly Bean). We incorporated our changes into 4.x and maintained the default full disk encryption system. During the normal operation of Mobiflage (i.e., when the user is not using hidden volumes), there are no noticeable differences to compromise the existence of hidden volumes.
4. We address several challenges specific to Android. For example, to avoid PDE-unfriendly features of the Ext4 file system (as used for the Android userdata partition), we implement our hidden volumes (userdata and external) within the FAT32-based external partition.
5. We analyze the performance impact of our implementation during initialization and for data-intensive applications. In a Nexus S device, our implementation appears to perform almost as efficiently as the default Android 4.x encryption for the applications we tested. However, the Mobiflage setup phase takes more time than Android FDE, due to a two-pass wipe of the external storage (our Nexus S required almost twice as long; exact timing will depend on the size and type of external storage).

The remainder of this paper is organized as follows. Section 2 presents our threat model and assumptions. In Section 3, we describe design choices of Mobiflage deniable disk encryption system for mobile devices. In Section 4, we discuss the implementation of Mobiflage for Android. In Section 5, we list several measures that must be observed to maintain deniability in a mobile environment. Section 6 describes sources of leakage and attacks that may compromise deniability on a mobile device. We analyze security implications and performance of our implementation in Sections 7 and 8 respectively. Section 9 discusses related work and Section 10 concludes.

2 Threat Model and Assumptions

In this section, we discuss Mobiflage's threat model and operational assumptions, and few legal aspects of using PDE in general. The major concern with maintaining plausible deniability is whether the system will provide some indication of the existence of any hidden data. Mobiflage's threat model and assumptions are mostly based on past work on desktop PDE solutions (cf. TrueCrypt [46]); we also include threats more specific to mobile devices.

Threat model and operational assumptions.

1. Mobiflage must be merged with the default Android code stream, or a widely used custom firmware based on Android (e.g., CyanogenMod[1]) to ensure that many devices are capable of using PDE. Then an adversary will be unable to make assumptions about the presence of hidden volumes based on the availability of software support. We do not require a large user base to employ PDE; it is sufficient that the capability is widespread, so the availability of PDE will not be a red flag. Similar to TrueCrypt [46], all installations of Mobiflage include PDE capabilities. There are no identifying technical differences between the default and PDE encryption modes. However, when more users enable default encryption, they help to obscure those that use PDE.
2. Mobiflage currently requires a physical or emulated FAT32 SD card. Devices, such as the Nexus S, which use an internal eMMC partition as opposed to a removable SD card are supported. Some devices, such

[1] http://www.cyanogenmod.org/

as the Galaxy Nexus, have neither physical nor emulated external storage. Instead, they use the media transfer protocol (MTP) and share a single Ext4-formatted partition for the (internal) app storage and (external) user accessible storage. These devices are not currently supported; possible solutions are outlined in Section 6.3.

3. The adversary has the encrypted device and full knowledge of Mobiflage's design, but lacks the PDE key (and the corresponding password). The offset of Mobiflage's hidden volume is dependent on the PDE password, and is therefore also unknown to the adversary.
4. The adversary has some means of coercing the user to reveal their encryption keys and passwords (e.g., unlock-screen secret), but will not continue to punish the user once it becomes futile (e.g., the adversary is convinced that he has obtained the true key, or the assurance that no such key actually exists). To successfully provide deniability in Mobiflage, the user is expected to refrain from disclosing the true key.
5. The adversary can access the user device's internal and external storage, and can have root-level access to the device after capturing it. The adversary can then manipulate disk sectors, including encryption/decryption under any decoy keys learned from the user; this can compromise deniability (e.g., the "copy-and-paste" attack [18]). Mobiflage addresses these issues.
6. The adversary model of desktop FDE usually includes the ability to periodically access or snapshot the encrypted physical storage (cf. [10, 1]). However, this assumption is unlikely for mobile devices and has therefore been relaxed (as the adversary will have access to the storage media only after apprehending the user).
7. In addition to the Dolev-Yao network attacker model [33, 13], we also assume that the adversary has some way of colluding with the wireless carrier or ISP (e.g., a state-run carrier, or subpoena power over the provider). Adversaries can collect network/service activity logs from these carriers to reveal the use of a PDE mode on suspected devices. This assumption significantly strengthens the attacker model, nonetheless, is quite realistic (see e.g., [11]).
8. We assume the mobile OS, kernel, and bootloader are malware-free, and while in the PDE mode, the user does not use any adversary controlled apps to avoid leaking information via those apps; i.e., in the PDE mode, the user is expected to use only *trusted* apps. The device firmware and baseband OS are also trusted. Control over the baseband OS may allow an adversary to monitor calls and intercept network traffic [50], which may be used to reveal the PDE mode. Mobile malware, and defining/verifying trusted code are independent problems, and are out of scope here.
9. We assume the adversary cannot capture the user device while in the PDE mode; otherwise, user data can be trivially retrieved if the device is unlocked. We require the user to follow certain guidelines, e.g., not using Mobiflage's PDE-mode for regular use; other precautions are discussed in Section 5. Following these guidelines may require non-trivial effort, but is required for maintaining deniability in our threat model.

Legal aspects. Some countries require mandatory disclosure of encryption keys in certain cases. Failure to do so may lead to imprisonment and/or other legal actions; several such incidents occurred in the recent past (e.g., [43, 44]). Cryptography can be used for both legal and illegal purposes and governments around the globe are trying to figure out how to balance laws against criminal use and user privacy. As such, laws related to key disclosure are still in flux, and vary widely among countries/jurisdictions; see e.g., Koops [30].

Some of our recommendations, such as spoofing the IMEI or using an anonymous/"burner" SIM card, may be illegal in certain regions. Local laws should be consulted before following such steps. Mobiflage is proposed here not to encourage breaking laws; we want to technically enable users to benefit from PDE, but leave it to the user's discretion how they will react to certain laws. Our hope is that Mobiflage will be predominantly used for good purposes; e.g., human rights activists in repressive regimes.

3 Mobiflage Design

In this section, we detail our design and explain certain choices we made. User steps for Mobiflage are also provided. Parts of the design are Android specific, as we use Android for our prototype implementation; however, we believe certain aspects can be abstracted to other systems. Challenges to port the current design into other OSes need further investigation (e.g., Apple iOS does not use FDE, and the file system and storage layout are also different).

3.1 Overview and Modes of Operation

We implement Mobiflage by hiding volumes in empty space on a mobile device's external (SD or eMMC) storage partition. We first fill the storage with random noise, to conceal the existence of additional encrypted volumes. We create two adjacent volumes: a userdata volume for applications and settings, and a larger auxiliary volume for accumulating documents, photos, etc. The exact location of the hidden volumes on the external storage is derived from the user's deniable password. We store all hidden volumes in the external storage, due to certain file system limitations discussed in Section 6.3.

We define the following modes of operation for Mobiflage. **(a)** *Standard mode* is used for day-to-day operation of the device. It provides storage encryption without deniability. The user will supply their decoy password at boot time to enter the standard mode. In this mode, the storage media is mounted in the default way (i.e., the same configuration as a device without Mobiflage). We use the terms “decoy” and “outer” interchangeably when referring to passwords, keys, and volumes in the standard mode. **(b)** *PDE mode* is used only when the user needs to gather/store data, the existence of which may need to be denied when coerced. The user will supply their true password during system boot to activate the PDE mode; we mount the hidden volumes onto the file-system mount-points where the physical storage would normally be mounted (e.g., `/data`, `/mnt/sdcard`). We use the terms “true”, “hidden” and “deniable” interchangeably when referring to passwords, keys, and volumes in the PDE mode.

3.2 Steganography vs. Hidden Volumes

There are currently two main types of PDE systems for use with FDE: steganographic file systems (e.g., StegFS [1, 32]) and hidden volumes (e.g., TrueCrypt [46] and FreeOTFE [17]). Steganographic file systems’ known drawbacks include: inefficient use of disk space, possible data loss, and increased IO operations. These limitations are unacceptable in a mobile environment, for reasons such as performance sensibility, and relatively limited storage space. (For more background on these systems, see Appendix A.) Consequently, we choose to use hidden volumes for Mobiflage. This implies: no altered file system drivers are required; IO is as efficient as a standard encrypted volume; and the chance of data loss is mitigated, although not completely eliminated. Most deniable file systems are lossy by nature. Hidden volumes mitigate this risk by placing all deniable files toward the end of the storage device. Assuming the user knows how much space is available for the deniable volume, they can refrain from filling the outer volume past the point at which the hidden volumes begin.

3.3 Storage Layout

The entire disk is encrypted with a decoy key and formatted for regular use; we call this the outer volume. Then additional file systems are created at different offsets within the disk and encrypted with different keys; these are referred to as hidden volumes. To prevent leakage, Mobiflage must never mount hidden volumes alongside outer volumes. Thus, we create corresponding hidden volumes, or RAM disks, for each mutable system mount point (e.g., `/userdata`, `/cache`, `/mnt/sdcard`).

