Mobiflage: Deniable Storage Encryption for Mobile Devices

Adam Skillen and Mohammad Mannan

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. Steganographic techniques and deniable encryption algorithms have been devised to hide the very existence of encrypted data. We examine the feasibility and efficacy of deniable encryption for mobile devices. To address obstacles that can compromise plausibly deniable encryption (PDE) in a mobile environment, we design a system called Mobiflage. Mobiflage enables PDE on mobile devices by hiding encrypted volumes within random data in a devices free storage space. We leverage lessons learned from deniable encryption in the desktop environment, and design new countermeasures for threats specific to mobile systems. We provide two implementations for the Android OS, to assess the feasibility and performance of Mobiflage on different hardware profiles. MF-SD is designed for use on devices with FAT32 removable SD cards. Our MF-MTP variant supports devices that instead share a single internal partition for both apps and user accessible data. MF-MTP leverages certain Ext4 file system mechanisms and uses an adjusted data-block allocator. These new techniques for storing hidden volumes in Ext4 file systems can also be applied to other file systems to enable deniable encryption for desktop OSes and other mobile platforms.

Index Terms—File system security, Mobile platform security, Storage Encryption, Deniable encryption

1 Introduction and Motivation

Smartphones and other mobile computing devices are being widely adopted globally. For instance, according to a comScore report [2], 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. [3] 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 file system for Linux (developed by Assange et al. [4]) 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 [5], 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 [6]. 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

This work is the extension of an NDSS 2013 publication [1].

- *A. Skillen is currently with the Carleton Computer Security Lab at Carleton University, Ottawa, Canada, and was at Concordia University, Montreal, Canada when performing this research. E-mail: askillen@ccsl.carleton.ca*
- *M. Mannan is with the Concordia Institute for Information Systems Engineering at Concordia University, Montreal, Canada. E-mail: m.mannan@concordia.ca.*

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., [7]). 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 introduce two variants of Mobiflage to address several challenges specific to different Android hardware profiles. Mobiflage for devices with removable SD cards (MF-SD) avoids PDE-unfriendly features of the Ext4 file system by storing hidden volumes within the FAT32-based external partition. Devices, such as the Nexus S, which use an internal eMMC partition to emulate removable SD storage are also supported by MF-SD. Newer devices, such as the Nexus 4 and HTC One, have neither physical nor emulated external storage. Instead, they rely on the media transfer protocol (MTP) and share a single Ext4-formatted partition for both the (*internal*) app storage and (*external*) user data storage. To support these devices, we present MF-MTP, by making subtle changes to the Ext4 file system. For the remainder of the document, the term Mobiflage will refer to the high level design. The terms MF-SD and MF-MTP will refer to the specific implementation variants.
4) We provide proof-of-concept implementations of our Mobiflage variants for Android 4.x (Ice Cream Sandwich and Jelly Bean). We incorporated our changes into 4.x as an optional feature (i.e., encryption without PDE is still available). There are no identifying technical differences between an instantiation of the default and PDE encryption modes.
5) We analyze the performance impact of our implementation during initialization and for data-intensive applications. We also perform file system benchmarks to determine the impact of our modified Ext4 driver. 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 storage (our Nexus S required almost twice as long; exact timing will depend on the size and type of storage).

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 is mostly based on past work on desktop PDE solutions (cf. TrueCrypt [6]); we also include threats more specific to mobile devices.

Threat model and operational assumptions.

1) Mobiflage must be merged with the 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 [6], all installations of Mobiflage include PDE capabilities.
2) The adversary has the encrypted device and full knowledge of Mobiflage's design, but lacks the PDE key and password. The existence and location of the hidden volume is therefore also unknown.
3) 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 in vain. To successfully provide deniability in Mobiflage, the user is expected to refrain from disclosing the true key.
4) The adversary can directly access the device's 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 [8]). Mobiflage addresses these issues.
5) The adversary model of desktop FDE usually includes the ability to periodically snapshot the encrypted physical storage (cf. [7]). However, this assumption is unlikely for mobile devices and has therefore been relaxed (as the adversary will have access to the storage only after seizing the user).

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

6) In addition to the Dolev-Yao network attacker model [9], [10], 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 activity logs from these carriers to reveal the use of a PDE mode on suspected devices.
7) 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. 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 [11], which may be used to reveal the PDE mode.
8) We assume the adversary cannot capture the user device while in the PDE mode; otherwise, user data can be trivially retrieved from the device. We require the user to follow certain guidelines, e.g., not using Mobiflage's PDE-mode for regular use; other precautions are discussed in the NDSS 2013 publication [1].

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., [12], [13]). 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 [14].

