Deniable Storage Encryption for Mobile Devices

A Thesis in the Department of Engineering and Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Applied Science in Information Systems Security
at the Concordia Institute for Information Systems Engineering
Concordia University, Montreal, Canada

Adam Skillen
Supervisor: Dr. Mohammad Mannan

April 3, 2013

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By:	**Adam Skillen**
Entitled:	**Deniable Storage Encryption for Mobile Devices**

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science in Information Systems Security

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Rachida Dssouli, Ph.D.	Chair
Otmane Ait Mohamed, Ph.D., P.Eng.	Examiner
Benjamin C. M. Fung, Ph.D., P.Eng.	Examiner
Mohammad Mannan, Ph.D.	Supervisor

Approved ______________________________
Chair of Department or Graduate Program Director

__________ 2013 ______________________________
Robin Drew, Ph.D., Dean
Faculty of Engineering and Computer Science

Abstract

Deniable Storage Encryption for Mobile Devices

Adam Skillen

Smartphones, and other mobile computing devices, are being widely adopted globally as the de-facto personal computing platform. Given the amount of sensitive information accumulated by these devices, there are serious privacy and security implications for both personal use and enterprise deployment.

Confidentiality of data-at-rest can be effectively preserved through storage encryption. All major mobile OSes now incorporate some form of storage 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. This dissertation explores the feasibility and efficacy of deniable storage encryption for mobile devices. A feature that allows the user to feign compliance with a coercive adversary, by decrypting plausible and innocuous decoy data, while maintaining the secrecy of their sensitive or contentious hidden data. A deniable storage encryption system, *Mobiflage*, was designed and implemented for the Android OS, the first such application for mobile devices.

Current mobile encryption mechanisms all rely, in some way, on a user secret. Users notoriously choose weak passwords that are easily guessed/cracked. This thesis offers a new password scheme for use with storage encryption. The goal is to create passwords that are suitably strong for protection of encryption keys, easier to input on mobile devices, and build on memorability research in cognitive psychology for a better user experience than current password guidelines.

Contents

List of Figures

List of Tables

List of Equations

List of Acronyms

AES	advanced encryption standard
ASIC	application-specific integrated circuit
CBC	cipher-block chaining
CD	connected discourse
CPA	chosen-plaintext attack
ECB	electronic codebook
ECC	elliptic curve cryptography
eMMC	embedded multi-media card
ESSIV	encrypted salt-sector initialization vector
FDE	full disk encryption
fMRI	functional magnetic resonance imaging
FPGA	field-programmable gate array
FTL	flash translation layer
HMAC	hash-based message authentication code
ICS	Ice Cream Sandwich
IMEI	international mobile equipment identity
IV	initialization vector
JB	Jelly Bean

JTAG	Joint Test Action Group
LBA	logical block address
LUKS	Linux unified key setup
LVM	logical volume manager
MTD	memory technology device
MTP	media transfer protocol
NLP	natural language processing
OTA	over-the-air
OTFE	on-the-fly encryption
PBKDF2	password based key derivation function 2
PDE	plausible deniable encryption
POS	part-of-speech
PRNG	pseudorandom number generator
RS	random sequence
SD	secure digital
SHA	secure hash algorithm
SIM	subscriber identity module
UDS	usability-deployability-security framework
UID	unique identification number
XEX	XOR-encrypt-XOR
XTS	XEX tweaked-codebook with ciphertext stealing

Chapter 1

Introduction

1.1 Motivation

Smartphones and tablets are now powerful computing devices, affording most of the capabilities found on laptop and desktop computers. The smartphone market share is rapidly expanding, and millions of new devices are sold every month.[1] Many users perform the majority of their communication, web browsing, and financial tasks on their mobile devices, creating serious privacy and security concerns. Corporate devices may accumulate confidential or proprietary documents; while personal devices have passwords, browsing history, and private communications.

To protect the user, in the event a device is lost or stolen, all major mobile OSes now provide some form of encryption to protect the data stored on these devices. In certain situations however, data confidentiality achieved through encryption is inadequate, to properly protect a user's sensitive data. A user may be coerced into disclosing their decryption keys. In this case, the sensitive data must be hidden so that its very existence can be denied. Steganographic techniques and deniable encryption algorithms have been explored to address this specific problem.

Given their portability and media capture capabilities, we assert that plausible deniable encryption (PDE) is a significant security requirement for mobile devices.

[1]http://www.gartner.com/it/page.jsp?id=2237315

The added protection offered by deniable storage encryption may be necessary when the consequences for possession of contentious material can result in incarceration, grievous bodily harm, or even death. Examples might include dissemination of censored news, literature of dissent, or evidence of atrocities.

Using PDE in these situations would not directly aid an individual in smuggling controversial data: the storage device could be confiscated, erased, or destroyed. However, PDE can protect the user, since the data may not be discovered. The user can even provide decoy passwords and data to simulate accordance with a coercive adversary's orders to decrypt the data.

All of the existing storage encryption mechanisms (including PDE) rely, to some extent, on a user secret, to protect the encryption keys. Users tend to choose poor device passwords, such as short numeric PIN codes, reminiscent of bank-card secrets [78]. On mobile devices, the user must also contend with arduous input mechanisms. Even simple passwords take almost twice as long to type on smartphone soft-keyboards, as compared with standard keyboards [47]. When given the choice, many users will pick poor passwords that are easy to remember and input, but also easy to guess/crack.

Relying on existing user-chosen secrets to protect encryption keys reduces the overall security of the cipher to the complexity of breaching the password. Password-stretching and key derivation functions (e.g., [58, 51, 71]) have been devised to mitigate this risk, but in many cases are not implemented in such a way as to offer a significant security advantage. In the case of PDE, a weak password can lead to the exposure of the hidden data, if the password is guessed. Disclosure of deniably-encrypted data may result in more sever consequences than exposure of a user's personal/private data.

1.2 Thesis Statement

The primary aim of this dissertation is to design and implement a suitable security mechanism for mobile devices, to contend with a coercive adversary. The additional objective of devising an appropriate and convenient authentication scheme for protecting encryption keys is also addressed.

The main research questions explored in this dissertation are:

- ***Question 1:*** From both feasibility and efficacy standpoints, is deniable storage encryption a practical goal for mobile devices?
- ***Question 2:*** Are there uncontrollable factors, such as communication channels and filesystem/storage technologies, that will betray or compromise the deniability of hidden data, despite a theoretically sound design?
- ***Question 3:*** Given the restrictive input mechanisms available on mobile devices, can a strong and memorable password scheme be devised, suitable for protecting mobile storage-encryption keys?

1.3 Contributions

Deniable storage encryption mechanisms exist for PCs (e.g., [97, 62]). However, at the time of writing this dissertation, no such system is available for mobile devices. The tight-coupling between mobile device hardware and software, along with the boot procedure, and distinctive storage features do not lend themselves to simply porting existing PC PDE schemes. Mobile devices also open up new existence-leakage vectors, and sources of compromise that can betray deniability. To assess the feasibility and efficacy of PDE for mobile devices, and address the aforementioned obstacles, the *Mobiflage* scheme was designed and implemented for the Android OS. We analyze the performance of the Mobiflage prototype on two mobile devices. We also explore the 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.

Symmetric encryption keys are generally chosen at random, and may be as long as 512-bits. It is unrealistic to expect a user to remember 64 random bytes, so random keys are often protected by encrypting them with a password-derived key. This reliance on user-chosen secrets to protect encryption keys reduces the overall security of the cipher to the complexity of the password. This problem is compounded, as users tend to choose poor passwords, such as short numeric PIN codes, on mobile devices, to contend with constrained input mechanisms [78]. For example, we were able to test all encryption keys derived from 4 and 5 digit PIN codes (110,000 in total) for Android in under ten minutes on commodity hardware (see Appendix B). This dissertation discusses a new password scheme for use with storage encryption. The goal of the *Myphrase* design is to facilitate passwords that are suitably strong for protection of encryption keys, easier to input on mobile devices, and alleviate the memory burden on the user. By building on research in cognitive psychology, Myphrase passwords are constructed to offer better memorability and security: familiar words and structures are randomly combined to create user-specific multi-word passphrases. The Myphrase system was designed and implemented for use with both PCs and mobile devices. We analyze the expected entropy of Myphrase passphrases with two datasets: the Enron email corpus, and the collected works of several popular authors from Project Gutenberg. We also evaluate Myphrase using a recently proposed framework of usability-deployability-security ratings and compare it against similar designs.

1.4 Related Publication

Conference Paper. The work on deniable storage encryption discussed in this dissertation has been peer-reviewed and published in the following article:

> On Implementing Deniable Storage Encryption for Mobile Devices. A. Skillen and M. Mannan. *Network and Distributed System Security Symposium (NDSS 2013)*, Feb. 24-27, 2013, San Diego, CA, USA.

Technical Report. The work on strong and memorable passwords is available as a technical report:

> Myphrase: Passwords from Your Own Words. A. Skillen and M. Mannan. *Technical Report: (http://spectrum.library.concordia.ca/976791/)*, Jan. 24, 2013.

1.5 Outline

The rest of this dissertation is organized as follows. In Chapter 2 we provide background information on storage encryption techniques, deniable encryption, and existing password-based encryption key protection methods. Chapter 3 presents the design and implementation details for our mobile-device deniable-storage mechanism. We present the threat model, evaluate the unique challenges present in the mobile environment, and assess the security and performance of the implementation. A novel password-compatible key-protection mechanism, along with implementation details, evaluation, and comparison against similar techniques, is introduced in Chapter 4. In Chapter 5, we summarize the findings, in the context of the thesis objectives, and provide some concluding remarks and future research directions.

Chapter 2

Background

This chapter covers some of the necessary background information and literature related to the work discussed in this dissertation. Specific literature reviews and background information pertinent to each project is contained within the relevant chapters.