Some hidden volumes may be decoys, but at least one hidden volume will contain the actual sensitive data and be encrypted with the true key. Since the outer volume is filled with random noise before formatting, there are no distinguishing characteristics between empty outer-volume blocks and hidden volume blocks. When the outer volume (or a hidden decoy volume) is mounted, it does not reveal the presence or location of any other hidden volumes. All hidden volumes are camouflaged amongst the random noise. The disk can be thought of as the concatenation of encrypted volumes, each with a different key:

$$E_{K1}(Vol_1)||E_{K2}(Vol_2)||...||E_{Kn}(Vol_n)$$

Here, $E_K(\cdot)$ represents a symmetric encryption function with key K and $||$ represents concatenation.

When the disk is decrypted with a given key, the other volumes will appear to be uniformly random data. When the user is coerced, she can provide the outer volume key and claim that no other volumes exist:

$$D_{K1}(Vol_1||Vol_2||...||Vol_n) = Vol_1||Rand$$

Here, $D_K(\cdot)$ represents the symmetric decryption function with key K (corresponding to $E_K(\cdot)$) and $Rand$ represents data that cannot be distinguished from random bits. Therefore, a forensic analysis of the decrypted outer volume will not indicate the existence of hidden volumes. However, some statistical deviations may be used to distinguish the random data from the cipher output; see Section 6.1. Also, the adversary may not trust the user to have disclosed all volume keys and continue to coerce her for additional keys. At this time, the user can provide decoy keys for other hidden volumes and insist that all the volumes have been exposed. Revealing the existence of any hidden volume may either help or hinder the user, depending on the situation; see Section 7, item (e).

Each decrypted volume will appear to consume all remaining disk space on the device. For this reason it is possible to destroy the data in the hidden volumes by writing to the currently mounted volume past the volume boundary. This is unavoidable since a visible limit on the mounted volume would indicate the presence of hidden volumes.

3.4 Offset Calculation

The offset to a hidden volume is generated as follows:

$$offset = 0.75 \times vlen - (\mathrm{H}(pwd||salt) \bmod (0.25 \times vlen))$$

Here, H is a PBKDF2 iterated hash function [26], $vlen$ is the number of 512-byte sectors on the logical block storage device, pwd is the true password, and $salt$ is a random salt value for PBKDF2. The salt value used here is the same as for the outer volume key derivation (i.e., stored in the encryption footer). Thus, we avoid introducing an additional field in the default encryption footer that may indicate the presence of hidden volumes. The generated offset is greater than one half and less than three quarters of the disk; i.e., the hidden volume's size is between 25-50% of the total disk size (assuming only one hidden volume is used). We choose this offset as a balance between the hidden and outer sizes: the outer volume will be used more often, the hidden volume is used only when necessary. To avoid overwriting hidden files while the outer volume is mounted, we recommend the user never fills their outer volume beyond 50%.

Deriving the offset in the above manner allows us to avoid storing it anywhere on the disk, which is important for deniability. For comparison, TrueCrypt uses a secondary volume header to store the hidden offset, encryption key and other parameters; all the header fields are either random or encrypted, i.e., indistinguishable from the encrypted volume data. In contrast, Android uses volume footers containing plaintext fields, similar to the Linux unified key setup (LUKS [18, 19]) header. Introducing a new field to store the offset would reveal the use of Mobiflage PDE, so we choose to derive the offset from the password instead. Other systems, e.g., FreeOTFE, mandate users to remember the offset; prompting the user for the offset at boot time may also be a red flag for the adversary. The obvious downside of a password-derived offset is that the user has no input on the size of the hidden volumes. One possible method to accommodate user choice is discussed in Section 4.4, item (2).

3.5 User Steps

Here, we describe how users may interact with Mobiflage, including initialization and use.

Users must first enable device encryption with PDE (e.g., through settings GUI). Mobiflage's initialization phase erases existing data on the external storage (SD card); this data should be backed-up before initiating Mobiflage. However, users can choose to preserve the outer volume's user-data partition within the internal storage; this partition may be encrypted in-place or initialized with random data (depending on user choice). Assuming a single hidden volume is used, the user then enters the decoy and true passwords, for the outer and hidden volumes respectively. Mobiflage then creates the hidden volumes, performs in-place encryption of the internal storage (if chosen) and reboots when complete. Unlike Android FDE, Mobiflage must initialize the external storage with random data for deniability. This makes Mobiflage slower than the default Android FDE initialization (see Section 8). However, the initialization step will likely be performed only occasionally.

For normal day-to-day use (e.g., phone calls, web browsing), the user enters the decoy password during pre-boot authentication to activate the standard mode. All data saved to the device in this mode will be encrypted but not hidden. It is important for the user to regularly use the device in this mode, to create a digital paper trail and usage timeline which may come under scrutiny during an investigation. The user gains plausibility by showing that the device is frequently used in this mode; i.e., she can demonstrate apparent compliance with the adversary's orders.

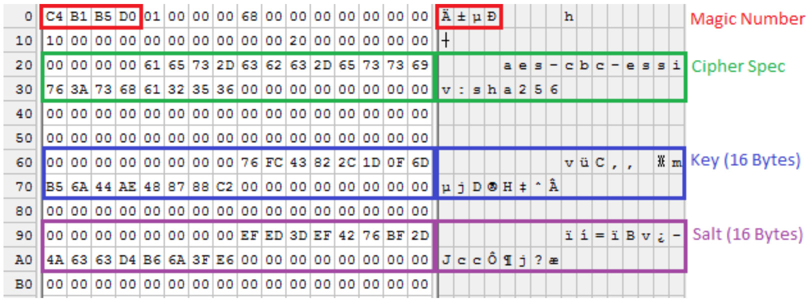
When the user requires the added protection of deniable storage, they will reboot their device and provide their deniable password when prompted. In the PDE mode, they can transfer documents from another device, or take photos and videos. Note that app/system logs in this mode are hidden or discarded; however, there is still a possibility of leakage through network interfaces. Section 5 provides a list of precautions the user should take to mitigate such risks.

After storing or transferring files to the deniable storage, the user should immediately reboot into the standard mode. The files are hidden as long as the device is either off, or booted in the standard mode. If the user is apprehended with the device in the PDE mode, deniability is lost. Even if the user shuts the device off shortly before being apprehended, there is a possibility that the adversary can obtain the key from data remanence in the RAM (e.g., the cold-boot attack [20]).

If the user is apprehended with her device, she can supply a decoy password, and claim that no hidden volumes exist. The adversary can examine the storage but will not find any record of the hidden files, apps, or activities. Depending on the situation, the user can provide additional decoy passwords, when faced with continued coercion. A rational adversary may not punish the user if they have no reason to believe that (further) hidden data exists on the device. Assuming the user can overcome any coercion the adversary attempts, and does not reveal the true key, the adversary will have no evidence of the hidden data.