Some of our recommendations, such as spoofing the IMEI or using an anonymous 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 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. We differentiate between the design goals of MF-SD and MF-MTP. 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.

3.1 Overview and Modes of Operation

We implement Mobiflage by hiding volumes in empty space within a mobile device's storage. We first fill the storage with random data, to conceal the existence of additional encrypted volumes. In MF-SD, the hidden volumes are created in the device's external (SD or eMMC) storage partition. We store the hidden volumes in the external storage, due to certain file system limitations discussed in Section 3.2. We create two adjacent volumes: a userdata volume for apps and settings, and a larger auxiliary volume to house documents, photos, etc. MF-MTP creates A hidden volume in the device's internal (userdata) partition. Only one hidden volume is necessary for MF-MTP, since the MTP exposed storage simply appears as a separate volume when it is in fact a directory on the userdata partition. We make subtle changes to the Ext4 file system to overcome the PDE unfriendly features discussed in Section 3.2. The exact location of the hidden data for both variants is derived from the user's deniable password.

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 filesystem 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 the PDE mode.

3.2 File-system considerations

The default file system for the internal storage in Android 4.x devices is Ext4. Ext volumes are divided into block groups, which contain some meta-data (e.g., inode/block bitmaps and inode table) and many data blocks (e.g., 32768 blocks per block group). In order to effectively hide data in the free space of an Ext file system, we must avoid overwriting meta-data structures and occupied data blocks. After decrypting the outer volume, the adversary can examine the meta-data structures. The absence, or visible corruption, of outer volume meta-data would be suspicious and give the adversary reason to assume that hidden data has been stored in that region.

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 [15]. This effectively spreads directories, and the data contained within, across the entire disk. Data written to the outer volume will likely collide with hidden data (and vice versa) regardless of where it is placed in an Ext4 file system. Therefore, in order to reliably hide data within an Ext4 volume, modification of certain file system behavior was necessary for MF-MTP; see Section 4.

3.3 Steganography vs. Hidden Volumes

There are currently two main types of PDE systems for use with FDE: steganographic file systems (e.g., StegFS [16]) and hidden volumes (e.g., TrueCrypt [6]). 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 the discussion of deniable storage encryption proposals in the NDSS 2013 publication [1]) . Consequently, we choose to use hidden volumes for Mobiflage. This implies that 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 hidden volume offset.

3.4 Storage Layout

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

Since the outer volume is filled with random data before formatting, there are no distinguishing characteristics between empty outer-volume blocks and hidden volume blocks. However, some statistical deviations may be used to distinguish the random data from the cipher output; see Section 5.1. When the outer volume is decrypted and mounted, it does not reveal the existence or location of the hidden volume (i.e., the hidden volume is camouflaged amongst the random data). When the user is coerced, she can provide the decoy key and deny the existence of hidden data.

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 we cannot place a visible limit on the mounted volume.

3.5 Offset Calculation

The offset to a hidden volume is generated as follows:

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

Here, H is a PBKDF2 iterated hash function [17], $vlen$ is the number of allocation units 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 the need to store an additional salt value. 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. 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%. For MF-SD, $vlen$ is measured in 512-byte sectors, as this is the granularity used by the FDE engine. For MF-MTP, $vlen$ is measured in 4096-byte file system blocks, for easy alignment of volumes; see Section 4.2.

Deriving the offset in the above manner allows us to avoid storing it anywhere on the disk, which is undesirable 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 [8]) header. 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 4.5, item (1).

Alternatively, the offset could have been fixed at a given location on the disk (e.g., always appearing at 50%). However, there is a minor security benefit in deriving the offset as shown in above equation: it complicates a dictionary attack, by mandating the adversary capture a larger portion of the disk. If the offset was at a known location, then an adversary could perform a dictionary attack on a few kilobytes of data captured from that region (only the key and file system magic-number are necessary to prove the existence of a hidden volume). With our approach, the adversary must capture at least 25% of the storage to mount an attack. Note that the efficiency of a dictionary attack is not affected by the offset location (see Section 6, item (a)). Copying 25% of the storage may reduce the adversary's ability to process a large number of target users (e.g., all individuals passing through a customs checkpoint).

3.6 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). MF-SD's initialization phase erases existing data on the external storage (SD card). However, users can choose to preserve the internal userdata partition or initialize it with random data. MF-MTP must erase the internal storage partition to create the hidden volumes; this data should be backed-up before initiating Mobiflage (e.g., using the `adb backup` command, or a third-party tool). The user then enters the decoy and true passwords, for the outer and hidden volumes respectively. Mobiflage then formats, and encrypts the outer and hidden volumes; see Fig. 1. Unlike Android FDE, Mobiflage must initialize the storage with random data. This makes Mobiflage slower than the default Android FDE initialization (see Section 7). However, the initialization step will likely be performed only occasionally.