2.1 Full Disk Encryption

Full disk encryption (FDE) employs symmetric-key block 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 and applications in what is sometimes referred to as on-the-fly encryption (OTFE). OTFE implies that encryption is performed as files are written to the volume, and decryption is performed during file access. In this manner, files only exist in a decrypted state while they are in RAM. All data stored on the physical drive is encrypted, even while it is actively mounted onto the filesystem. 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). All major desktop OSes now offer storage encryption with FDE support (e.g., Windows BitLocker, Mac OS X FileVault, and the Linux unified key setup (LUKS)).

The traditional encryption threat model (e.g., Kerckhoffs's Desiderata [53]) discusses encryption in the context of network communications. Encryption of storage-at-rest must be treated differently, for example, since keys and IVs must be stored locally (or else easily derived from local material). Certain crypto primitives, that are entirely suitable for network communications, may be exploitable in the context of FDE. Likewise, a vetted FDE implementation may not be appropriate for PDE; see Appendix A for some examples.

Mobile devices may contain personal information, such as contacts and passwords. Being small and portable, these devices can be easily lost or stolen. To mitigate the consequences of such a physical security compromise, all major mobile OSes now provide some form of encryption to protect the data stored on these devices. Some use per-file encryption, while others use FDE to encrypt entire storage devices or volumes. Table 2.1 provides a comparison of existing mobile encryption implementations.

2.2 Plausible Deniable Encryption

Semantic security is a property of an encryption function which states that possession of a ciphertext alone will provide no information about the plaintext message (without knowledge of the key) [34]. PDE defines a level of protection beyond semantic security, in which a coercive adversary can demand the plaintext message or decryption key. The premise of a PDE cipher, is that two or more distinct messages will result in the same ciphertext, when encrypted with different random inputs (e.g., different keys or nonces).

Mobile OS	Cipher Spec	Key Length	Type	SD Card Encrypted	Key Storage
Android	AES-CBC ESSIV:SHA256	128-bit	FDE	No	PBKDF2[a] (2000): Volume Footer
BlackBerry	AES-CBC Random IVs	256-bit	File	Yes	PBKDF2 (20000): Internal Keystore
iOS	AES-CBC Random IVs	256-bit	Filesystem + File	N/A	PBKDF2 (10000): Internal Keystore
Symbian	AES-XTS Plain IVs	256-bit[b]	File	Yes	PBKDF2 Internal Keystore
Windows Phone	AES-CBC Modified ESSIV	128-bit	File System	No	PBKDF2 FS Header

[a]PKCS#5 [51]
[b]128-bit AES + 128-bit XEX tweak

Table 2.1: *Existing storage encryption mechanisms employed by major mobile OSes*

Deniable encryption ciphers are classified in three ways [15]: **(a)** by which parties are being coerced (sender, receiver, or bi-deniable); **(b)** by the underlying encryption schemes (symmetric or public-key); and **(c)** by the time at which the decoy messages can be created—either at the time of coercion (ad-hoc), or at the time of encryption (plan-ahead).

For instance, a simplification of one of Canetti's original examples describes a public-key PDE scheme [15]: Bob wants to encrypt a sensitive message for Alice. Along with the intended message, he also creates a decoy message that he can provide to a coercive adversary without penalty. He also generates a pair of random looking nonces that, when encrypted with each message, will create identical ciphertexts under Alice's public key. When faced with a coercive eavesdropper, Bob can forfeit the *decoy* message and nonce. Since neither Bob nor the adversary have access to Alice's private key, they can only re-encrypt the message with her public key, and observe that the resulting ciphertext is identical to what had been captured earlier. This example demonstrates a sender-deniable, plan-ahead, public-key PDE scheme.

Several new ciphers, or enhancements to existing ciphers, have been proposed to create PDE schemes (e.g., [15, 69, 27, 56]). However, most of these proposals strive to enable PDE in network communications, and are not directly applicable to storage encryption.

Some PDE schemes use flexible deniability, in which ciphers have separate *honest* and *dishonest* encryption algorithm variants [15]. These ciphers tend to have a non-negligible detection probability (i.e., the adversary can detect whether the honest or dishonest algorithm was used during encryption). Other schemes use single-algorithm ciphers that would only be chosen by a user to enable PDE (i.e., using such a cipher reveals the desire to enable PDE, and would not normally be used in any other situation).

The *Mobiflage* scheme described below enables deniability through existing symmetric-key block ciphers, as opposed to specially designed PDE ciphers. Mobiflage is, strictly speaking, a steganographic (data hiding) technique, although it achieves the same behaviour as a PDE cipher. In terms of the PDE paradigm, Mobiflage is receiver-deniable, in that the user is expected to possess the key to decrypt the message (i.e., volume), and may instead decrypt a decoy message (i.e., decoy volume). Mobiflage is a plan-ahead construction, in that the user must decide on a fixed number of decoy messages (i.e., decoy volumes) at encryption time. See Section 3.9 for existing deniable storage proposals.

2.3 Password Protected Encryption Keys

The problem of using a low-entropy secret, such as a password, to protect a high-entropy encryption key gave rise to key-derivation and password-stretching algorithms [52]. The general idea is to use computationally expensive functions to add a known amount of complexity to a low-entropy secret, hence slowing an exhaustive search of the keyspace. The most widely used such algorithm is possibly the PKCS#5 standard: password based key derivation function 2 (PBKDF2) [51].

In the original PBKDF2 proposal, Kaliski suggests 1000 iterations of the HMAC-SHA1 hash algorithm [51]. At the time, in the year 2000, it was believed that 1000 iterations would require one second to derive a single key on commodity hardware. The negligible delay for a single password would compound to significantly slow a brute-force attack. Since the time of publication, computer performance has grown exponentially. The implication of Moore's law [66] (which has, for the most part, held true) is that computing power has roughly doubled every two years. So the original 1000 iterations should now, thirteen years later, require closer to 100,000 iterations to achieve the same one second delay. Unfortunately, most implementations use relatively low iteration counts (e.g., Google Android uses 2000 and Apple iOS uses 10,000). Additionally, custom single-purpose hardware (e.g., ASICs and FPGAs) can be designed to perform the computations much faster than commodity hardware.

Other password-stretching algorithms have been proposed (e.g., [58, 71]). The scrypt [71] proposal relies on both computationally complex and memory expensive functions, to mitigate the advantage gained by custom hardware. NIST approved an AES key-wrapping function [83] for the protection of random encryption keys. The drawback is that the key-encryption-key (KEK) must be a valid AES key-length, and is therefore likely to be used in combination with a passphrase stretching function.

In addition to the problems with iteration count and custom hardware, the user's choice of password can also reduce the necessary search space. When given the choice, users will pick poor passwords that are easy to remember and input, but also easy to guess/crack. Users rarely choose random passwords, and instead choose sequences that are easy to remember and manage. Many users even choose dictionary words, or their derivatives. New probabilistic password guessing algorithms (e.g., [106]) have significantly reduced the time to attack user-chosen passwords.

All of the existing mobile storage encryption mechanisms rely, to some extent, on a user secret, to protect the encryption keys. The overall security of the crypto-system is reduced to the complexity of the weakest component. Given the poor decisions made in key-derivation implementations, as well as advances in password guessing, it is unwise to rely on user-chosen secrets to protect encryption keys.

Chapter 3

Mobiflage: Deniable Storage Encryption for Mobile Devices

This chapter discusses the design, implementation, and evaluation of a deniable storage encryption scheme suitable for mobile devices. Existing, and newly discovered, challenges that can compromise PDE in a mobile environment are explored. To address these obstacles, a system called *Mobiflage* that enables PDE on mobile devices was designed and implemented for the Android OS. By leveraging lessons learned from known issues in deniable encryption for the desktop environment, new countermeasures for threats specific to mobile systems are incorporated into the design. To restrict attacks that may compromise deniability, strict user compliance is necessary, including the use of a strong password to protect the PDE key (see our design in Chapter 4 for an example scheme).

3.1 Introduction and Motivation

Smartphones and other mobile computing devices are being widely adopted globally. For instance, according to a comScore report [18], 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 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. Plausible deniable encryption (PDE) was first explored by Canetti et al. [15] 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 [96], 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 [97]. 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 boot-loader is only designed to chain-load 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., [21]). 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 favourable 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.

The academic contributions of this research 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).

3.2 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 [97]); 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 [97], 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 as the Galaxy Nexus, have neither physical nor emulated external storage. Instead, they use the

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

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 3.6.3.

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 [32]). 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. [21, 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 [67, 24], 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., [22]).
8. We assume the mobile OS, kernel, and boot-loader 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 [103], 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 3.5. Following these guidelines may require non-trivial effort, but is required for maintaining deniability in our threat model.
10. The storage medium is assumed to be divided, either physically or logically, into 512-Byte sectors, for use with the FDE system. All calculations below (e.g., Equation 3.3) are performed modulo 512-Bytes.

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., [93, 94]). 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 [57].

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.

3.3 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).

3.3.1 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 3.6.3.

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.

3.3.2 Steganographic File-systems vs. Hidden Volumes

There are currently two main types of PDE systems for use with FDE: steganographic file systems (e.g., StegFS [1, 62]) and hidden volumes (e.g., TrueCrypt [97] and FreeOTFE [31]). 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 Section 3.9.2.) 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.