4 Mobiflage Implementation

We developed and tested Mobiflage on a Google/Samsung Nexus S phone using the 4.0 (ICS) and 4.1 (JB) Android source code. The addition of PDE functionality to the Android volume mounting daemon (`vold`) required less than one thousand additional lines of code, and subtle changes to the default kernel configuration. We also discuss current limitations of Mobiflage. In addition to the Nexus S, we also tested the portability of our prototype to a Motorola Xoom.



```
 0  C4 B1 B5 D0 01 00 00 00 68 00 00 00 00 00 00 00  Ä±µÐ    h
10  10 00 00 00 00 00 00 00 00 00 20 00 00 00 00 00  ┼
20  00 00 00 00 61 65 73 2D 63 62 63 2D 65 73 73 69      aes-cbc-essi
30  76 3A 73 68 61 32 35 36 00 00 00 00 00 00 00 00  v:sha256
40  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60  00 00 00 00 00 00 00 00 76 FC 43 82 2C 1D 0F 6D          vüC‚, ☼m
70  B5 6A 44 AE 48 87 88 C2 00 00 00 00 00 00 00 00  µjD®H‡ˆÂ
80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90  00 00 00 00 00 00 00 00 EF ED 3D EF 42 76 BF 2D          ïí=ïBv¿-
A0  4A 63 63 D4 B6 6A 3F E6 00 00 00 00 00 00 00 00  JccÔ¶j?æ
B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 1: Android FDE footer (note that the cipher specification field is stored in clear text)

4.1 Changes to Android FDE

We first provide a brief introduction to Android FDE, as Mobiflage has been implemented by enhancing this scheme. We then discuss the changes we introduced.

The Android encryption layer is implemented in the logical volume manager (LVM) device-mapper crypto target: dm-crypt [12]. Encryption takes place below the file system and is hence transparent to the OS and applications. The AES cipher is used in the CBC mode with a 128-bit key. ESSIV is used to generate *unpredictable* IVs to prevent watermarking attacks (Fruhwirth [18]; see also Section 6.1). A randomly chosen master volume key is encrypted with the same cipher by a key derived from 2000 iterations of the PBKDF2 [26] digest of the user's screen-unlock password and a salt value. To enable encryption, the user must choose either an unlock password or PIN (i.e., pattern and "Face Unlock" secrets may not be used). The cipher specification, encrypted master key and salt are stored in a footer located in the last 16KB of the userdata partition; see Figure 1 for an example Android encryption footer.

When the device is booted and fails to find a valid Ext4 file system on the userdata partition, the user is prompted for their password. The master key is decrypted from their password-derived key. Storage read/write operations are passed through the device mapper crypto target, so encryption/decryption is performed on-the-fly for any IO access. If a valid Ext4 file system is then found in the dm-crypt target, it is mounted and the system continues to boot as usual. Otherwise, the user is asked to re-enter their password. By default, removable SD cards are not encrypted; however, emulated *external* storage (i.e., a physical eMMC partition, mounted at /mnt/sdcard) is encrypted.

We made three important changes to the default Android encryption scheme that are necessary to defend deniability: (a) we use the XTS-AES cipher instead of CBC-AES; (b) we enable encryption of removable storage; and (c) we wipe the SD card with random data. XTS-AES is chosen as a precaution against copy-and-paste and malleability attacks (see Section 6.1 for details). We use a 512-bit key (256-bit for AES and 256-bit for XEX tweak). This gives the cipher additional strength over the 128-bit Android key-length, but more importantly makes the key exactly one disk sector in size for easy alignment of hidden volumes. Note that, although the 256-bit random key strengthens AES, the overall security of the system defaults to the strength of the password used to protect the volume key. The xts and gf128mul kernel crypto modules were compiled for our development devices, to enable the XTS mode. These modules are available in the Linux kernel since version 2.6.24.

Android encryption can be performed in-place (i.e., reading each sector, encrypting it, and writing it back to the disk), or by first formatting the storage media. We perform the wipe operation on the SD card even when the user enables in-place encryption. We enhance the wipe operation to fill the flash media with random data to address data remanence issues and to hide the PDE volumes (see Section 6.2 for details). These changes are necessary even when encrypting without PDE, to make the default encryption indiscernible from PDE. Our changes should not negatively affect the security of Android FDE.

4.2 Partitions and File-system Support

Here we describe the Android storage layout and file systems, as well as the Mobiflage storage structure used to implement PDE.

Device storage partitions. The exact storage layout of a mobile device is manufacturer/device specific; Table 1 shows volumes typically found. Android 4.x has two partitions that store user data: the internal Ext4 userdata partition and the (emulated or physical) external FAT32 partition. The userdata partition stores apps and settings, while the external partition stores documents, downloads, photos, etc. We create both a hidden userdata partition and a hidden external partition for use in the PDE mode. This allows the user to store hidden files as well as install hidden apps. The OS and kernel are stored on read-only volumes, and can be safely shared between the two modes. This also simplifies system updates, since updating the kernel/OS in the standard mode will be reflected in the PDE mode. Over-the-air system updates make use of the /cache partition, which is not persistent in the PDE mode, so updates must always take place from the standard mode. The default file system for

Volume name	Mount point	Mode	Description
Boot	N/A	N/A	Bootloader and kernel image
Recovery	N/A	N/A	Recovery tools and backup kernel
System	/system	RO	OS binaries, Dalvik VM, etc.
Cache	/cache	RW	Temporary space for OS and apps (e.g., OTA updates and downloaded .apk packages)
Device log	/devlog	RW	Persistent system logs
Userdata	/data	RW	Apps and settings
External	/mnt/sdcard or /storage/sdcard0	RW	App and user data (e.g., photos, maps, music)

Table 1: Typical volumes on common Android devices (RO: read-only; RW: read-write; N/A: not applicable)

the internal userdata partition is Ext4. For reasons outlined in Section 6.3, we cannot reliably hide a volume within an Ext4 file system. Instead, we store the hidden volumes in the FAT32 formatted external partition.

The FAT32 file system is much less complex than Ext4. FAT32 stores the allocation tables and all meta-data at the beginning of a disk. The remaining space is uninterrupted data blocks—i.e., no FAT backups or meta-data exist in further areas of the disk. Writing a hidden partition to an unused area of a FAT32 file system will not create any noticeable discrepancies, as would be visible in Ext4. We create a hidden Ext4 partition to store apps and settings, and a hidden FAT32 partition to store files such as photos and videos; see Figure 2. To prevent leakage into the outer volumes, when the hidden volumes are mounted, we use tmpfs[2] RAM disks for /cache and /devlog. We also discuss persistent cache and device log partitions in Section 7, item (c).

Mobiflage on-disk structure. Our prototype currently supports the creation of only one hidden volume offset (i.e., no additional decoy hidden volumes). The outer userdata and external volumes are first encrypted through the dm-crypt target. The footer, containing only the outer volume key (encrypted with the decoy-password derived key) and other default fields, is written to the disk in the usual manner. Before encrypting the outer external volume, it is first filled with random data produced with the XTS-AES cipher under two random, discardable keys (see Section 6.1). This is not performed by Android FDE, which may lead to data remanence attacks via the flash wear-leveling mechanism as discussed in Section 6.2.

We then generate an offset from the true password as described in Section 3.4. A randomly generated hidden volume key is then encrypted with a key derived from the true password. The encrypted hidden volume key is stored in the external partition at the derived offset. The hidden volumes immediately follow the key on the disk, and the volumes are encrypted by creating new dm-crypt mappings with the hidden key. The hidden userdata volume is 256MB and the hidden external volume consumes the remaining space. We choose 256MB for the userdata partition assuming this will be sufficient for the installation of several hidden apps (e.g., custom browser, secure VoIP and texting apps). This size may be user configurable (e.g., up to a maximum of 25% of the hidden space). However, the bulk of the user data is stored in the external storage (e.g., photos, downloads, maps), warranting a larger size.

To assess the portability of Mobiflage on other hardware profiles, we tested our prototype on a Motorola Xoom. The Xoom uses the shared internal/external MTP paradigm, but also contains an SD card slot. The shared MTP storage is treated as the primary external storage, and all external app data is stored at this location (essentially ignoring the SD card). We altered Mobiflage by embedding the location of the SD card block special file, to create and mount the hidden partitions. In this particular configuration, it is perhaps a better idea to create a single hidden Ext4 partition, since it will house all internal and external data, and the SD card is inaccessible to installed apps (i.e., 256MB will be insufficient for MTP devices). Other subtle tweaks may be necessary to support Mobiflage on different hardware profiles.

4.3 User Interface and Pre-boot Authentication

The default Android encryption mechanism can be enabled through the settings GUI. This prompts the user for their screen-unlock password, which is used to derive the volume encryption key. The system then shuts down non-essential services and starts encrypting the internal storage in-place. A user with root privileges can also use the vdc command-line tool (e.g., from a PC to which the Android device is connected) to enable encryption either in-place or with data wipe, as follows: "vdc cryptfs enablecrypto <inplace|wipe> <pwd>." In this case, pwd can be any password (i.e., independent of the screen-unlock password).

Currently, the user can activate Mobiflage PDE using vdc as follows: "vdc cryptfs pde <inplace|wipe> <outer_pwd> <hidden_pwd>." Note that, the default Android shell, sh, does not maintain history between sessions (i.e., command history cannot be retrieved from a captured Android device). In-place encryption is used only for the internal storage. We wipe the SD card to reliably fill the physical media with random noise. On flash media, it is

[2] http://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt