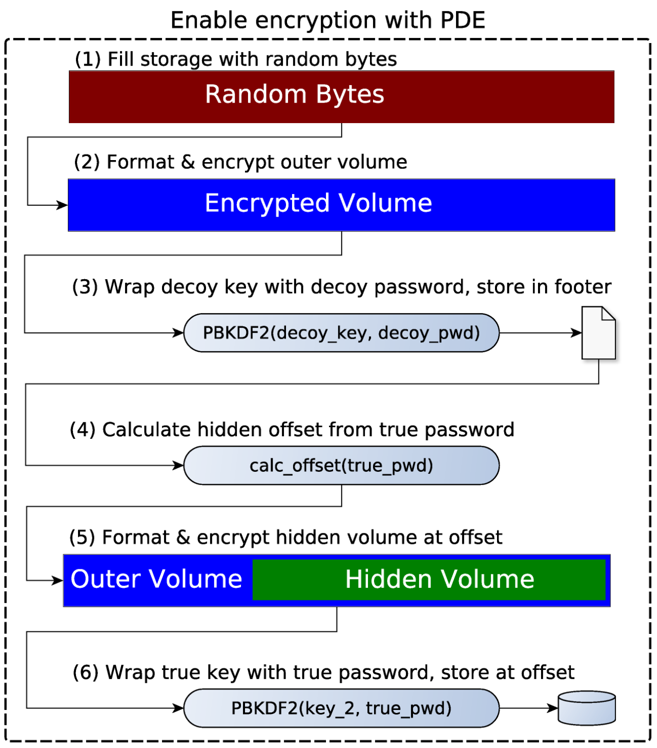
Enable encryption with PDE
(1) Fill storage with random bytes
Random Bytes
(2) Format & encrypt outer volume
Encrypted Volume
(3) Wrap decoy key with decoy password, store in footer
PBKDF2(decoy_key, decoy_pwd)
(4) Calculate hidden offset from true password
calc_offset(true_pwd)
(5) Format & encrypt hidden volume at offset
Outer Volume
Hidden Volume
(6) Wrap true key with true password, store at offset
PBKDF2(key_2, true_pwd)

Fig. 1: Mobiflage initialization process: only steps 2 and 3 are performed by the default Android FDE. Note there is no boundary between the outer and hidden volumes (i.e., their address space overlaps).

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; see Fig. 2. 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 time-line 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

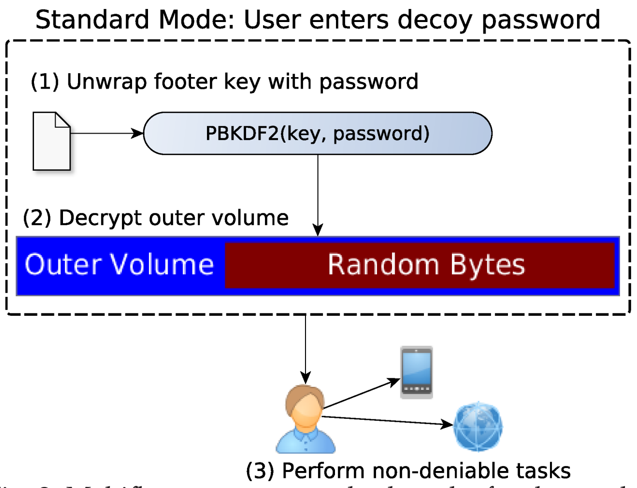
Standard Mode: User enters decoy password
(1) Unwrap footer key with password
PBKDF2(key, password)
(2) Decrypt outer volume
Outer Volume
Random Bytes
(3) Perform non-deniable tasks

Fig. 2: Mobiflage usage – standard mode: for day-to-day use of the device (i.e., no deniability). Note the hidden volume will appear as random bytes in the outer volume free space. The outer volume has no boundary - i.e., it occupies the entire storage medium.

their deniable password when prompted; see Fig. 3. 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 (see the NDSS 2013 publication [1].).