3.3.3 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:

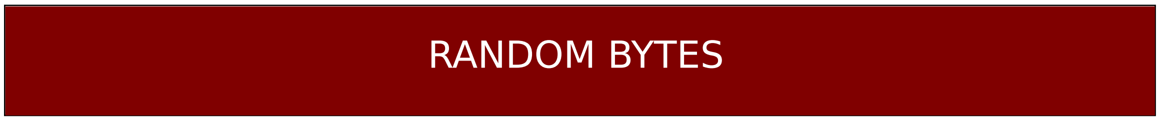
$$E_{K1}(Vol_1)||E_{K2}(Vol_2)||...||E_{Kn}(Vol_n) \tag{3.1}$$

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

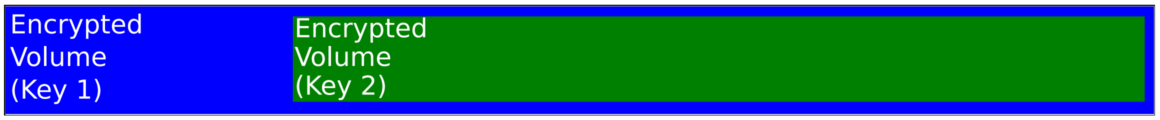
When the disk is decrypted with a given key, the other volumes will appear to be uniformly random data. When the user is coerced, she can provide the outer volume key and claim that no other volumes exist:

$$D_{K1}(Vol_1||Vol_2||...||Vol_n) = Vol_1||Rand \tag{3.2}$$

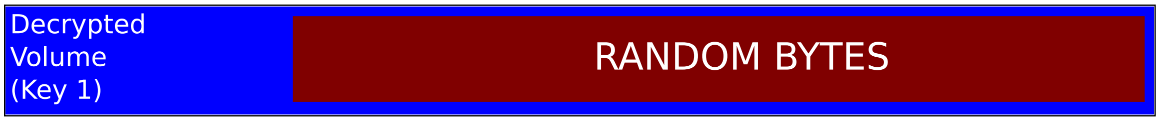
Here, $D_K(\cdot)$ represents the symmetric decryption function with key K (corresponding to $E_K(\cdot)$) and *Rand* represents data that cannot be distinguished from random bits. Therefore, a forensic analysis of the decrypted outer volume will not indicate the existence of hidden volumes. However, some statistical deviations may be used to distinguish the random data from the cipher output; see Section 3.6.1. Also, the adversary may not trust the user to have disclosed all volume keys and continue to



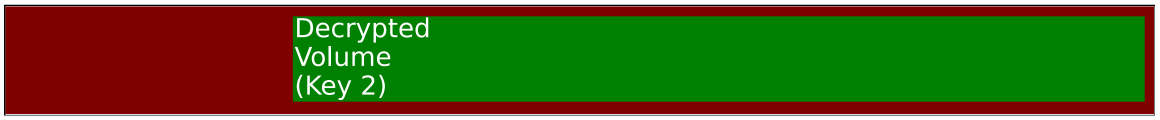
(a) *The encrypted disk will appear as uniformly random bytes*



(b) *Encrypted volumes are created at different offsets using different keys; each volume is formatted to consume all remaining space*



(c) *When the outer volume is decrypted, it appears to consume the entire disk and the inner volume data is indistinguishable from random bytes*



(d) *When an inner volume is decrypted, all other volumes are hidden among the random noise*

Figure 3.1: *Hidden volume storage layout*

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 3.7, 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. Figure 3.1 depicts a graphical representation of the storage layout.

3.3.4 Offset Calculation

The offset to a hidden volume is generated as follows (calculations are performed modulo 512-Bytes):

$$offset = \lfloor 0.75 \times vlen \rfloor - \Big(\mathrm{H}(pwd||salt) \bmod \lfloor 0.25 \times vlen \rfloor \Big) \tag{3.3}$$

Here, H is a PBKDF2 iterated hash function [51], *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 (Equation 3.3) 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 [32, 33]) 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 3.4.4, item (2).

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 Equation 3.3: 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 couple of kilobytes of data captured from that region (only the key and filesystem 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 3.7, 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).

3.3.5 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 the settings GUI). Mobiflage's initialization phase erases existing data on the external storage (SD

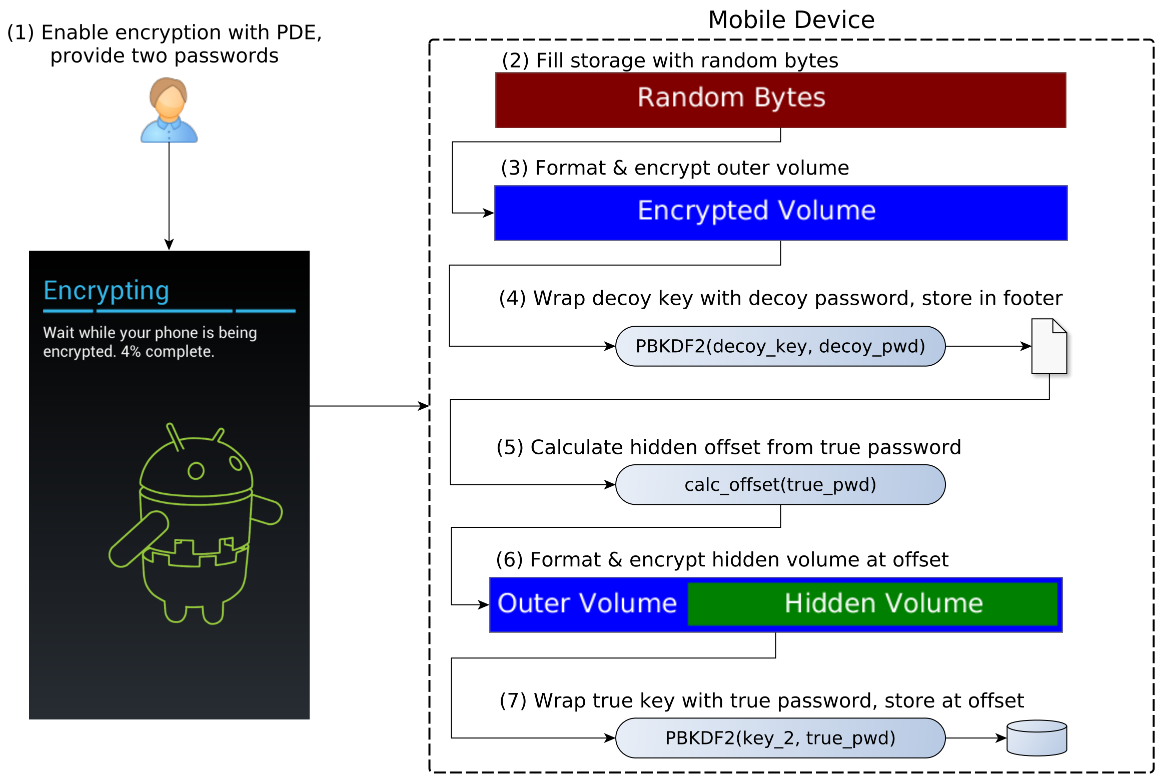

(1) Enable encryption with PDE, provide two passwords
Encrypting
Wait while your phone is being encrypted. 4% complete.
Mobile Device
(2) Fill storage with random bytes
Random Bytes
(3) Format & encrypt outer volume
Encrypted Volume
(4) Wrap decoy key with decoy password, store in footer
PBKDF2(decoy_key, decoy_pwd)
(5) Calculate hidden offset from true password
calc_offset(true_pwd)
(6) Format & encrypt hidden volume at offset
Outer Volume
Hidden Volume
(7) Wrap true key with true password, store at offset
PBKDF2(key_2, true_pwd)

Figure 3.2: *Mobiflage initialization process; steps 1, 3, and 4 are performed by the default Android FDE, the others are unique to Mobiflage*

card); this data should be backed-up before initiating Mobiflage. However, users can choose to preserve the outer volume's userdata 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; see Figure 3.2. Unlike Android FDE, Mobiflage must initialize the external storage with random data for deniability. This

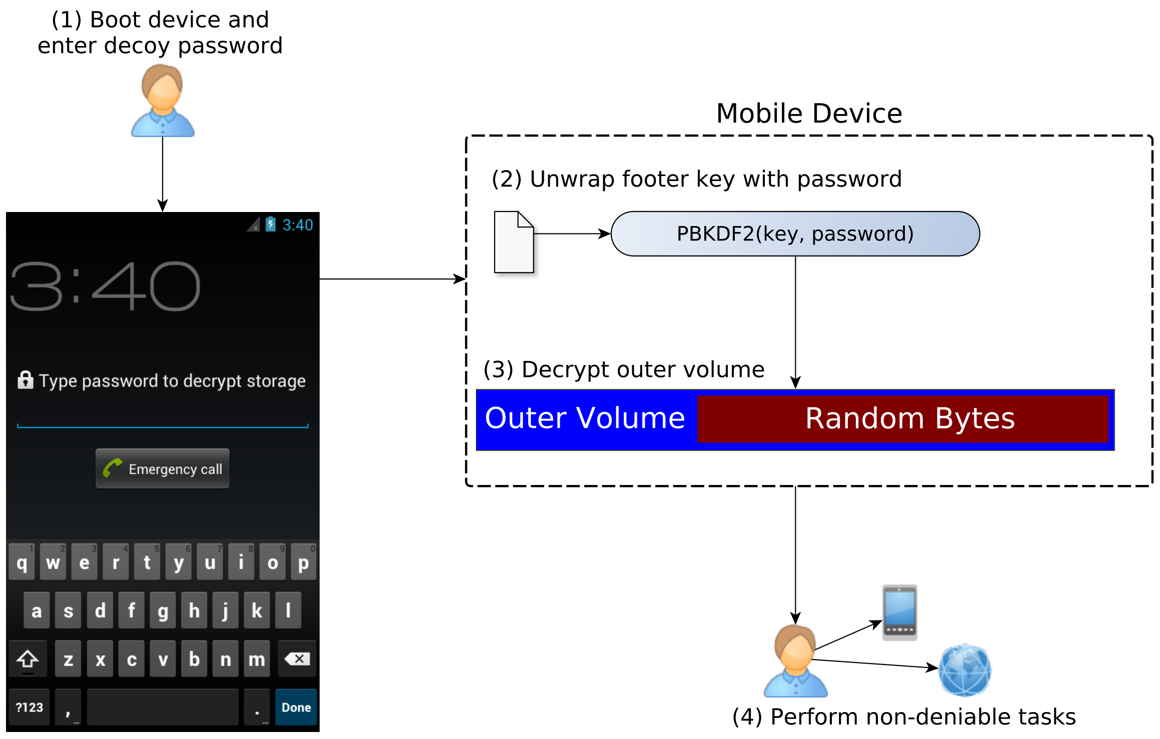

(1) Boot device and enter decoy password
Mobile Device
(2) Unwrap footer key with password
PBKDF2(key, password)
3:40
Type password to decrypt storage
Emergency call
(3) Decrypt outer volume
Outer Volume
Random Bytes
(4) Perform non-deniable tasks

Figure 3.3: *Mobiflage usage – standard mode; for day-to-day use of the device (i.e., no plausible deniability)*

makes Mobiflage slower than the default Android FDE initialization (see Section 3.8). 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 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. Figure 3.3 illustrates the standard mode boot process.