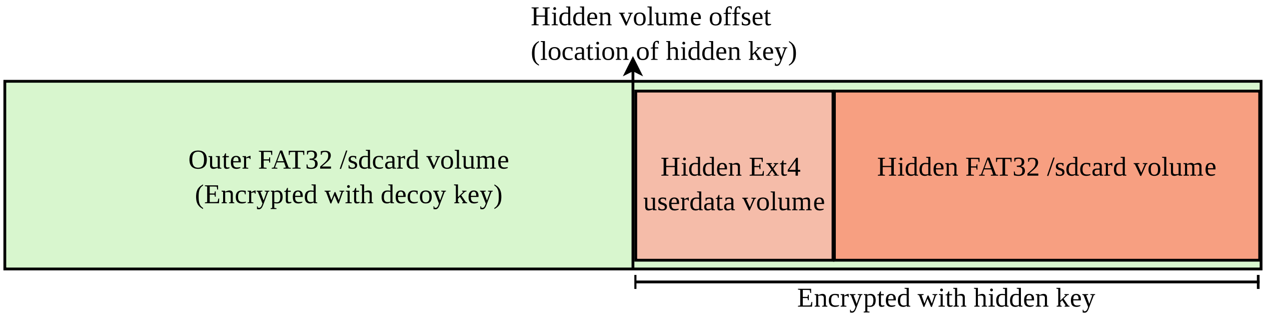

Hidden volume offset
(location of hidden key)
Outer FAT32 /sdcard volume
(Encrypted with decoy key)
Hidden Ext4
userdata volume
Hidden FAT32 /sdcard volume
Encrypted with hidden key

Figure 2: Mobiflage SD card PDE layout

apparently sufficient to completely fill the logical address space twice, as noted by Wei et al. [49]. Since internal storage does not house any hidden volumes, we forgo the random wipe and encrypt in-place. This allows the user to preserve their apps and settings in the standard mode and creates a fresh install (i.e., factory reset) in the PDE mode. We plan to incorporate the PDE options into the settings GUI in a future version of Mobiflage.

When the device is booted up, the system will attempt to mount the userdata volume. If a valid Ext4 file system is not found, the user is prompted for a password, assuming storage encryption is in use. The system then attempts to mount the volume with the stored key (decrypted with the password-derived key). If it fails, instead of asking the user to try again, it will calculate the volume offset from the supplied password. The external storage sector found at this offset is decrypted with the PBKDF2 derived key. Using the result as a volume key, the system will attempt to mount a volume beginning at the next external storage sector after the offset. If a valid Ext4 file system is found at this location, it is mounted. After mounting the hidden userdata and external volumes, the boot procedure continues as usual. If a hidden file system cannot be found at the derived offset, the system will prompt the user to try again, just as it would if PDE were not enabled.

4.4 Limitations

Limitations of our current Mobiflage design and prototype include the following:

1. Mobiflage currently requires a separate physical FAT32 storage partition (SD or eMMC). Devices that use MTP and share a single partition for internal and external storage are not currently supported. We discuss the problems inherent to Ext4, and provide suggestions for other file systems (e.g., HFS+, Ext2/3) in Section 6.3.
2. Users currently cannot set the desired size of a hidden volume; the size is derived from a user's password to avoid the need to store the offset on the device. An expected size may be satisfied as follows (not currently implemented). We can ask users for the desired size and iterate the hash function until an offset close to the requested size is found. For example, we can perform 20 additional hash iterations and report the closest size available with the supplied password. The user could then choose to either accept the approximate size or enter a new password and try again. Storing the iteration count is not needed. At boot time, the system will perform consecutive iterations until a valid file system is found, or a maximum count is reached (cf. [4]). This would slow down the boot process somewhat while searching for the correct offset.
3. Currently, we support only one hidden volume offset. Creating additional (decoy) hidden volumes will require a collision prevention mechanism to derive offsets. A method, such as the iteration count mentioned above, can be used to ensure enough space is left between hidden offsets (e.g., 1.5GB). This increases the chance of corrupting hidden data. Each hidden volume would appear to consume the remaining SD card storage, but the address space would overlap with other hidden volumes. We discuss the implications of multiple hidden volumes in Section 7, item (e).
4. Transferring data between outer and hidden volumes may be necessary on occasion; e.g., if time does not permit switching between modes before taking an opportunistic photo. We do not offer any safe mechanism for such transfers at present. Mounting both volumes simultaneously is a straightforward solution, but may compromise deniability (e.g., usage log data of a hidden file may be visible on the decoy volume). The user can transfer sensitive files to a PC as an intermediary, then transfer the files to the PDE storage. In this case, data remanence in the outer volume is an issue. Another possibility is to keep a RAM disk mounted in the standard mode for storing such opportunistic files (and then copy to the PDE storage via a PC). However, some apps, such as the camera app, do not offer an option to choose where files are saved.

5 Precautions against Colluding Carriers

In this section, we discuss threats from a colluding wireless carrier, and list a number of precautions that may help maintain deniability in the presence of such a carrier.

Mobile devices are often connected to a cellphone network. It is likely that the wireless carrier maintains activity records, with identifying information and timestamps, of devices interfacing with the network. These records can demonstrate that the device is online and communicating at a given time. The use of the PDE mode is likely to cause discrepancies between the carrier's logs and the device's standard mode (outer volume) logs. For example, if the carrier has records of a phone call at a given time, that occurred when the device was booted in the PDE mode, the device will not have a record of the call in the standard mode. In certain situations, an adversary may be able to collude with the carrier (e.g., a state-based carrier), or compel the carrier to disclose user records (e.g., by court orders). If the user has provided the adversary with the decoy password, the adversary may find discrepancies between the device logs and the carrier's logs. This would give the adversary reason to believe that the user has not been completely forthcoming. They may then continue to coerce the user for any additional passwords or keys.

To restrict the above threats, we provide a list of user practices that must be adhered to when booted in the PDE mode; some of these practices may be onerous to the user. This list shows pitfalls of using PDE systems in practice; however, it is not meant to be comprehensive.

1. When using the device in the PDE mode, it should be left in "Airplane mode" (i.e., antennae off), and the SIM card should be removed. This may prevent the wireless carrier from identifying the device/user in their activity logs.
2. A secondary anonymous SIM card should be used, and the phone's identifying information spoofed, if connecting to a mobile network while in the PDE mode (e.g., IMEI spoofing and software MAC address spoofing). This will restrict any carrier or ISP from directly identifying a suspected device.
3. We strongly discourage the use of mobile data networks in favor of public WiFi hot-spots or Internet pass-through/tethering from a PC. Identifying network traffic information (e.g., destination IP address) should be spoofed or obscured with a tool such as Tor[3] or a trusted (e.g., employer controlled) VPN when using any type of network connection. This may also restrict an ISP or carrier from correlating the user's behavior (e.g., if the user is known to frequent a certain file hosting service, or news agency).
4. When using the PDE mode, any web services (e.g., email, social networking) should not be used unless a secondary account is created under a pseudonym and is *only* used in the PDE mode. This will prevent any collusion between the adversary and web service providers with which the user is known to have an account. This includes the device registration account (e.g., Google or iTunes). It is also recommended that auto-backup features (e.g., iCloud or Google Drive) are disabled in the PDE mode.

6 Sources of Compromise

We examine three leakage vectors that may compromise deniability of a PDE scheme on mobile devices: known issues in crypto-systems and software implementations of desktop PDE schemes, as well as issues specific to current mobile storage systems. Below we discuss these challenges and how they are addressed in Mobiflage.

6.1 Leakage from Crypto Primitives

Crypto primitives used in a PDE implementation must be chosen carefully. Below we discuss issues related to random data generation and encryption modes.

PRNG. A fundamental requirement for PDE schemes implemented with hidden volumes is that the whole disk must appear to contain cryptographically secure random data. For this requirement, the cipher output must be indistinguishable from random bits (cf. IND$-CPA [40]). However, certain statistical deviations between cipher and PRNG output may exist (see e.g., [16, pp. 137–161]). To sidestep any potential statistical inconsistencies, we draw randomness from the same distribution as the ciphertext space by using the encryption function itself as the PRNG (in a two pass random-wipe, each pass with a new random key). Under statistical analysis, empty sectors in an outer volume will appear the same as the sectors in a hidden volume, when either encrypted or decrypted with a decoy key. For comparison, TrueCrypt uses a built-in PRNG to fill empty volume space, with the assumption that the cipher output will be indiscernible from their PRNG output.

Encryption modes. Encryption of data at rest has different considerations than the traditional communication encryption model. For example, to enable random-access, FDE implementations treat each disk sector as an autonomous unit and assign sector-specific IVs for chaining modes such as CBC. These IVs are long-term and must be easily derived from or stored in the local system. When FDE is implemented with a CBC-mode cipher, information leakage about the plaintext disk content may occur without knowledge of the encryption key or cipher used (see e.g., [18]).

[3] Tor on Android: https://www.torproject.org/docs/android.html.en

Tweakable block cipher modes (e.g., LRW and XTS) have been designed specifically for disk encryption to prevent attacks such as watermarking, malleability, and copy-and-paste. These attacks are particularly important for PDE, as they may be used to identify hidden volumes without recovering any hidden plaintexts.

The default Android FDE uses CBC. We choose to move away from the Android default and instead, use XTS-AES [23, 34] to prevent known attacks against CBC. XTS-AES is a code book mode (i.e., no block chaining) and uses a secondary "tweak" key to make unpredictable use of the disk sector index. XTS-AES is not an authenticated mode, and as such is considered malleable [23]. However, unlike CBC, XTS is not malleable at a bit granularity: a modified ciphertext block will decrypt to a random plaintext block, preventing an attacker from making a predictable change. The absence of authentication tags also allows for a copy-and-paste attack (i.e., successful decryption of sectors that have been moved from other disk locations). Using CBC with random IVs will garble only the first block, but successfully decrypt all subsequent blocks in the moved sector. XTS-AES does not rely on block chaining, and uses the tweak to entangle plaintext/ciphertext block pairs with their disk sector location. As such, all blocks in a moved sector will decrypt to random plaintext. A watermark attack relies on predictable IVs, and is mounted by convincing the user to encrypt and store a file that has been specifically crafted to effectively zero out the IVs. The watermark manifests itself as identical ciphertext blocks at the beginning of consecutive disk sectors. The attacker can then examine the encrypted storage and locate the watermark. Both XTS-AES (Mobiflage) and CBC with ESSIV (Android FDE), effectively prevent watermarking attacks.