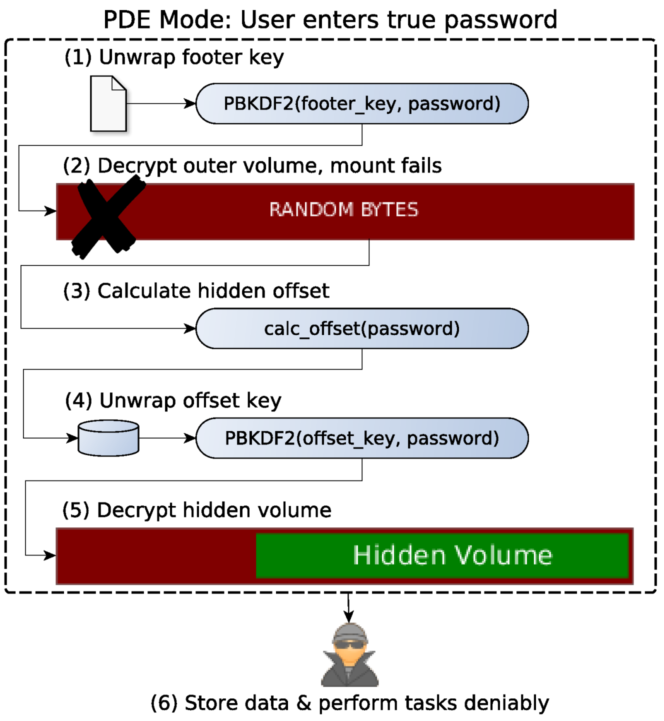
PDE Mode: User enters true password
(1) Unwrap footer key
PBKDF2(footer_key, password)
(2) Decrypt outer volume, mount fails
RANDOM BYTES
(3) Calculate hidden offset
calc_offset(password)
(4) Unwrap offset key
PBKDF2(offset_key, password)
(5) Decrypt hidden volume
Hidden Volume
(6) Store data & perform tasks deniably

Fig. 3: Mobiflage usage – PDE mode: the user can store data or perform tasks which can later be denied. In Step 3, mount fails because the password provided was incorrect for the outer volume. Instead of immediately prompting the user to try again, Mobiflage will attempt to decrypt the hidden volume using that password.

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 [18]).

If the user is apprehended with her device, she can supply the 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. Assuming the user does not reveal the true key, there will not be any evidence of the hidden data.

4 Mobiflage Implementation

We focus on the MF-MTP variant in this section and refer the reader to the NDSS 2013 publication [1] for details on MF-SD. Testing of MF-MTP was performed on a LG Nexus 4 device and the 4.2.2 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. Changes to the 3.4.0 kernel Ext4 driver, resulting in 56 additional lines of code, were necessary to implement MF-MTP. We also discuss current implementation limitations of Mobiflage, and compare MF-SD against MF-MTP.

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 [19]. 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 [8]; see also Section 5.1). A randomly chosen master volume key is encrypted with the same cipher by a key derived from 2000 iterations of the PBKDF2 [17] 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.

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 dm-crypt, 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 storage is not encrypted; however, emulated *external* storage (i.e., an internal eMMC partition, mounted at `/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) for MF-SD, we enable encryption of removable storage; and (c) we wipe the storage with random data. XTS-AES is chosen as a precaution against known attacks (e.g., the *copy-and-paste* attack see Section 5.1 for details). We use a 512-bit key (256-bit for AES and 256-bit for XEX tweak). 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. MF-SD performs the wipe operation on the SD card even when the user enables in-place encryption. MF-MTP does not allow in-place encryption, and always wipes the storage. 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 5.2 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 MF-MTP Implementation

We discuss the problem of creating hidden volumes within an Ext4 file system (or any complex file system with distributed meta-data and non-sequential block assignment) in Section 3.2. Since many new Android devices no longer provide an external storage partition, and instead use the MTP/shared-internal-storage paradigm with Ext4 file systems, we introduce two new mechanisms to contend with the distributed meta-data, and non-sequential block assignment characteristics of Ext4. Our hope is that MF-MTP can also provide a generalized PDE solution for other devices that lack removable storage (e.g., Apple iOS devices). Performance consequence of the modified driver are discussed in Section 7.

Bad-block marking. After creating the outer volume, and before creating the hidden volume, we calculate a list of the outer volume blocks that contain meta-data. Since Android does not use the flexible block group feature, all meta-data is contained in deterministic locations. The file system meta-data blocks are: **(a)** the backup superblocks and group descriptor tables in sparse block groups, and **(b)** the block/inode bitmaps and inode tables in all block groups. We overlay the outer volume meta-data blocks on the hidden volume file system, by subtracting the hidden offset from the outer volume block numbers, and pass the list of intersecting blocks to the `mke2fs` tool as *bad blocks* when formatting the hidden volume. We fix the size of blocks, block groups, and inodes when formatting, to easily calculate the quantity and location of meta-data blocks in each block group. The `mke2fs` tool will avoid writing any data to blocks marked as bad, hence preserving the outer volume meta-data blocks. One small caveat is that `mke2fs` will still attempt to write the backup superblocks to bad regions. This is likely due to the fact that backup superblocks are not referenced in the bad block inode or block bitmaps, and so cannot be marked as used to prevent data from being written to them. We discuss the implications of this issue further below under volume formatting.