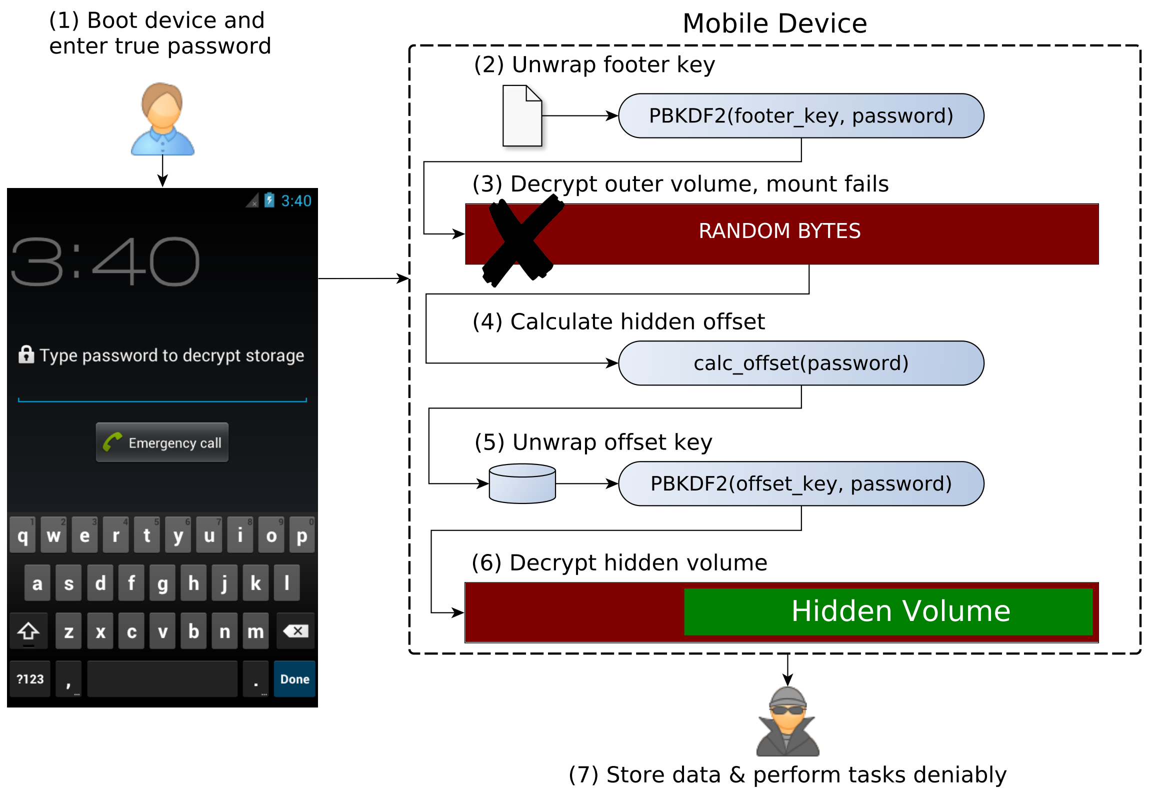

(1) Boot device and enter true password
Mobile Device
3:40
3:40
Type password to decrypt storage
Emergency call
Done
(2) Unwrap footer key
PBKDF2(footer_key, password)
(3) Decrypt outer volume, mount fails
RANDOM BYTES
(4) Calculate hidden offset
calc_offset(password)
(5) Unwrap offset key
PBKDF2(offset_key, password)
(6) Decrypt hidden volume
Hidden Volume
(7) Store data & perform tasks deniably

Figure 3.4: *Mobiflage usage – PDE mode; in this mode, the user can store data or perform tasks which can later be denied*

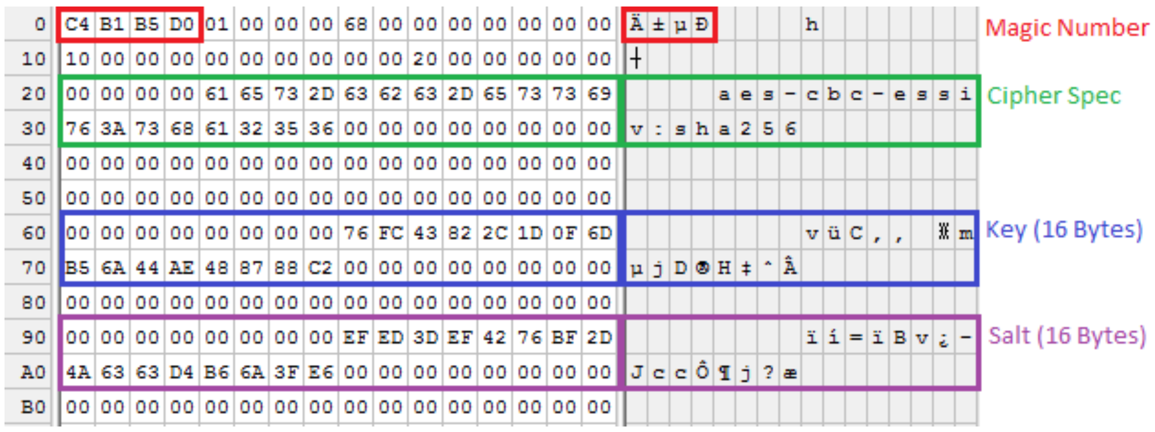
When the user requires the added protection of deniable storage, they will reboot their device and provide their deniable password when prompted. In the PDE mode, they can transfer documents from another device, or take photos and videos. Note that app/system logs in this mode are hidden or discarded; however, there is still a possibility of leakage through network interfaces. Section 3.5 provides a list of precautions the user should take to mitigate such risks. Figure 3.4 illustrates the PDE mode boot process.

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 [36]).

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.

3.4 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 3.5: *Android FDE footer; note that some fields (e.g., the cipher specification) is stored in clear text)*

3.4.1 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 [23]. 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 [32]; see also Section 3.6.1). A randomly chosen master volume key is encrypted with the same cipher by a key derived from 2000 iterations of the PBKDF2 [51] 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 3.5 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 3.6.1 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

Volume	Mount point	Mode	Description
Boot	N/A	N/A	Boot-loader 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 3.1: *Typical volumes on common Android devices (RO: read-only; RW: read-write; N/A: not applicable)*

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 3.6.2 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.

3.4.2 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 3.1 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

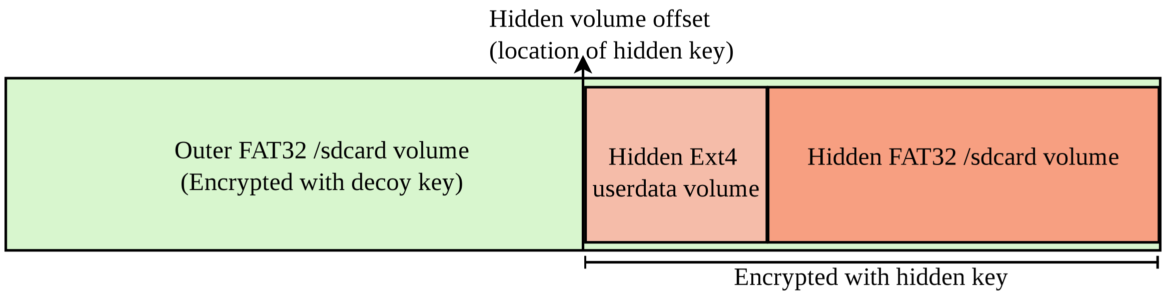

Hidden volume offset
(location of hidden key)
Outer FAT32 /sdcard volume
(Encrypted with decoy key)
Hidden Ext4
userdata volume
Hidden FAT32 /sdcard volume
Encrypted with hidden key

Figure 3.6: *Mobiflage SD card PDE layout*

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 the internal userdata partition is Ext4. For reasons outlined in Section 3.6.3, 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 3.6. 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 3.7, item (c).

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

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 3.6.1). This is not performed by Android FDE, which may lead to data remanence attacks via the flash wear-levelling mechanism as discussed in Section 3.6.2.

We then generate an offset from the true password as described in Section 3.3.4. 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.

3.4.3 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 apparently sufficient to completely fill the logical address space twice, as noted by Wei et al. [102]. 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; see Figure 3.4. 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.

3.4.4 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 3.6.3.

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. [12]). 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 3.7, 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.

3.5 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; see Figure 3.7. 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 ad-

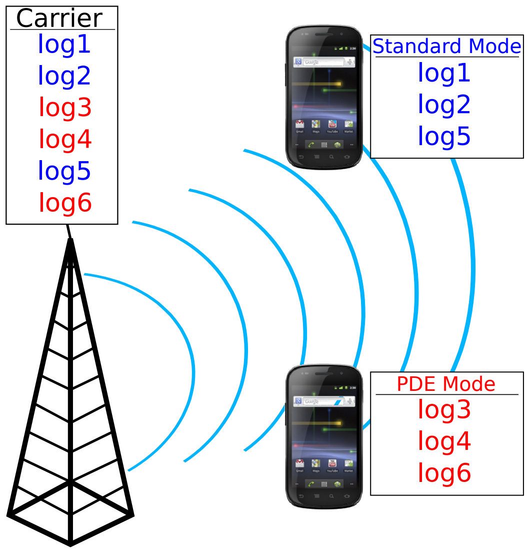


Figure 3.7: *The adversary may collude with a service provider; discrepancies between device logs and the carrier's activity logs may compromise deniability*

versary 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.