6.2 Leakage from Flash-storage

In this section, we provide an overview of flash storage technologies typically found in mobile devices. We also discuss flash leakage vectors that affect PDE and, to some extent, FDE.

Overview of flash storage. Mobile devices generally use NAND-based flash storage. Flash memory is not divided into sectors in the same way as magnetic disks. Write operations take place on a page level (e.g., 4KB page) and can only change information in one direction (e.g., changing 1 to 0, but not the inverse). Thus, write operations can only take place on an empty page. An erase operation takes place on a group of several pages, called an erase block (e.g., 128 pages per block). Flash memory cells have a finite number of program/erase cycles before becoming damaged and unusable. Therefore, flash memory is often used with a wear-leveling mechanism to prevent the same cell from being repeatedly written. In effect, logical block addresses (LBAs) on the disk are mapped to different physical memory pages for each write operation. Thus, storage on flash memory is not a linear arrangement as in traditional magnetic disks.

When a logical disk region is overwritten, it is usually simply remapped to an empty page without erasing the original page. This can continue until there are no empty pages, at which time unmapped pages in erase blocks are consolidated by the garbage collector, and empty erase blocks are wiped. Otherwise, the erase blocks must be completely wiped and rewritten to change a single page. This requires reading the entire erase block into cache, modifying the affected page, wiping the erase block, and finally writing the block back to the media.

Generally, two types of flash media are used in Android devices. Older Android devices use the memory technology device (MTD) for internal storage. An MTD is analogous to a block or character device, specifically designed for flash memory idiosyncrasies. To emulate a block device on an MTD, a software flash translation layer (FTL) is used. The FTL enables the use of a standard block file system (e.g., Ext4, FAT32) on top of the raw flash media. Newer Android devices use embedded multi media card (eMMC) for internal storage and secure digital (SD) for external storage. eMMC combines the flash memory and hardware controller in one package. SD has a dedicated controller and removable storage. Both technologies are presented to the system as block devices. The FTL for eMMC and SD storage is implemented in firmware on the controller as opposed to a software FTL as used by MTD.

Wear-leveling issues. Flash memory does not have the same data remanence issues as seen in magnetic storage. However, the wear-leveling mechanism may leave old copies, or fragments of files in unmapped pages on the flash disk. When making changes to hidden files it is possible that (encrypted) fragments of the original file will still exist in unmapped pages. This would provide an adversary with a partial time-line, or partial snapshots, of changes made to the disk. If the adversary can demonstrate that the regions affected do not coincide with disk activity in the outer volume, they can conclude existence of hidden volumes.

The software FTL used by the Linux MTD driver (`mtdblock`) is simplistic and does not use a wear-leveling mechanism.[4] Some file systems (e.g., YAFFS2) are designed to work directly with the raw flash memory instead of using an FTL. Such file systems may implement their own wear-leveling mechanisms. This was the default technology for devices prior to Android 3.0, but has largely been replaced by eMMC storage. The SD [41] and eMMC [25] specifications do not address wear-leveling requirements, so it is up to the manufacturers to decide if and how to implement wear-leveling in hardware FTLs.

[4] The MTD subsystem for Linux: http://www.linux-mtd.infradead.org/faq/general.html

Mobiflage stores all hidden volumes on the SD card. Therefore, exploiting the unmapped, wear-leveling pages would require bypassing the hardware controller and reading the raw flash memory, as opposed to acquiring a logical image (e.g., as produced with the `dd` tool). The adversary would need to read the physical to logical block allocation map and reconstruct the physical layout of the disk. Existing studies of raw flash performed by Wei et al. [49] have focused on writing specific strings to the media through the hardware controller FTL, then bypassing the controller to search for those strings in the raw physical flash. It is unknown how successful an adversary may be in demonstrating that a given unused page was part of a hidden volume and hence compromising deniability. Further work is needed to measure the extent to which unmapped/obsolete pages can be correlated to LBAs.

Mobile forensic tools that focus on logical data acquisition (e.g., viaExtract,[5] Paraben[6]) cannot mount this attack. Physical acquisition mechanisms exist for MTD storage (see e.g., Hoog [22, pp. 266–284]); however, they tend to be costly, time consuming, and generally destroy the mobile device.

Wear-leveling has implications for both non-deniable and deniable encryption schemes. If a disk is encrypted in-place, plaintext fragments that existed before encryption may still remain accessible. Wei et al. [49] show that most flash media contains between 6-25% more storage than advertised to the system. The additional storage is used by the wear-leveling mechanism. For this reason, Wei et al. suggest that the entire address space of a flash disk should be overwritten twice with random data, to ensure all erase blocks have been affected, before encrypting the device. Their findings show that in most cases, this is sufficient to ensure that every physical page on the device is overwritten. Therefore, Mobiflage performs a two-pass wipe, before encryption of the external partition, to avoid leaving any plaintext fragments on the media, and to ensure the continuity of random data, which is crucial for PDE. Currently, the default Android FDE does not take this precaution into consideration, and the *wipe* operation is performed by simply re-formatting the file system.

A recent proposal by Reardon et al. [38] explores secure deletion for flash memory. All file system data is encrypted with per-block keys. To securely delete a file system block, the associated key is wiped from the physical flash with an ERASE command. The data blocks are rendered un-readable, hence data remanence is not an issue. Currently, their implementation only works with MTD storage, and would need to be integrated into the SD/eMMC hardware controller FTL to afford secure deletion to these devices [38].

Special "discard" operation. The discard operation can be issued from a file system to the flash hardware controller. This command informs the host controller that a certain LBA is no longer storing file system data and can be wiped at any time. When all LBAs in an erase block are discarded, the controller's garbage collector will erase the block in the background. Discard effectively speeds up write access time, as an empty block can be directly written to without a read-modify-erase-write cycle. The ERASE command (or the TRIM command for ATA controllers) takes place on the physical layer, and when used, will zero out regions of the physical flash media, not visible to the logical file system. Thus, the adversary may recognize the physical blocks that are actually used to store file data. If the adversary knows the decoy key, he may correlate physical blocks and LBAs to discover which blocks are used by the hidden volume. As a security consideration, the dm-crypt mapper does not forward discard commands [5], hence ensuring the continuity of random data on the underlying physical storage.

6.3 Leakage from File-system and OS

The default file system for the internal storage in Android 4.x devices is Ext4. Ext4 introduced several new features including extents, uninitialized block groups, and flexible block groups [28]. The flexible block group (flex group) feature allows a block group's meta-data e.g., inode/block bitmaps and inode table, to be located anywhere on the disk, instead of at the beginning of each individual block group, as in earlier Ext file systems. The default setup is to store the meta-data for 16 consecutive block groups in the first block group of each flex group. This would make it possible to hide files inside an empty flex group without overwriting any meta data for that group. However, as Ext4 places backup superblocks and group descriptor tables in some block groups within each flex group, any hidden data stored in a flex group could overwrite these structures.

Additionally, the absence of backup superblocks and group descriptor tables would be suspicious and give the adversary reason to assume that data has been hidden in these flex groups. The locations of the backup structures and file data would need to be known when creating hidden volumes. Furthermore, when creating directories in the root of an Ext4 file system, the directories are placed in the most vacant block group available on the disk [27]. This effectively spreads directories, and the data contained within, across the entire disk. Standard volumes, unaware of the hidden volume location, will likely collide with hidden data regardless of where it is placed in an Ext4 file system. Therefore, we cannot reliably hide data within an Ext4 volume (without upstream changes in Ext4, e.g., by making directory spread optional).

[5] https://viaforensics.com

[6] http://www.paraben-forensics.com

One way to overcome the Ext4 backup superblock problem is to indicate those regions of the disk as damaged or "bad blocks" when creating the hidden volume. The hidden file system would then avoid writing data to those locations. When the outer volume is mounted there would be no indication of tampering nor reason for suspicion. Unfortunately, due to the Ext4 directory spread, this would not be a feasible solution for Android MTP devices without removable storage. However, this method may be used to implement PDE in other file systems such as NTFS, HFS+ and Ext2/3 that employ a sequential write policy (i.e., they do not use a directory spread mechanism as in Ext4). Another partial solution is to logically partition the internal storage to include a FAT32 volume. In the standard mode, this volume would be mounted to the SD card mount point, instead of using MTP. This partition would house the hidden volumes mounted in the PDE mode. MTP would be sacrificed in favor of the older USB mass storage functionality when connected to a PC.