Sequential inode allocator. Since the directory-spread feature of Ext file systems will likely cause block collisions between the outer and hidden volumes (see Section 3.2), MF-MTP must not be used with such a feature. Our Ext4 driver was modified to use a sequential inode allocator. The first block group with both a free inode and a free block is always selected, when creating a file. This is sufficient to ensure the disk is filled linearly, since the block allocator already tries to place data blocks within the same block group as a file's inode.

Volume alignment. To facilitate the bad-block marking mechanism, file system blocks in both volumes must be aligned. Therefore, we adjust the offset calculation (see Section 3.5) to work on 4096-byte file system blocks instead of 512-byte disk sectors. Note that although Ext4 supports a few different block sizes, we fix the block size at 4096 bytes to simplify our calculations.

The first three blocks of a volume must not be marked as bad when creating an Ext4 file system. The format operation will simply fail if the primary superblock and group descriptor table cannot be written to the first few blocks. Therefore, after marking the bad blocks, we check the alignment of the hidden volume, and ensure the first three hidden volume blocks do not collide with outer volume meta-data blocks (i.e., we ensure the first hidden volume blocks are not marked as bad blocks). If a collision is detected, we simply walk through the block indices until three sequential good blocks are found. There will be at most 516 bad (meta-data) blocks in a row, due to the block and block group sizes. We then adjust the size of the hidden volume by the number of blocks skipped, and proceed to format the volume at the new *clean* offset. When booting the device, this same calculation is performed to locate the clean offset.

Along with the primary superblock, the hidden volume offset itself must not fall on a bad block, since the hidden volume key must not collide with outer volume meta-data structures. The same procedure as above is used in this case. Since all operations are now block-aligned (instead of being sector aligned) we also need to alter how the PDE key is stored on the disk. There are two viable solutions: we can continue to place the key at the offset, then skip ahead $4096 - 512 = 3584$ bytes to format the volume at the next block. Otherwise, we can write the key to the sector preceding the offset, and format the volume at the offset proper. Both methods will ensure the hidden volume is block-aligned with the outer volume. We implemented the first method, despite the fact that a small amount of space is wasted, it maintains our stipulation that the hidden volume will consume no more than 50% of the disk (the second method places the key outside of the hidden region and could consume one sector more than 50%).

Volume formatting. Backup superblocks are written to (some) block groups, during the formatting of a volume. The *sparse-super* flag can be set during formatting, to create backup superblocks in some block groups instead of all block groups. The backup superblocks are the only meta-data structure that will ignore the presence of bad blocks, and attempt to write to the block even when marked as bad. In MF-SD, we write the outer volume first, and then the hidden volume. Due to the ignorant backup superblock policy, we instead write the hidden volume first, and then write the outer volume, to ensure the outer volume's meta-data remains unadulterated. We adjust all meta-data structures in the hidden volume to avoid being overwritten, except for the stubborn backup superblocks (note that all other meta-data structures will avoid writing to bad blocks, and instead seek ahead to the next good block). Therefore, it is only the backup superblocks in the hidden volume that may be overwritten if such a collision occurs; Overwriting the backup superblocks in the hidden volume is of no consequence, since these superblocks are only ever read or modified when attempting to manually mount the volume in an advanced recovery mode (which is currently unavailable to the Android FDE pre-boot authenticator).

Additional considerations. The following is a list of additional considerations that are important to the implementation, adoption, and usage of MF-MTP:

1) The userdata partition cannot be encrypted in-place with MF-MTP, since the entire storage space with random bytes during initialization (see Section 5.2 under the Wear-leveling heading). Users are responsible for backing up and restoring apps and data before initializing MF-MTP.
2) The e2fsprogs package was cross-compiled for the ARM CPU, as we required the `mke2fs` and `tune2fs` tools. The `make_ext4fs` tool, included with the default Android OS, does not support marking bad-blocks during format. The e2fsprogs package is distributed under the GNU GPLv2 license which may be in conflict with the Android Open Source Project, as it is provided under the Apache license. The required functionality could be independently implemented in the `make_ext4fs`

tool to overcome this issue.

3) The modified MF-MTP Ext4 driver may also be a feasible PDE solution for other Linux based OSes that use Ext4. The bad block marking and linear allocator could also be adapted for other complex file systems to enable PDE in a wide range of devices/platforms. However, the presence of the modified file system driver may be a red flag to an adversary. Incorporating the linear allocator upstream to Ext4 (i.e., by making directory spread an optional feature) may mitigate this concern; although we do not know the performance impact of such an option on magnetic disks at this time.