3.6 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.

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

3.6.1 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 [79]). However, certain statistical deviations between cipher and PRNG output may exist (see e.g., [30, 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 Appendix A).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 [43, 68] 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 [43]. 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.

3.6.2 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-levelling 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-levelling issues. Flash memory does not have the same data remanence issues as seen in magnetic storage. However, the wear-levelling 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-levelling 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-levelling mechanisms. This was the default

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

technology for devices prior to Android 3.0, but has largely been replaced by eMMC storage. The SD [86] and eMMC [49] specifications do not address wear-levelling requirements, so it is up to the manufacturers to decide if and how to implement wear-levelling in hardware FTLs.

Mobiflage stores all hidden volumes on the SD card. Therefore, exploiting the unmapped, wear-levelling 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. [102] 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 [41, pp. 266–284]); however, they tend to be costly, time consuming, and generally destroy the mobile device.

Wear-levelling 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. [102] show that most flash media contains between 6-25% more storage than advertised to the system. The additional

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

storage is used by the wear-levelling 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. [74] 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 [74].

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 [13], hence ensuring the continuity of random data on the underlying physical storage.

3.6.3 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 [55]. 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; see Figure 3.8.

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

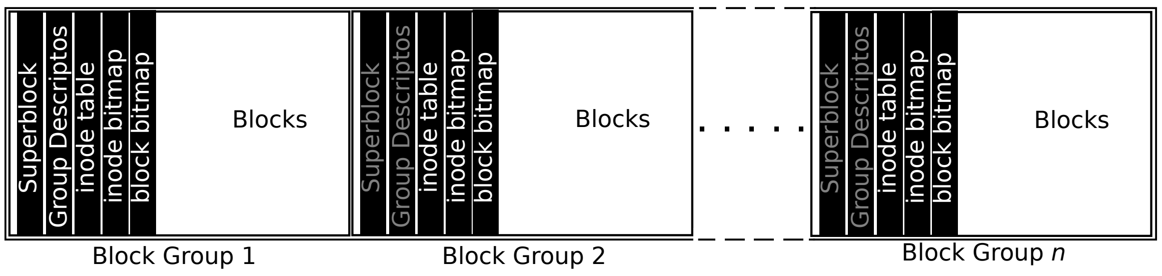


Figure 3.8: *Ext4 filesystem layout; overwriting the meta-data structures (dark regions) with hidden volume data would alert the adversary to the use of PDE*

group available on the disk [54]. 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).

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, and HFS+ that employ a more 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., [21]). 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 3.7 (item (c)) that log files, swap space, and temporary storage are effectively isolated between the two modes of Mobiflage.

3.7 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 time-out 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., [100]) can make offline guessing even more efficient. See Appendix B for more information concerning offline dictionary attacks against Android.

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., [58, 71]) can also be used to mitigate custom hardware attacks. In the end, we require users to choose a strong password resilient to guessing. We introduce a design in Chapter 4 that may mitigate a password guessing attack.

(b) Cipher issues. An implementation flaw can expose FDE ciphers to a theoretical watermarking attack that has been documented for software such as LUKS [20]. 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. [21]. 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.

Partial snapshots may also allow the adversary to learn the location of the offset: by identifying changes to regions in the latter portion of the disk (which likely coincide with hidden volume disk activity), and calculating the distance from the first FAT32 data block to the hidden offset. Knowing the offset location can speed up a password guessing attack. Currently, the adversary must execute the PBKDF2 function once. The result is used twice: to calculate the offset, and as an AES key to decrypt the volume-key at that offset. The adversary must then determine if the volume-key decrypts a valid filesystem (e.g., by identifying proper FS magic number). If the adversary is aware of the proper offset location, then he can skip the latter two steps, if the first operation does not produce the correct offset. This means each incorrect guess is identified before the two AES operations.

(e) Practical security of multiple hidden volumes. There is some debate over the effectiveness of multiple hidden volumes [28]. 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

Cipher-spec	Key-length (bits)	Speed (KB/s)		Speed reduction	
		Nexus S	Xoom	Nexus S	Xoom
Unencrypted	N/A	5880±260	4767±238	-	-
AES-CBC-ESSIV (Android 4.x)	128	5559±76	4168±186	5.46%	12.57%
AES-XTS-Plain64 (Mobiflage)	512 (256+256)	5288±69	3929±146	10.07%	17.58%

Table 3.2: *Observed read/write performance of external (SD card) storage; note that Mobiflage reduces IO performance by approximately 5% over Android FDE*

limitation of PDE schemes and may be alleviated (to some extent) by using a special password to make the hidden data permanently inaccessible (cf. [28]).

3.8 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 3.2 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 [46], 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.

3.9 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. We also discuss academic deniable-storage proposals. See Section 2.2, for information about PDE ciphers.

3.9.1 Software Implementations

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). To contend with a coercive adversary, PDE adds another layer of secrecy over FDE.

Most mobile OSes also offer data encryption (without PDE). BlackBerry devices use a password derived key to encrypt an internal storage AES key, and an ECC private key [77]. 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 (beyond the default UID-key wrapping, which only protects the data if the storage is removed and attacked without the device's crypto processor) [99, 2]. 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 [107]) to encrypt the device storage. WhisperCore enhances the raw flash file system, YAFFS2, which has been superseded on current Android devices in favour of the Ext4 file system.

Disk encryption software such as TrueCrypt [97] and FreeOTFE [31] use hidden volumes for plausible deniability. TrueCrypt is an open source encryption system available for Windows, Mac OS X, and GNU/Linux. The system is currently at revision 7.1a and has been around since early 2004. The software complies with ISO, NIST, FIPS, and PKCS standards and specifications. TrueCrypt offers encryption under several ciphers including AES, TwoFish, Serpent, and cascades of these ciphers in the XTS mode. In addition to FDE, TrueCrypt also allows for mounting encrypted file containers onto the filesystem. When encrypting a hard disk, TrueCrypt does not alter the partition table. In this manner, there is no signature or indication of the volume as being a TrueCrypt volume. TrueCrypt has withstood attacks (e.g., [75]), suggesting that it is a robust FDE implementation. The open source model affords expert advice and scrutiny, and weaknesses that have been identified are quickly rectified (e.g., PDE leakage issues [21]).

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 (i.e., a temporary root filesystem). 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 filesystem with the `pivot_root` system call. The TrueCrypt boot-loader (or RAM disk for Linux) is not encrypted and exposes the use of TrueCrypt to an attacker. An alternative configuration is to erase the boot-loader and volume key from the hard disk, and keep a copy of them on some external media (e.g., CD or USB storage). This can be seen as a form of two-factor authentication, since both the external media and user's password are required to boot the system. Configuring the system in this way does not reveal the use of TrueCrypt: as there is no need for an unencrypted boot-loader, the entire storage area will appear as random bytes.

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 boot-loader, implementing a soft keyboard, would be needed to capture the password (cf. [90]). 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. A custom boot-loader may speed up the boot procedure, since the framework would not be required to mount a tmpfs on `/data` while waiting for the user to enter their password. Mounting a tmpfs incurs additional overhead to pivot the `/data` volume (*unmount-then-mount*), but should not have any impact on security under our threat model.

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 some other OTFE frameworks, TrueCrypt does not store a hash of the passphrase (as with FreeOTFE), nor the cipher specification (as is stored in the Android footer) in the volume header. 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 [19]. 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).

FreeOTFE [31] is another open source project that offers FDE with PDE for Microsoft Windows. FreeOTFE does not currently implement a pre-boot authentication mechanism, so it is important that the boot partition is not encrypted (as of Windows Vista, this is a separate partition). The FreeOTFE volume header, known as the critical data block, is encrypted with the user's passphrase and contains the master volume key for the volume image (along with other information about the encrypted volume). Unlike TrueCrypt, FreeOTFE volumes can contain an arbitrary number of hidden volumes. Each of these hidden volumes will have an associated offset within the outer volume. A hidden critical data block is written to this offset, and the hidden volume data immediately follows. The user is tasked with choosing the offset to mark the beginning of the hidden volume. The user must also remember these offsets and supply them when mounting the hidden volumes.

3.9.2 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 [62] 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[7] 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. [70] design a system where blocks used by hidden files are in fact marked as occupied in the block bitmap. This alleviates reliability issues and

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

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 [112] 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 [38]) 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.

Other Linux deniable implementations, such as RubberhoseFS [3], and Magikfs,[8] employ techniques similar to StegFS for hiding data in file system free space. 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.

3.10 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. [21]). 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 at http://users.encs.concordia.ca/~a_skil/mobiflage/.

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

Chapter 4

Myphrase: Strong Multiword Passwords

This chapter describes a multi-word password scheme that aims to improve memorability and strength of user-chosen passwords. In our *Myphrase* scheme, a small dictionary is created from user-authored content such as sent emails and blogs. A master passphrase is constructed by randomly selecting words from the dictionary. Words in the passphrase are expected to be memorable to users and can be efficiently entered by leveraging the auto-suggest feature. Myphrase is discussed here in the context of web applications, as that provides an easy framework for evaluation and comparison against other authentication schemes. Myphrase passphrases offer better complexity than most user-chosen passwords, and as such Myphrase is a suitable key-protection mechanism. Our Android prototype can be used with Mobiflage (see Chapter 3), to provide strong protection for PDE keys.

4.1 Introduction and Motivation

To the dismay of security proponents and website administrators, many regular and expert users consistently choose *weak* passwords, such as, 123456, iloveyou, ieee2012, and princess (see e.g., leaked passwords from IEEE.org [26] and Rockyou.com [44]). Most users do not understand the security implications of choosing a password; passwords are just *words* with which they can get access to their digital resources. Thus, it is unsurprising that currently, common dictionary words and their predictable variants are heavily used as passwords. For users, this is apparently the most sensible choice for password creation and long-term management [39]. To exploit this behavior, offline and online password guessing attacks use common passwords as their starting point. To restrict these attacks, some websites forbid certain *obvious* passwords; see e.g., Twitter [91]. Password authentication is so deeply entrenched in modern technology that digital providers are reluctant to adopt new back-end security mechanisms. Most sites enforce password rules to restrain users from choosing passwords they desire; often these rules are too complex for users to cope with [45]. On the other hand, to facilitate user choice, Schechter et al. [84] suggested to place a threshold in the use of popular passwords; users are free to choose any password as long as that password has not been chosen by too many other users of the website. This may reduce the total number of compromised accounts on a given site, but does not protect the individuals that choose weak passwords.