Most work in deniable disk encryption investigates data or existence leakage of hidden files into temporary files, swap space, or OS logs (see e.g., [10]). For example, a word processor that performs auto-save functions to a central location may have backups and fragments of files edited from a hidden volume. If such backups are present, and no evidence of the files are found on the disk, then the adversary can assume the existence of hidden files and demand the true decryption key. We explain in Section 7 (item (c)) that log files, swap space, and temporary storage are effectively isolated between the two modes of Mobiflage.

7 Security Analysis

In this section, we evaluate Mobiflage against known attacks and weaknesses.

(a) Password guessing. We rely on the user to choose strong passwords to protect their encryption keys. The current Android encryption pre-boot authentication times-out for 30 seconds after ten failed password attempts. The timeout will slow an online guessing attack, but it may still be feasible, especially when weak passwords are used.

An offline dictionary attack is also possible on an image of the device's storage. The adversary does not know the password to derive the offset, but the salt is found in the Android encryption footer. The salt is used with PBKDF2, and is a precaution against pre-generated dictionaries and rainbow tables. The salt cannot be stored at the hidden offset as it is used in the offset calculation. Using the same salt value for both modes enables the adversary to compute one dictionary of candidate keys (after learning the salt), to crack passwords for both modes. Exacerbating the problem is Android's low PBKDF2 iteration count. On a single core of an Intel i7-2600, at 2000 iterations, we were able to calculate 513.37 ± 1.93 keys per second using the OpenSSL 1.0.1 library. Custom hardware (e.g., FPGA/GPU arrays) and adapted hash implementations (e.g., [48]) can make offline guessing even more efficient.

We tested different hash iteration counts in PBKDF2 and found that 200,000 iterations is apparently a fair compromise between security and login delay. On the Intel i7-2600, at 200,000 iterations, we were able to calculate 5.21 ± 0.01 keys per second (i.e., guessing attack becomes 100 times slower). On our Nexus S (1GHz Exynos-3 Cortex-A8) development phone, it required an additional 0.67 ± 0.01 seconds to calculate a single key. Our Motorola Xoom (1GHz Tegra-T20 Cortex-A9) required an additional 0.41 ± 0.001 seconds and an HTC EVO3D (1.2GHz MTM8660 Scorpion) required an additional 0.70 ± 0.01 seconds. This would slow down the boot procedure by approximately two seconds; note that, booting into the PDE mode requires three invocations of PBKDF2: to test the key in the footer, to calculate the offset, and to decrypt the hidden volume key. Possible computational and memory-wise expensive replacements for PBKDF2 (e.g., [31, 37]) can also be used to mitigate custom hardware attacks. In the end, we require users to choose a strong password resilient to guessing.

(b) Cipher issues. An implementation flaw can expose FDE ciphers to a theoretical watermarking attack that has been documented for software such as LUKS [9]. The issue occurs when the disk is sufficiently large and the size of the disk sector index (n) is small. For example, if n is a 32-bit integer, and there are more than 2^{32} 512-byte sectors on the disk, the value of n will eventually roll-over and repeat itself. If the adversary can create a special file with duplicate plaintext blocks at correct locations and convince the user to store the file in their hidden volume, then the adversary can demonstrate the existence of a hidden volume. In the given example, the duplicate plaintext blocks would need to be repeated at 2TB intervals. The adversary will not know what the corresponding ciphertext blocks will be, but finding identical ciphertexts spaced at the correct distance would be strong evidence. This is an implementation issue, and not an issue with the cipher algorithm itself. The problem occurs for all FDE ciphers, including XTS and CBC-ESSIV, that use a sector index smaller than the total number of disk sectors. To mitigate this problem, a longer integer (e.g., 64-bit) is commonly used for the sector index. We use the 64-bit sector index available in dm-crypt which will not roll over until 8192 Exabytes.

(c) Software issues. Mobiflage seems to effectively isolate the outer and hidden volumes. Apps and files installed in the hidden volumes leave no traces in the outer volume. Android does not use dedicated swap space. When the OS needs more RAM for the foreground app, it does not page entire regions of memory to the disk. Instead, it unloads background apps after copying a small state to the userdata partition. For example, the web browser may copy the current URLs of open tabs to disk when unloading, instead of the entire rendered page. When the browser is loaded again, the URL is reloaded. Leakage into swap space and paging files was shown to be an issue for desktop PDE implementations by Czeskis et al. [10]. As the outer and hidden userdata partitions are isolated from one another in Mobiflage, we do not take any specific measures against leakage through memory paging.

The Android Framework is stored in the `/system` partition which is mounted read-only. The Linux kernel is stored in a read-only boot partition which is not mounted onto the OS file system. Leakage through these immutable partitions is also unlikely.

Android logs are stored in a RAM buffer, and application logs are stored in the userdata partition. Leakage is also unlikely through logs as the userdata partitions are isolated and RAM is cleared when the device is powered off. Some devices keep persistent logs at `/devlog`, for troubleshooting between boots. To prevent leakage through these logs, we mount a tmpfs RAM disk to this mount point when booting into the PDE mode. The logs will remain persistent between standard mode boots, but no PDE mode logs are kept.

Android devices typically have a persistent cache partition used for temporary storage. For example, the Google Play store will download application packages to this partition before installing them on the userdata volume. To prevent leakage through the cache partition, we mount a tmpfs RAM disk to `/cache` in the PDE mode; this partition takes 32MB of RAM. An alternative to tmpfs, without sacrificing RAM, is to mount the volume through dm-crypt with a randomly generated one-time key. The key is discarded on reboot, effectively destroying the data on the partition.

(d) Partial storage snapshots. If the adversary has intermittent or regular access to the disk, they may be able to detect modifications to different regions of the disk. If a decoy key has already been divulged, the adversary may surmise the existence of hidden data by correlating file system activities to the changing disk regions. We exclude this possibility assuming the adversary will have access only after acquiring the device from the user, and does not have past snapshots of the storage. If the user is aware that the storage has been imaged (e.g., at a border crossing), they should re-initialize Mobiflage to alter every sector on the disk.

(e) Practical security of multiple hidden volumes. There is some debate over the effectiveness of multiple hidden volumes [15]. Whether or not the user gains any advantage is defined by the scenario. If the user cannot be held indefinitely, and cannot be punished on the suspicion of PDE data alone, she may feign compliance by relinquishing decoy keys. This may be advantageous to the user as, in the absence of indisputable evidence, she will eventually be released. In other scenarios, revealing the existence of one hidden volume may cause the adversary to suspect the existence of additional hidden volumes. If they can hold the user indefinitely, then they can continue to demand keys. It may in fact hinder the user to reveal any hidden volumes in this situation. However, irrespective of multiple hidden volumes, the adversary can keep punishing a suspect up until the *true* password is revealed. This is an inherent limitation of PDE schemes and may be alleviated (to some extent) by using a special password to make the hidden data permanently inaccessible (cf. [15]).

Cipher-spec	Key-length (bits)	Speed (KB/s) Nexus S	Speed (KB/s) Xoom
Unencrypted	N/A	5880±260	4767±238
AES-CBC-ESSIV (Android 4.x)	128	5559±76	4168±186
AES-XTS-Plain64 (Mobiflage)	512	5288±69	3929±146

Table 2: Read/write performance comparison

8 Performance Evaluation

To understand the performance impact on the regular use of a device, we run several tests on our prototype implementation of Mobiflage. This section summarizes our findings.

We use Mobiflage on Nexus S and Motorola Xoom development devices by reading from and writing to the SD card. The command-line tool `cp` is used to duplicate files on the SD card. We run 20 trials on four files between 50MB and 200MB. We evaluate the performance on unencrypted storage, under the default Android encryption, and the Mobiflage scheme. Table 2 summarizes our results.

Note that, removable SD storage (as in the Xoom) is apparently much slower than eMMC storage (as in the Nexus S), for all cases. Compared to the unencrypted case, on our Nexus S, Mobiflage reduces IO throughput by almost 10%; in contrast, Android FDE reduces the throughput by 5.5%. On the Motorola Xoom, Mobiflage reduces throughput by 17.6% and Android FDE by 12.6%. Mobiflage seems to decrease throughput by roughly 5% over Android FDE. However, the decreased IO throughput is negligible for regular apps and should not hinder the use of the device. For example, a standard definition 30fps video file may have a combined audio/video bit-rate of 192 KBps. High definition video (e.g., Netflix) is generally below 1024 KBps. The reduced speed of Mobiflage (3929 KBps) will still provide adequate buffering to ensure that jitter will not be an issue in these video apps. Note that Blu-ray has a maximum bitrate of 5000 KBps and may cause playback issues, if it is not first re-encoded. The observed decrease in throughput may be attributed to the chosen cipher: XTS requires two AES operations per block; and AES-256 uses fourteen rounds of operations while AES-128 uses ten.