4.3 User Interface and Pre-boot Authentication

The default Android encryption mechanism can be enabled through the settings GUI. Currently, the user can activate Mobiflage PDE using the `vdc` command-line tool 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). Currently, the user can activate Mobiflage PDE using the `vdc` command-line tool 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).

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 footer 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 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 logical block after the offset. If a valid Ext4 file system is found at this location, it is mounted. After mounting the hidden volume, 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.

4.4 Limitations

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

1) 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. [20]).
2) 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 all remaining storage, but the address space would overlap with other hidden volumes.
3) Transferring data between outer and hidden volumes may be necessary on occasion; e.g., if time does not permit switching 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.
4) File system failures may occur in the outer volume that could corrupt hidden volume data. Furthermore, the user may write to the outer volume beyond the boundary and cause hidden volume corruption. Currently, correcting hidden volume errors requires mounting with a backup superblock and using tools such as e2fsck to attempt to recover the file system. Using such tools is possibly beyond the capabilities of average users. To prevent overwriting the hidden volume, the user could be prompted for the hidden volume password when mounting the outer volume. This would provide the hidden offset and allow write blocking beyond the boundary. However, this would also enable identifying a hidden volume when the outer volume is mounted. Currently, to avoid writing beyond the boundary, we recommend that the user does not fill the outer volume beyond 50%.

4.5 Comparison of Mobiflage Variants

Below we compare the benefits and drawbacks between the Mobiflage variants.

1) MF-SD is a cleaner solution than MF-MTP, as FAT32 accommodates hidden volumes natively, without the need for a modified file system driver.
2) MF-MTP does not fix the size of the application storage volume, `/data`, (e.g., fixed at 256MB) providing greater flexibility for installing hidden apps.
3) With MF-SD, it is possible to maintain several hidden environments, by initializing Mobiflage on several SD cards. Each SD card may contain different apps and data. This is not currently a possibility for MF-MTP due to the fact that Mobiflage currently only supports one hidden offset.
4) Relying on FAT32 formatted external storage may not be a valid method of implementing PDE for much longer. Although many manufacturers still produce devices with removable storage (e.g., Samsung Galaxy S4), the trend seems to be moving toward MTP enabled devices without SD cards. Several other platforms (e.g., Apple iOS) also lack removable storage.

5 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. Mobile devices are also often connected to a cellphone network. We discuss possible collusion attacks with the help of wireless carriers in the NDSS 2013 publication [1].

5.1 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 [21]). However, certain statistical deviations between cipher and PRNG output may exist 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., [8]). 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 [22], [23] to prevent exploitation of known weaknesses in CBC when used for disk encryption. We refer the reader to the NDSS 2013 publication [1] for further information.

5.2 Leakage from Flash-storage

In this section, we provide an overview of flash storage 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 multimedia card (eMMC) for internal storage and some use secure digital (SD) removable 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.

The software FTL used by the Linux MTD driver (`mtdblock`) is simplistic and does not use a wear-leveling mechanism [24]. 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 [25] and eMMC [26] 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.

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. For example, it is possible that an adversary with access to the raw flash could distinguish between random data and encrypted data. Since a flash page is generally larger than one sector, small modifications to a file (i.e., changes within one sector) will result in copying unmodified sectors to a new page, along with the modified sector. Hence identical ciphertexts will exist in separate physical locations on the storage. The probability of identical outputs from an RNG should occur with a much higher period (e.g., once every 2^{64} 128-bit blocks [27, pp. 137–161]). 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. Some log-structured disk encryption schemes (e.g., WhisperYAFFS [28]) address this issue. However, such file systems are not suitable for the block device nature of eMMC. We offer two possible mitigations for block structured file systems: **(a)** increase the XTS data unit size in dm-crypt to that of a flash page. Although most XTS implementations use 512 bytes as the data unit size (as this is the size of a traditional disk sector) the IEEE P1619 [22] standard allows any size. Since XTS has a cascade effect, changing any bit in the flash page will result in a completely different ciphertext. **(b)** Alter the Ext4 file system driver to ensure that any modification to a file involves a full copy of all file data to new blocks. The new blocks will map to different disk sectors, which will result in different ciphertexts under XTS. This option is less favorable, as it may result in increased IO and data fragmentation.

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. [29] 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 unmapped 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[2]) cannot mount this attack. Physical acquisition mechanisms exist for MTD storage (see e.g., Hoog [30, pp. 266–284]); however, they tend to be costly, time consuming, and generally destroy the 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. [29] 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. [31] 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 [31].