Password entry from constrained input interfaces of mobile devices may further influence users to choose dictionary words as passwords (cf. [48]). Recently, multi-word password schemes have been revisited as an alternative to regular passwords by leveraging auto-correct and auto-suggest features in mobile devices [16, 47]. However, as long as users are free to choose words in a multi-word phrase, no significant improvement is apparent in password strength (see e.g., [59, 11, 104, 85]). In terms of usability, non-text authentication solutions for mobile devices have also been proposed (e.g., implicit authentication [48]). However, as many users access their password-protected accounts from both traditional computers and mobile devices, new solutions must consider both environments.

With *Myphrase*, we explore a different approach, which is apparently more in-line with user desire and can support both desktop and mobile platforms. Instead of discouraging users from choosing what they are comfortable with, we leverage users' *own* personal vocabulary to generate strong passwords; i.e., any words can be used irrespective of being considered too *obvious* or *taboo*. A Myphrase passphrase consists of multiple randomly chosen words from a user-created/selected dictionary. The dictionary is user-specific, e.g., generated from user-created text content, such as emails; users may also choose a pre-generated dictionary aligned with their interests e.g., lexicon from a favorite poet. Users can even use a list of common passwords as their dictionary, e.g., the 3546-word John the Ripper most common password list.[1] Dictionaries need not be private.

We examine two variations for passphrase construction. The first variant, called Random Sequence (RS), randomly selects words without regard for syntax. For an expected level of entropy, the required number of words in an RS-passphrase can be easily set; e.g., for 60 bits of entropy, six words must be chosen from a 1024-word dictionary. Memorability of RS-phrases is expected to benefit from the user's familiarity with the words (i.e., frequency of occurrence/use; cf. [82, 42, 35]). The second variant, called Connected Discourse (CD), constructs proper sentences using rudimentary natural language processing (NLP). Dictionary words are tagged using a part-of-speech (POS) engine, and inserted into pre-created sentence templates (cf. Mad Libs [73]). Calculating entropy for this variant is not as straightforward as the RS variant; entropy depends on the selected sentence template and the breakdown of words in each POS category from the dictionary. Memorability of CD-phrases is expected to benefit

[1]http://www.openwall.com/passwords/wordlists/password-2011.lst

from high-order syntactic and semantic structures (cf. [63, 25, 60, 29, 61]), in addition to word familiarity.

To reduce memory load, we expect users to memorize only one Myphrase master passphrase, and use it across multiple websites. A site password is generated by salting the phrase with the site's URL domain; the salted password is also hashed and converted into a server-compatible password (e.g., consisting of alphanumeric characters only).

We test Myphrase by creating dictionaries for users in the Enron email corpus and authors from Project Gutenberg. We find that our frequency ranked dictionaries have similar POS break-down, as compared to text parsed from a large collection of prose. We also compare the similarity between user's dictionaries to determine how personal or unique they are. We used these findings to identify sentence templates that maximize a passphrase's complexity.

In summary, Myphrase offers the following benefits.

1. Passphrase from user-specific words: Myphrase introduces a middle-ground between conflicting choices for text passwords: machine-generated (strong but memory-unfriendly), and user-chosen (memory-friendly but weak). Myphrase phrases are machine generated but consist of a user's personally-meaningful words; these passphrases are expected to be both memory-friendly *and* strong.

2. Stronger passwords: To restrict offline attacks, Myphrase passwords offer a significant entropy gain; e.g., 60 bits in Myphrase's default setting (cf. an estimated 10-20 bits of entropy in current passwords [9]). Passwords are also further strengthened using the PBKDF2 key-stretching function by a factor of

2^{15}. The exact entropy of Myphrase passwords can easily be determined for the RS variant and closely approximated for the CD variant. Entropy of user-chosen passwords can at best be roughly estimated (see e.g., [14, 105]).

3. Scalable and resilient to site-password leaks: Users need to memorize only one Myphrase master passphrase and can use it for all web logins—irrespective of varying levels of site security. If a site-password is leaked from the server-side, the user can update only that password, while maintaining their master passphrase and all other site-passwords.

4. Cross-platform/device compatibility: Auto-suggesting words after typing (or tapping) a couple of characters may reduce the input time of the passphrase; such a feature makes Myphrase suitable for mobile devices with a constrained/touch-based keyboard. We have implemented proof-of-concept prototypes for both desktop and mobile platforms.

Additionally, Myphrase passwords are resilient to phishing attacks. Attackers get a password specific to their phishing domain, instead of the target domain; they also do not receive the master passphrase. These benefits of Myphrase and limitations are explained further in Section 4.5.

4.2 Myphrase Description

Below we describe Myphrase, including our assumptions and user steps.

Operational and threat model assumptions. The custom dictionary can be generated from different types of user content, such as: (a) user-authored content, e.g., sent emails, tweeted messages, blog posts, comments at social networking sites, academic papers, and other such documents; and (b) user-liked content, e.g., favorite ebooks, song lyrics, and emails from certain contacts. Alternatively, users may select a pre-generated dictionary according to their interests or familiarity such as: an urban dictionary, medical or technology dictionary. For the connected discourse (CD) variant of Myphrase, the dictionary must contain the following POS categories: nouns, verbs, adjectives and adverbs. However, the random sequence (RS) variant has no such constraints (e.g., the dictionary may be comprised entirely of nouns, as in names of cities or film actors). The dictionary itself and sources of the dictionary words are not security-sensitive, and can be made public. However, the dictionary may be privacy-sensitive, if users do not want to reveal that certain words appear in their dictionary.

We assume that words in the created/selected personal dictionary are familiar enough to users that they can memorize a relatively long sequence (e.g., six) of these words as their master passphrase; see Section 4.5.4 for a discussion on memorability. To make use of auto-suggest, for minimal typing and easy recognition, Myphrase requires the dictionary be available in all user devices (e.g., via manual copy, email attachment, browser sync mechanisms or web hosting). The dictionary can be re-created from previously selected sources. Similar to regular passwords, the Myphrase master passphrase is vulnerable to host malware and shoulder-surfing attacks; such attacks are out-of-scope.

Myphrase design. When constructing the user's dictionary, we rank words by occurrence and keep only the most frequently used words. During this operation, we omit 215 common conjunctions, articles, prepositions, and pronouns (e.g., 'at', ‘and’, ‘she’) from the frequency ranking, and append them to the dictionary before passphrase generation. These words are necessary parts-of-speech, and must be present regardless of their observed frequencies (for the CD variant). Conversely, these parts-of-speech tend to be over-represented in a user’s dictionary, as compared to the POS breakdown observed in a large collection of prose. Removing these words from the frequency ranking should allow us to capture more personally meaningful, and hopefully memorable, words from the input. This also allows us to more accurately estimate the entropy for a given CD passphrase; see Section 4.5.1. We experimented with different lists of common words ranging from 100 to 1000. We found that 215 was the optimal size to ensure enough variation was available for the CD sentence templates, and balance dictionary POS break-down and passphrase entropy.

In Myphrase, passphrases are generated by randomly selecting words from the user’s vocabulary in the following ways. (*i*) Random sequence: n words are randomly selected from the user’s dictionary. (*ii*) Connected discourse: words are classified based on their POS tag (e.g., verb, noun); an n word-long sentence template is randomly selected from a pool of pre-created templates; dictionary words are then randomly chosen for each position in the template, based on their POS class.[2] The value of n depends on the dictionary size and expected level of entropy. Currently, we recommend that the passphrase offer at least 60 bits of entropy to restrict offline dictionary attacks; e.g., $n = 6$ for a 1024-word dictionary for the RS variant. The entropy for a CD passphrase cannot be directly calculated. Through experimentation we found that the complexity of a CD-phrase is roughly 65% of an equivalent

[2]This variant resembles the popular word game Mad Libs [73], in which players in turn choose words of particular types to fill-in the blanks of a given sentence template. However, there are no fixed words in our templates, and all words are chosen randomly from a user’s vocabulary.

length RS-phrase, so $n = 8$ words from a 4096-word dictionary is recommended; see Section 4.5.1 for details.

Specific words in the generated passphrase can be indicated for replacement, and Myphrase will offer another random word. However, entropy of the phrase may suffer, depending on the dictionary size and the number of iterations used for a specific word. We limit the loss of entropy by restricting the number of times a user may selectively regenerate words in the phrase to n, e.g., 8 times for an 8 word passphrase; see Section 4.5.2.

The master passphrase is used to generate unique site-specific passwords as follows:

$$pwd = \text{Hash2Text}(h^i(passphrase||domain||updatecount)) \tag{4.1}$$

Here, h is a cryptographic hash/key-stretching function, e.g., SHA-1 or PBKDF2; i is the number of hash iterations e.g., $i = 32768$; and *updatecount* is the number of updates made to a specific site's password (by default $updatecount = 0$, see also "Updating site passwords" below). A higher value of i will make brute-force attacks on the passphrase more computationally intensive, but values that are too high may slow down password generation. To increase the cost of offline cracking attacks, we use PBKDF2 in our implementation, which is more complex than SHA-1 alone, when implemented in a custom circuit. Recently proposed computational and memory-wise expensive functions can also be used (e.g., [58, 71]). The Hash2Text function encodes the binary hash result into a server-compatible password, e.g., passwords with only alphanumeric characters; cf. PwdHash [80].