Android apps are first loaded into RAM and do not run directly off the disk. Mobiflage should not affect run time performance of apps. The increase in app load time should also be practically negligible; as of Sept. 2012, the average Android app size is about 6MB [24], although the size of certain apps (e.g., gaming) is increasing rapidly. Some hardware, such as the camera, may use direct memory access (DMA) and may be affected: instead of writing directly to the disk, the camera data is processed by the CPU when passing through the dm-crypt layer. We tested the camera on our Nexus S device while in the Mobiflage PDE mode, and did not notice any performance impact.

The required time to encrypt the device is increased on account of the two pass random wipe. The exact time will depend on the size of the external storage partition. Android FDE encrypts *external* eMMC partitions in-place. As such, Mobiflage will take twice as long to encrypt these partitions. Removable SD cards are not encrypted by Android FDE, so we cannot provide a static comparison. Our Nexus S has only 1GB internal, and 15GB eMMC external storage. After three initializations, we found that on average the default Android FDE required one hour and five minutes, and Mobiflage required just under two hours. The Motorola Xoom required one hour and fifteen minutes on average for the default Android FDE to encrypt the 32GB internal storage. Encrypting with Mobiflage required an additional 73 minutes when used with a 8GB SanDisk SD card.

Power consumption will likely be increased for disk activity. This problem is inherent to all FDE, and is not unique to Mobiflage. Background processes that have high IO activity should be disabled, or IO should be buffered and batched to reduce power consumption.

9 Related Work

In this section, we discuss deniable encryption implementations related to Mobiflage, and provide an overview of available data encryption support as built into major desktop and mobile OSes. For details on deniable-storage proposals, see Appendix A. Several new ciphers, or enhancements to existing ciphers, have been proposed to create PDE schemes (e.g., [6, 35, 14, 29]). However, most of these proposals strive to enable PDE in network communications, and are not directly applicable to storage encryption.

All major desktop OSes now offer storage encryption with FDE support (e.g., Windows BitLocker, Mac OS X FileVault, and Linux eCryptfs). FDE uses ciphers to encrypt entire storage devices or partitions thereof. Encryption is performed on small units, such as sectors or clusters, to allow random access to the disk. FDE subsystems typically exist at or below the file system layer and provide transparent functionality to the user. FDE schemes generally focus on providing strong confidentiality, making efficient use of the storage media (i.e., no excessive data expansion), and being relatively fast (i.e., no significant decrease in IO throughput). PDE adds another layer of secrecy over FDE.

Most mobile OSes also offer data encryption (but no PDE). BlackBerry devices use a password derived key to encrypt an internal storage AES key, and an ECC private key [39]. When a device is locked, the storage and ECC keys are wiped from RAM. Any messages received while the device is locked are encrypted with the ECC public key, and decrypted after unlock. Removable storage can also be encrypted. Per-file keys are generated and wrapped with a password derived key, and/or a key stored in the internal storage. iOS devices use a UID (device unique identifier) derived key to encrypt file system meta-data, effectively tying the encrypted storage to a particular device [2]. Per-file keys are stored in this meta-data and used to encrypt file contents. File keys can be wrapped with a UID derived key, or a UID and password derived key, depending on the situation (e.g., if the file must be opened while the device is locked, only a UID key is used). Unlike the transparency afforded by FDE, app developers must explicitly call the encryption API to protect app data [47]. The advantage of file based encryption over FDE is that the device is actually *encrypted* when the screen is locked (i.e., keys are wiped from RAM). Older Android 2.3 (Gingerbread) devices can make use of third party software (e.g., WhisperCore [51]) to encrypt the device storage. WhisperCore enhances the raw flash file system, YAFFS2, which has been superseded on current Android devices in favor of the Ext4 file system.

Disk encryption software such as TrueCrypt [46] and FreeOTFE [17] use hidden volumes for plausible deniability. TrueCrypt offers encryption under several ciphers including AES, TwoFish, Serpent, and cascades of these ciphers in the XTS mode. On Windows systems, TrueCrypt can encrypt the OS system partition. A special boot loader is used to obtain the user's password and decrypt the disk before the OS is loaded. On Linux systems, similar functionality can be achieved using an early user-space RAM disk. This is not a straightforward solution for Android devices since the soft keyboard mechanism required to obtain the password is part of the OS framework and not immediately available on boot. A custom bootloader, implementing a soft keyboard, would be needed to capture the password (cf. [42]). The dm-crypt volume could then be mounted before loading the Android framework. We choose instead to work with the existing Android technique of partially loading the framework to access the built-in keyboard.

TrueCrypt volumes contain a header at the very beginning of a volume. All fields in the header are either random data (e.g., salt) or are encrypted, giving the appearance of uniform random data for the entire volume. Unlike Android FDE, the cipher specification is not stored. Therefore, when a TrueCrypt volume is loaded, all supported ciphers and cascades of ciphers, are tried until a certain block in the header decrypts to the ASCII string "TRUE". The header key is derived from the user's passphrase using PBKDF2. If the header key successfully decrypts the ASCII string, then it is used to decrypt the master volume key, which is chosen at random during the volume's creation.

A secondary header, adjacent to the primary header, is used when a hidden volume exists. The secondary header contains the same fields as the primary header, along with the offset to the hidden partition. When mounting a TrueCrypt volume, the hidden header is tested before the primary header. To combat leakage, when using hidden volumes, TrueCrypt recommends the use of a hidden OS. The hidden OS is currently only an option for the Windows implementation. When encrypting a system volume for use with PDE, TrueCrypt creates a second partition and copies the currently installed OS to the hidden volume within. The user should only mount hidden volumes when booted into a hidden OS, to ensure that any OS/application-specific leakage stays within a deniable volume (e.g., logs, page file, hibernation file). When booted into a hidden OS, all unencrypted volumes and non-hidden encrypted volumes are mounted read-only. The alternative to a hidden OS for Linux, is to use a live CD when mounting hidden volumes. A hidden OS is not necessary in Mobiflage since the system volume on an Android device is mounted read-only, and we attach hidden volumes, or RAM disks, to all mutable volume mount-points to prevent leakage.

There is a recent effort to port TrueCrypt to Android [8]. The current version (Dec. 2012) provides a command-line utility to create and mount TrueCrypt volume-container files (for rooted devices with LVM and FUSE kernel support). Hidden volumes are possible within these container files; but FDE/pre-boot authentication is not currently supported. Several leakage vectors also remain unaddressed (e.g., through file system structures, software logs, and network interfaces).

Other Linux deniable implementations, such as RubberhoseFS [3], and Magikfs,[7] employ techniques similar to StegFS for hiding data in file system free space; see Appendix A. Several of these projects are no longer maintained and existing implementations are also mostly incompatible with the modern Linux OS. The presence of specialized file system drivers designed to hide data would be a red flag to an adversary.

10 Concluding Remarks

Mobile devices are increasingly being used for capturing and spreading images of popular uprisings and civil disobedience. To keep such records hidden from authorities, deniable storage encryption may offer a viable technical solution. Such PDE-enabled storage systems exist for mainstream desktop/laptop operating systems. With Mobiflage, we explore design and implementation challenges of PDE for mobile devices, which may be more useful to regular users and human rights activists. Mobiflage's design is partly based on the lessons learned from known attacks and weaknesses of desktop PDE solutions. We also consider unique challenges in the mobile environment (such as ISP or wireless carrier collusion with the adversary). To address some of these challenges, we need the user to comply with certain requirements. We compiled a list of rules the user must follow to prevent leakage of information that may weaken deniability. Even if users follow all these guidelines, we do not claim that Mobiflage's design is completely safe against any leaks (cf. [10]). We want to avoid giving any false sense of security. We present Mobiflage here to encourage further investigation of PDE-enabled mobile systems. Source code of our prototype implementation is available on request.

Acknowledgements

We are grateful to our shepherds William Enck and Thomas Ristenpart, and anonymous NDSS2013 reviewers for their insightful suggestions and advice. We also appreciate the helpful feedback from Jeremy Clark and David Barrera. We thank members of the Concordia's Computer Security Lab for their enthusiastic discussion on this topic. The second author is supported in part by an NSERC Discovery Grant and Concordia University Start-up Program.

References

[1] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *International Workshop on Information Hiding (IH'98)*, Portland, Oregon, USA, 1998.

[2] Apple. iOS security. Technical document (May 2012). http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf.

[3] J. Assange, R.-P. Weinmann, and S. Dreyfus. Rubberhose: Cryptographically deniable transparent disk encryption system. Project website: http://marutukku.org/.

[4] X. Boyen. Halting password puzzles: Hard-to-break encryption from human-memorable keys. In *USENIX Security Symposium*, Boston, MA, USA, 2007.

[5] M. Broz and A. G. Kergon. dm-crypt: optionally support discard requests. Patch documentation (Aug. 2011). https://github.com/torvalds/linux/commit/772ae5f54d69c38a5e3c4352c5fdbdaff141af21.

[7] Magikfs http://magikfs.sourceforge.net/

[6] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable encryption. In *CRYPTO'97*, Santa Barbara, CA, USA, 1997.

[7] comScore. comScore reports September 2012 U.S. mobile subscriber market share. Press release (Nov. 2, 2012).

[8] Cryptonite. EncFS and TrueCrypt on Android. Open-source project (2012). https://code.google.com/p/cryptonite/.