2. https://viaforensics.com

5.3 Leakage from Apps and the OS

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., [7]). 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 6 (item (b)) that log files, swap space, and temporary storage are effectively isolated between the two modes of Mobiflage.

6 Security Analysis

In this section, we evaluate Mobiflage against known attacks and weaknesses. We refer the reader to the NDSS 2013 publication [1] for other attacks/concerns.

(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 time-out will slow an online guessing attack, but it may still be feasible on a weak password.

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., [32]) 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: our Nexus S (1GHz Exynos-3 Cortex-A8) development phone, it required an additional 0.67 ± 0.01 seconds to calculate a single key, while the guessing attack becomes 100 times slower on our PC.

(b) 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. [7]. As Mobiflage isolates the outer and hidden volumes, 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`. To prevent leakage through these logs, we mount a RAM disk to this mount point, when booting into the PDE mode. Logs will persist between standard mode boots, but PDE mode logs are not 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.

(c) 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.

7 Performance Evaluation

This section summarizes several tests on our prototype implementations, to understand the performance impact on the regular use of a device.

Cipher performance. To determine the performance impact resulting from the cipher, 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. Although these results were obtained using MF-SD, the impact of the cipher should affect MF-MTP in a similar way.

Mobiflage seems to decrease throughput by roughly 5% over Android FDE. The full results are discussed in the NDSS 2013 publication [1]. The observed decrease in throughput may be attributed to the chosen cipher: XTS-512 requires two AES-256 operations per block; and AES-256 uses fourteen rounds of operations vs. ten rounds in AES-128 for Android FDE.

The required time to encrypt the device is increased on account of the two pass random wipe. MF-MTP will take twice as long as the default Android FDE to encrypt internal partitions. The exact time will depend on the size and type of the storage.

Modified file system performance. The performance of MF-MTP will be affected by both the cipher and the modified file system behavior. Due to the FTL, there is already a good deal of redirection and seeking involved in flash data access, even when blocks are logically sequential from the file system's point-of-view. However, any increase in seeking is not as cumbersome for flash storage as for magnetic storage, since there is not any time-consuming physical rotation or access-arm adjustment in flash storage (i.e., seeking any block in flash storage is performed in constant $O(1)$ time complexity). Note that at this time, we have not tested the linear Ext4 allocator on magnetic storage, which will likely be affected more adversely than flash.

To determine the performance impact of our modified Ext4 driver, on flash storage, we use the file system benchmarking tool Bonnie++ [33]. We compared the default Ext4 against our modified version on our Samsung Nexus. We used the Nexus S to avoid large-file issues experienced on the Nexus 4 (Bonnie++ requires tests performed on a file that is twice the size of the system RAM). We observed that throughput was not significantly affected, however latency for certain operations increased and decreased for others; see Tables 1 through 3. The improved latency (e.g., sequential read) may be due to data locality: since flash operates on segments that are larger than file system blocks, the total number of requests may be reduced when the blocks are stored sequentially (i.e., only one erase-block must be requested to access several sequential file system blocks). The increased latency (e.g., sequential create) may also be caused by data locality: since only a certain number of erase-blocks may be open for modification at a given time (i.e., modifying data in different erase-blocks may be performed in parallel, while modifying data in a single erase-block must be done successively).

	Throughput (KB/s)	
	Seq read	Seq write
Ext4 default	9012±163	22501±439
Ext4 linear	8906±97	22580±553
% deviation	-1.18	0.35

TABLE 1: Throughput of a 925MB file, at 4096-byte granularity, in KB/s, of default Ext4 and Ext4 with linear inode allocation; averaged across ten trials measured with Bonnie++ (Seq: sequential).

8 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 a discussion on academic deniable-storage proposals, we refer the reader to the NDSS 2013 publication [1].

All major desktop OSes now offer storage encryption with FDE support. 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. To contend with a coercive adversary, PDE adds another layer of secrecy over FDE.

Most mobile OSes also offer storage encryption (without PDE). BlackBerry devices use a password derived key to encrypt an internal storage AES key [34]. Removable storage data 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 [35]. 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 (beyond the default UID-key wrapping, which only protects the data if the storage is removed and attacked without the device's crypto processor) [36], [35]. 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). This is not possible with the current Android architecture, since background read/write operations would fail. Older Android 2.3 (Gingerbread) devices can make use of third party software (e.g., WhisperCore [37]) 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 [6] and FreeOTFE [38] 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

	file system operations/s				
	Rnd seek	Seq create	Seq delete	Rnd create	Rnd delete
Ext4 default	545±13	858±15	8475±1383	903±21	1303±99
Ext4 linear	540±21	796±17	9489±848	866±21	1200±82
% deviation	-1.00	-7.23	11.96	-4.10	-7.87

TABLE 2: Observed operations per second of default Ext4 and Ext4 with linear inode allocation; averaged across ten trials measured with Bonnie++ (Seq: sequential, Rnd: random).