User steps: dictionary and master passphrase generation. The following steps are required to generate the master passphrase; see Figure 4.1. ***(a)*** Users first select

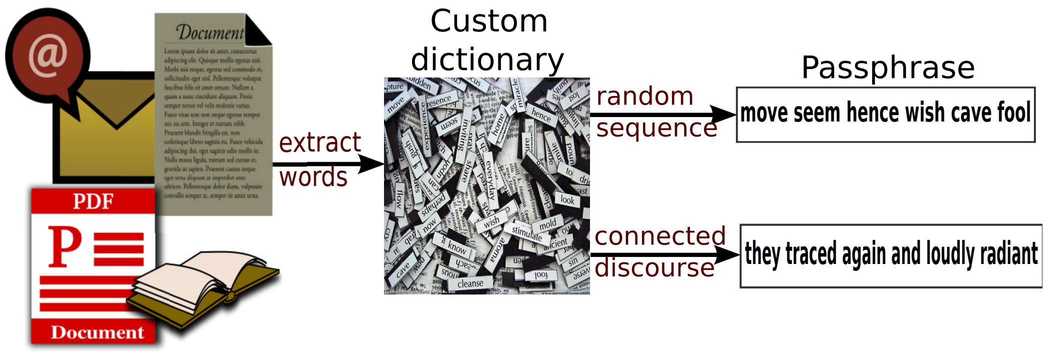


Figure 4.1: *Myphrase basic mechanism with RS and CD variants*

word sources for their custom word dictionary, e.g., emails, and ebooks. ***(b)*** The Myphrase tool extracts words from these sources, ranks the words by frequency, and selects a pre-specified number (e.g., 2048) of the most frequent words as the dictionary. After creation, users may choose to manually update the words in their dictionary. ***(c)*** The tool creates a multi-word passphrase from the dictionary created in the previous step (or a pre-selected dictionary) using the RS or CD variant. ***(d)*** Users can continue generating new phrases until they are satisfied. When the user is satisfied with a passphrase, she is expected to memorize the word sequence verbatim, or to write it down in a *secure* place, (cf. [111, 40]). The exact spelling of words need not be memorized due to the auto-suggest feature of Myphrase. However, users must not rearrange the words in a phrase, as such modifications may reduce effective entropy.

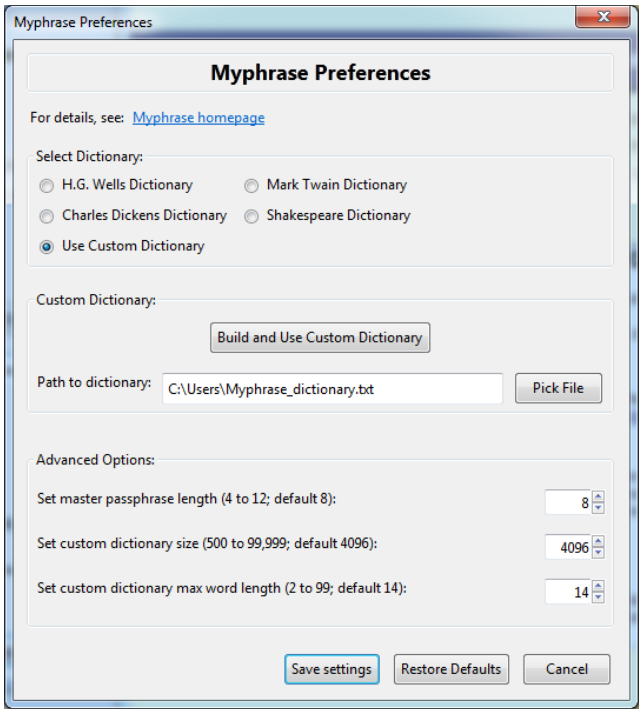
User steps: site-specific password generation. ***(a)*** Users enter their passphrase; assuming the dictionary is available as a browser add-on or with the Myphrase application, users need to type only the first couple of characters for each word and then select the correct word from the auto-suggestion list. ***(b)*** The site-specific password is created using the formula shown in Equation 4.1. The site password is sent to the authenticating site.

Updating site passwords. Users may want to update their Myphrase site passwords for various reasons, including a periodic password update policy, and passwords compromised at server-side (e.g., LinkedIn's 6.5M password leak [6]). We allow password update without requiring the master passphrase to be changed, by increasing the value of *updatecount* (see the site-password generation formula in Equation 4.1). This value is site-specific and starts from zero; when a site's password needs to be updated, *updatecount* is incremented by one, and the updated value is used only for the target site (i.e., other site passwords will still be generated with $updatecount = 0$, if not already changed). This solution is similar to the index mechanism as proposed by Halderman et al. [37]. Site URLs with updated count values must be stored and synced across devices; e.g., via Firefox Sync.

4.3 Implementation

We implemented Myphrase for PCs as a Firefox addon, and for Android devices as a custom soft-keyboard. The desktop version provides an interface to build dictionaries, generate passphrases, and insert site passwords. The mobile version currently makes use of dictionaries and passphrases generated in the desktop version to insert passwords into websites. A web interface is also created to facilitate password generation when other tools are unavailable (e.g., a temporary device). We currently handle only English words; internationalization would require POS taggers for additional languages. Details are discussed below.

Preferences. The Firefox addon offers a few user customizable settings; see Figure 4.2. We provide some pre-built dictionaries, but encourage users to create per-


Myphrase Preferences

Myphrase Preferences

For details, see: Myphrase homepage

Select Dictionary:

H.G. Wells Dictionary Mark Twain Dictionary

Charles Dickens Dictionary Shakespeare Dictionary

[✓ Use Custom Dictionary]

Custom Dictionary:

Path to dictionary: C:\Users\Myphrase_dictionary.txt

Advanced Options:

Set master passphrase length (4 to 12; default 8): 8

Set custom dictionary size (500 to 99,999; default 4096): 4096

Set custom dictionary max word length (2 to 99; default 14): 14

Figure 4.2: *Myphrase preferences*

sonalized ones. To help guide the user experience, we provide the following default options: (*a*) type of master passphrase: connected discourse; (*b*) dictionary: lexicon from the works of H. G. Wells; (*c*) length of master passphrase: eight words; (*d*) size of custom dictionary: 4096 words; and (*e*) maximum word length: 14 characters. The addon also has site-specific settings for each generated site password. These allow Myphrase to conform to provider specific constraints, and store each site's password update count. The defaults are as follows: (*a*) length of site passwords: 12; (*b*) special characters: disabled; and (*c*) password update count: 0.

Dictionary construction. The Firefox addon can build a personal dictionary from the user's outbound emails, and plain-text, HTML and XML files. Our text and HTML/XML parsers can be used to build dictionaries from PDF and other document formats after saving as text, ebooks (e.g., ePub, a compressed XML format), and web content saved as HTML. For email parsing, we choose to collect messages from the Simple Mail Firefox addon [92], which enables managing several email accounts, including POP/IMAP exposed webmails. It aggregates emails across accounts in a single folder structure (i.e., emails sent from all accounts are stored in one "Sent" folder). This allows us to gather words that may originate from disparate vocabularies (e.g., work emails with professional terms vs. personal emails with slang or informal words). Sent messages are collected by querying an SQLite database. The message text is parsed to extract the user's original text; i.e., we eliminate quoted text from forwarded or replied emails.

Using a series of regular expressions, we decode HTML/XML entities and eliminate mark-up, punctuation, digits, and other non-word strings. By default, we capture strings that are 14 characters or less. Note that short words do not compromise the security of Myphrase; however, longer words may require more typing if the user's dictionary is not available for auto-suggest completion.

We retain the top 4096 words in the user's dictionary by default. All characters are reduced to lower-case before the word frequency analysis. The dictionary is saved as a text file, which the user may further customize (manually). Users may also choose to use a pre-built dictionary; we provide a few default dictionaries created using top free ebooks from Project Gutenberg.

Master passphrase generation. After selecting a dictionary, users can generate a passphrase in the Firefox addon. To select random words from the dictionary, we use Mozilla's PRNG, *nsIRandomGenerator* (assumed to be cryptographically secure; uses mouse movement events within the Firefox window). For the RS variant, a passphrase is generated simply by selecting n random words from the dictionary. For the CD variant, we first randomly choose a pre-built sentence template for the given passphrase length ($n = 4$ to 12 words). A number of templates were created by parsing top free ebooks from Project Gutenberg, and selecting a few which appear to yield high entropy passphrases (discussed more in Section 4.5.1). We then use a Javascript POS tagger [101] to classify the dictionary words. A passphrase is generated by filling the template with randomly selected words from the classes that appear in the template. For both variants, users can regenerate new phrases until they are satisfied; a limited number of selective regeneration of words in a selected phrase is also allowed. The master passphrase is not stored within our software, and cannot be regenerated; users are expected to memorize it.

Site password generation. We anticipate that users will build personal dictionaries and generate passphrases only occasionally. The day-to-day use of Myphrase will largely consist of deriving site passwords. When faced with a login page, the user right-clicks on the password field and selects the Myphrase insert option from the browser context menu. On a mobile device, the user can pull down the notification bar, to switch from their default keyboard to the Myphrase keyboard; see Figure 4.3.