[9] cryptsetup. Setup virtual encryption devices under dm-crypt Linux. Online document (July 2012). https://code.google.com/p/cryptsetup/wiki/FrequentlyAskedQuestions.

[10] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier. Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS and applications. In *USENIX Workshop on Hot Topics in Security (HotSec'08)*, San Jose, CA, USA, 2008.

[11] Dailymail.co.uk. Government spy programme will monitor every phone call, text and email... and details will be kept for up to a year. News article (Feb. 20, 2012).

[12] DMCrypt. dm-crypt: Linux kernel device mapper crypto target. Online document (July 2012). https://code.google.com/p/cryptsetup/wiki/DMCrypt.

[13] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, Mar. 1983.

[14] M. Dürmuth and D. Freeman. Deniable encryption with negligible detection probability: An interactive construction. In *Eurocrypt*, Tallinn, Estonia, 2011.

[15] EKR. Protecting your encrypted data in the face of coercion. Blog post (Feb. 11, 2012). http://www.educatedguesswork.org/2012/02/protecting_your_encrypted_data.html.

[16] N. Ferguson, B. Schneier, and T. Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, Mar. 2010.

[17] FreeOTFE. FreeOTFE - Free disk encryption software for PCs and PDAs. Version 5.21 (Nov. 2012). http://www.freeotfe.org/.

[18] C. Fruhwirth. New methods in hard disk encryption. Technical report, Vienna University of Technology (July 2005). http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf.

[19] C. Fruhwirth. TKS1 – an anti-forensic, two level, and iterated key setup scheme. Online manuscript (july 2004). http://clemens.endorphin.org/TKS1-draft.pdf.

[20] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, San Jose, CA, USA, 2008.

[21] J. Han, M. Pan, D. Gao, and H. Pang. A multi-user steganographic file system on untrusted shared storage. In *Annual Computer Security Applications Conference (ACSAC'10)*, Orlando, Florida, USA, 2010.

[22] A. Hoog. *Android Forensics: Investigation, Analysis, and Mobile Security for Google Android*. Syngress (Elsevier), June 2011.

[23] IEEE Computer Society. IEEE standard for cryptographic protection of data on block-oriented storage devices. IEEE Std 1619-2007 (Apr. 2008).

[24] ITProPortal. Android and iOS app sizes rise dramatically. News article (Oct. 18, 2012). http://www.itproportal.com/2012/10/18/android-and-ios-app-sizes-rise-dramatically/.

[25] JEDEC. eMMC card product std v4.41 (JESD84-A441). Technical specification (Mar. 2010). http://www.jedec.org/sites/default/files/docs/JESD84-A441.pdf.

[26] B. Kaliski. PKCS #5: Password-based cryptography specification, version 2.0, Sept. 2000. RFC 2898 (informational).

[27] kernel.org. Ext4 disk layout. Online document (July 2012). https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[28] kernel.org. Ext4 specification. Online document (July 2012). http://kernel.org/doc/Documentation/filesystems/ext4.txt.

[29] M. Klonowski, P. Kubiak, and M. Kutyowski. Practical deniable encryption. In *Theory and Practice of Computer Science (SOFSEM'08)*, Novy Smokovec, Slovakia, 2008.

[30] B.-J. Koops. Crypto law survey: Overview per country. Online document (version 26.0, July 2010). http://rechten.uvt.nl/koops/cryptolaw/cls2.htm.

[31] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *CRYPTO'10*, Santa Barbara, CA, USA, 2010. Also published as RFC 5869.

[32] A. D. McDonald and M. G. Kuhn. StegFS: A steganographic file system for Linux. In *International Workshop on Information Hiding (IH'99)*, Dresden, Germany, 2000.

[33] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[34] NIST. Recommendation for block cipher modes of operation: The XTS-AES mode for confidentiality on storage devices. NIST Special Publication 800-38E (Jan. 2010).

[35] A. ONeill, C. Peikert, and B. Waters. Bi-deniable public-key encryption. In *CRYPTO'11*, Santa Barbara, CA, USA, 2011.

[36] H. Pang, K. lee Tan, and X. Zhou. StegFS: A steganographic file system. In *International Conference on Data Engineering (ICDE'02)*, San Jose, CA, USA, 2002.

[37] C. Percival. Stronger key derivation via sequential memory-hard functions. In *BSD Conference (BSDCan'09)*, Ottawa, Canada, 2009.

[38] J. Reardon, S. Capkun, and D. Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *USENIX Security Symposium*, Bellevue, WA, USA, 2012.

[39] RIM. Blackberry enterprise server 5.0.2–security technical overview. Technical document (Mar. 2011). http://docs.blackberry.com/en/admin/deliverables/16648/.

[40] P. Rogaway. Nonce-based symmetric encryption. In *Workshop on Fast Software Encryption (FSE'04)*, volume 3017 of *LNCS*, pages 348–358, Delhi, India, 2004.

[41] SD Card Association. Physical layer simplified specification ver3.01. Technical specification (May 2010). https://www.sdcard.org/downloads/pls/simplified_specs/.

[42] TeamWin. TeamWin recovery project (TWRP). Version 2.3.1 http://teamw.in/project/twrp2.

[43] TheRegister.co.uk. UK jails schizophrenic for refusal to decrypt files. News article (Nov. 24, 2009). http://www.theregister.co.uk/2009/11/24/ripa_jfl/.

[44] TheRegister.co.uk. Youth jailed for not handing over encryption password. News article (Oct. 6, 2010). http://www.theregister.co.uk/2010/10/06/jail_password_ripa/.

[45] Toronto Star. How a Syrian refugee risked his life to bear witness to atrocities. News article (Mar. 14, 2012). http://www.thestar.com/news/world/article/1145824.

[46] TrueCrypt. Free open source on-the-fly disk encryption software. Version 7.1a (July 2012). http://www.truecrypt.org/.

[47] K. Wadner. iOS security. Technical document (July 2011). http://www.sans.org/reading_room/whitepapers/pda/security-implications-ios_33724.

[48] B. Wallace. Transferable state attack on iterated hashing functions. Tech. document (July 2012). http://firebwall.com.

[49] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson. Reliably erasing data from flash-based solid state drives. In *USENIX File and Storage Technologies (FAST'11)*, San Jose, CA, USA, 2011.

[50] R.-P. Weinmann. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In *USENIX Workshop on Offensive Technologies (WOOT'12)*, Bellevue, WA, USA, 2012.

[51] WhisperSystems. WhisperCore: Device and data protection for Android. Beta version (0.5.5). http://whispersys.com/whispercore.html.

[52] X. Zhou, H. Pang, and K.-L. Tan. Hiding data accesses in steganographic file system. In *International Conference on Data Engineering (ICDE'03)*, Bangalore, India, 2003.

A Deniable Storage Encryption Proposals

File encryption schemes with PDE support, called steganographic file systems, have been first proposed by Anderson et al. [1]. One of their solutions uses a series of cover files initially filled with random data, and assumes the attacker has no knowledge of the plaintext content of a file. The hidden files are embedded by modifying and XORing a linear combination of some cover files. The password and file name are used to determine which cover files are used. This solution requires storing a large number of cover files (e.g., 1000); also, these files must be relatively large to accommodate files of arbitrary length. The second solution [1] is built on existing block ciphers. The disk is initially filled with random data. Files are then stored at disk block addresses derived from the file name and password (e.g., using a hash function). The files are encrypted with a key derived in a similar manner. An adversary would not be able to distinguish between empty blocks and blocks containing hidden files. However, as discussed [1], the probability of file blocks colliding increases as disk blocks are filled. As a mitigation, writing each block to several disk locations has been suggested. However, high storage and IO overhead of these solutions make them less suitable for performance-sensitive mobile devices.

StegFS [32] is an Ext2 based file system inspired by the second approach of Anderson et al. [1]. It uses several security levels (up to 15), each with a separate password. Its deniability relies on how many levels of hidden files are present, not on denying the fact that hidden files exist. An external block allocation table (stored in the non-deniable disk space) with entries for each disk block is used. When a given security level is closed, there is no way to prevent overwriting that level's blocks, so redundant blocks are used to mitigate collisions. The existence of the modified Ext2 driver, and the external block table, would indicate that PDE is in use. The project website[8] explains that only 6% of the storage space can actually be used for file storage, as the rest is used for meta-data and collision avoidance. Also, as hidden and regular files are present on the same file system, data leakage may occur when security levels are open.

Other StegFS-based systems improve efficiency and reliability of the original implementation. Pang et al. [36] design a system where blocks used by hidden files are in fact marked as occupied in the block bitmap. This alleviates reliability issues and IO inefficiencies, as storing multiple copies of a block is not required. Hidden files do not have a directory record in the standard inode table. Since the blocks are marked as used, but not referenced in a directory entry, the adversary can conjecture that hidden files exist. The adversary can also estimate the amount of disk space utilized by hidden files. Three mechanisms are used to frustrate such estimation. Some empty or "abandoned" blocks are marked as used even though they do not contain hidden data. When a new hidden file is created, several blocks are allocated that are not actually filled with file data. Dummy hidden files are created and periodically updated in the background to prevent snapshot analysis from determining the exact blocks used by hidden files. These mechanisms make it more difficult to determine which blocks actually store hidden data, but are not disk space efficient.

Further work [52] expands the above idea by adding dummy transactions to obscure hidden files in network/cloud storage. This improves reliability and IO efficiency, but disk space utilization for dummy files and abandoned blocks remains a concern, especially for resource constrained mobile devices. Also, strong deniability cannot be offered as the adversary is aware that hidden files exist. Deniability is a result of an adversary being unable to determine how much space is used by hidden files.

The dummy-relocatable steganographic (DRSteg [21]) file system is proposed for use in multi-user environments. DRSteg adds dynamic updating to dummy files to prevent snapshot analysis. When coerced, a user can provide some of their hidden file passwords and blame the additional hidden storage on dummy files and other users' hidden files. However, the adversary is still aware that hidden files exist.

[8] StegFS https://albinoloverats.net/projects/stegfs