	Operational latency (ms)						
	Seq read	Seq write	Rnd seek	Seq create	Seq delete	Rnd create	Rnd delete
Ext4 default	3868±450	217±85	597±122	332±75	502±161	384±140	948±485
Ext4 linear	3995±526	182±71	614±129	516±239	460±126	320±135	1474±512
% deviation	3.28	-16.33	2.72	55.46	-8.30	-16.52	55.43

TABLE 3: Operational latency, in milliseconds, of default Ext4 and Ext4 with linear inode allocation; averaged across ten trials measured with Bonnie++ (Seq: sequential, Rnd: random).

systems, similar functionality can be achieved using an early user-space RAM disk (i.e., a temporary root file system). TrueCrypt does not perform this configuration, and requires the user to set up a RAM disk with the TrueCrypt binary to capture the password and unlock the disk before the kernel attempts to mount the actual root file system.

System encryption with pre-boot authentication 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. [39]). The dm-crypt volume could then be mounted before loading the Android framework. However, since the OS partition is read-only on Android devices, it is not encrypted. So we choose 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 the OS/applications leaking knowledge of hidden data (e.g., into logs, swap space, or temporary files) 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. 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 to prevent leakage 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.

There is a recent effort to port TrueCrypt to Android [40]. 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).

9 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. [7]). 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.

References

[1] A. Skillen and M. Mannan, "On implementing deniable storage encryption for mobile devices," in *Network and Distributed System Security Symposium (NDSS 2013)*, San Diego, CA, USA, Feb. 2013.

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

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

[4] J. Assange, R.-P. Weinmann, and S. Dreyfus, "Rubberhose: cryptographically deniable transparent disk encryption system," 1997, project website: http://marutukku.org/.

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

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

[7] 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.

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

[9] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Communications of the ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[10] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, Mar. 1983.

[11] 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.

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

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

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

[15] kernel.org, "Ext4 disk layout," 2012, online document (July 2012). https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

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

[17] B. Kaliski, "PKCS #5: password-based cryptography specification, ver2.0," Sep. 2000, RFC 2898 (informational).

[18] 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.

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

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

[21] P. Rogaway, "Nonce-based symmetric encryption," in *Workshop on Fast Software Encryption (FSE'04)*, ser. LNCS, vol. 3017, Delhi, India, 2004, pp. 348–358.

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

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

[24] infradead.org, "The MTD subsystem for Linux," 2012, online document (July 2012). http://www.linux-mtd.infradead.org/faq/general.html.

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

[26] JEDEC, "eMMC card product std ver4.41 (JESD84-A441)," 2010, technical specification (Mar. 2010). http://www.jedec.org/sites/default/files/docs/JESD84-A441.pdf.

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

[28] WhisperSystems, "WhisperYAFFS: Encrypted filesystem support for YAFFS2," 2011, https://github.com/WhisperSystems/WhisperYAFFS.

[29] 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.

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

[31] 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.

[32] B. Wallace, "Transferable state attack on iterated hashing functions," 2012, tech. document (July 2012). http://firebwall.com.

[33] Russell Coker, "Bonnie++ file system benchmark suite," 2009, version 1.96 (Jul. 5, 2009). http://www.coker.com.au/bonnie++/.

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

[35] Apple, "iOS security," 2012, technical document (May 2012). http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf.

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

[37] WhisperSystems, "WhisperCore: device and data protection for Android," 2012, beta version (0.5.5). http://whispersys.com/whispercore.html.

[38] FreeOTFE, "FreeOTFE - free disk encryption software for PCs and PDAs," 2012, version 5.21 (Nov. 2012). http://www.freeotfe.org/.

[39] TeamWin, "TeamWin recovery project (TWRP)," 2012, version 2.3.1 http://teamw.in/project/twrp2.

[40] Cryptonite, "EncFS and TrueCrypt on Android," 2012, open-source project (2012). https://code.google.com/p/cryptonite/.

Adam Skillen received his Masters degree in Information Systems Security in 2013 at Concordia University in Canada. In 2013 Adam began PhD research at Carleton University looking into privacy and security of mobile devices. Adam's main interests are storage encryption, authentication, and access control mechanisms.

Mohammad Mannan received the PhD degree in Computer Science from Carleton University in 2009 in the area of Internet authentication and usable security. He is an assistant professor at the Concordia Institute for Information Systems Engineering, Concordia University, Montreal. His research interests lie in the area of Internet and systems security, with a focus on solving high-impact security and privacy problems of todays Internet.