The user is then prompted to enter her master passphrase. The prompt allows the user to view the passphrase text, or have it shadowed in the case she is in a public place. An auto-complete suggestion list is populated with words from the user's dictionary to speed up this task; see Figure. 4.4. This feature is especially useful on mobile devices where typing tends to be slower. Passphrases are constructed in such a way that the user will not require any modifiers (e.g., shift) or changing keyboard views (e.g., numbers, special characters) which should greatly increase entry time for mobile devices. At this stage, the user can also set site-specific constraints

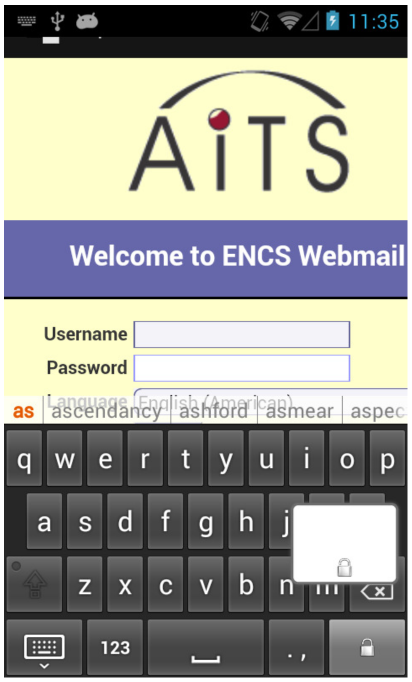
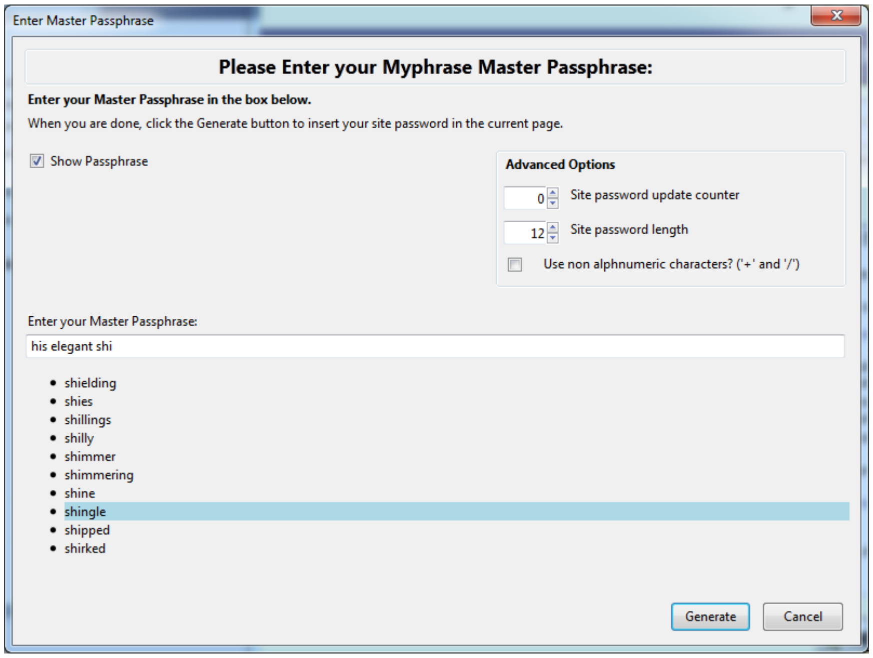


Figure 4.3: *Myphrase mobile keyboard; the auto-complete suggestions are visible above the keys*



Enter Master Passphrase

Please Enter your Myphrase Master Passphrase:

Enter your Master Passphrase in the box below.

When you are done, click the Generate button to insert your site password in the current page.

☑ Show Passphrase

Advanced Options

0 Site password update counter

12 Site password length

☐ Use non alphnumeric characters? ('+' and '/')

Enter your Master Passphrase:

his elegant shi

- shielding
- shies
- shillings
- shilly
- shimmer
- shimmering
- shine
- shingle
- shipped
- shirked

Figure 4.4: *Myphrase site password generation – Firefox addon*

(e.g., password length) and increment the update counter. The update count and constraints for each domain are saved as site specific preference within Firefox and Android, so will only need to be set by the user once.

We then iterate the PBKDF2_HMAC_SHA-1 function 32768 times, using the passphrase as the HMAC secret, and the site's domain and update count as the salt; see Equation 4.1. The choice of our iteration count is explained in Section 4.5.2. The result is then encoded into a compatible password and inserted in the password field. Note that, both the addon and Android soft-keyboard make only the site-password available to the authenticating domain, which could be a legitimate or phishing site. The master passphrase remains inaccessible to all websites.

Myphrase mobile. We implemented Myphrase for Android devices as a custom soft-keyboard. This enables Myphrase authentication for any password field on the device including websites, apps and even device unlock; the current prototype implementation supports only website login with the full feature set. Users can quickly switch to the Myphrase keyboard from the Android notification bar to enter a password. For the Android/Mobiflage pre-boot authentication mechanism, the application package must be installed as part of the system framework, as opposed to installation in the user application store. The user's text is displayed above the keyboard while they are typing, but not entered into the password field until they press Enter. As with the desktop software, auto-complete dictionary suggestions are displayed to speed up passphrase entry. The auto-complete feature can only be made available to the pre-boot authenticator when an unencrypted SD card is present (i.e., auto-complete suggestions are not available during pre-boot for Mobiflage encrypted devices). After the boot process, the dictionary is available for all other authentication attempts (e.g., web sites and apps).

4.4 Related Work

Multi-word passwords and several-related variants including first-letter mnemonics have been proposed decades ago (see e.g., [5, 72, 59]). Similar schemes have been recently revived to increase password entropy,[3] and to make password input more user-friendly in devices with touch-screen keypads [16, 47]. The use of readily available auto-correct and auto-suggest features can significantly reduce input issues in these devices (e.g., compared to inputting a password with mixed case letters and special characters). Generally, there are two types of multi-word passwords: words and their sequence chosen by a user (e.g., [72, 47]), and words selected randomly from a fixed, system-chosen dictionary (e.g., Cheswick [16]). Cheswick's proposal uses a 1020-word dictionary of *iPhone-friendly* English words. Diceware [76] is another random passphrase generator using one or more dice as the random number generator. Each word in the phrase is chosen from a five digit number generated by five rolls of a die. Any list of 7776 (6^5) unique words can be used; word lists are currently available in several languages.

Smith proposed a word association authentication scheme [89] that requires users to register a list of (cue, response) pairs consisting of personally-meaningful words; obvious pairs such as (black, white) are disallowed. During login, users must provide correct responses to cues chosen randomly by the system. As cues and responses are user-selected and likely to vary significantly from user to user, Smith argues that this

[3]For example, see the popular XKCD cartoon at http://xkcd.com/936/.

scheme would offer adequate benefits in terms of user-acceptance, memorability and security.

The use of natural language processing techniques to create multi-word passwords has also been explored. Atallah et al. [4] generate a meaningful/humorous phrase and associated mnemonic for authentication. Another technique [50] creates sentences for a given random password. News headlines are used as templates by substituting similar words; several variants of the same sentence are generated using a POS tagger and WordNet [65]. To generate multiple passwords from a master mnemonic sentence, Topkara et al. [95] propose to split a password into two parts: a memorized mnemonic sentence that is used to perform a table-lookup on a helper-card to derive site passwords.

User-chosen phrases or word sequences are almost as memorable as regular passwords [109]; it has long been known that memorability of a sequence of items such as words or phonemes is dependent on familiarity of those items [64, 42, 35]. However, such phrases do not offer much improvement in terms of entropy as users generally choose common phrases; see e.g., Kuo et al. [59]. Another recent analysis of over $100,000$ possible user-selected passphrases from the Amazon PayPhrase system also reported similar results [11]. Generic attack techniques against multi-word passwords have also been proposed; see e.g., Weir [104], Schmitz [85]. In contrast, random words from a system-chosen dictionary offer better entropy, but may not be ideal in terms of memorability. We allow users to reuse the same passphrase safely for all web logins, reducing their memory load and combating the multiple password interference problem [17].

4.5 Comparison and Evaluation

In this section, we discuss entropy of passphrases for both RS- and CD-Myphrase variants, memorability of the passphrases and limitations of the scheme. We also use a slightly modified version of the recently-proposed UDS (usability, deployability, security) framework [10] for an analytical evaluation of Myphrase.

4.5.1 Myphrase Entropy Estimation

Equal-length CD-phrases will provide a lower entropy count than RS-phrases, as each template element is being chosen from a subset of the dictionary words. Experimental entropy estimation of CD-phrases is discussed below. We also allow selective regeneration of words in a phrase. By keeping the regeneration count small, we can limit the loss of entropy. We provide an analysis of selective regeneration and estimated efforts required to launch brute-force guessing attacks against Myphrase passwords in Section 4.5.2.

By default, we expect users to choose six words from a 1024-word dictionary for the RS variant (providing 60 bits of entropy), and eight words from a 4096-word dictionary for the CD variant (providing 60 bits of entropy on average). For easy memorization, users may want to reduce the word count, especially for the RS variant (e.g., three words). This may still provide higher entropy than current passwords (cf. 10-20 bits of entropy as found in a large-scale password study [9]). However, such a three-word master password can be practically retrieved from a leaked site password; this may mandate changing all site passwords. If the adversary has the user dictionary, he can attempt all 3 word combinations until he finds a match with the leaked site password.

On average, this will require 2^{29} attempts. Without the dictionary, he will need to brute force lower-case strings of indeterminable length (the worst case scenario is 3 one-letter words, which only provides about 14 bits of entropy). As high-impact password leaks happen not so rarely (e.g., [26, 6, 88]), we suggest users to invest in a longer master passphrase. Creating a larger dictionary will also increase the effective complexity of the passphrase.

Experimental entropy estimation of the CD-Myphrase variant. We obtain real world entropy estimates of CD-phrases by examining the POS breakdown for two datasets: Enron emails (from www.cs.cmu.edu/~enron/) and Project Gutenberg ebooks. We also test similarity between user dictionaries.

To generate sentence templates, we parse the 25 most popular ebooks from Project Gutenberg between the period of Aug. 29–Sept. 28, 2012. We identify $60,921$ unique strings. The POS class breakdown is as follows: nouns (71%), verbs (18%), adjectives (7%), adverbs (3%), and others (1%). We then select sentence templates from the parsed text, capitalizing on sentences that contained more of the highly populated POS classes. We originally chose ten templates for each possible passphrase length (4 to 12 words) that we believed would yield high entropy phrases; after evaluation, we retain the 7 highest performing templates for each length. The sentence templates used in the 8-word Myphrase connected discourse variant are listed in Table 4.1.

When constructing a user's dictionary we rank words by frequency and select only the most common words in the user's vocabulary. This could skew the user's POS breakdown in such a way that our selected templates would not produce ideal passphrases (e.g., if the frequency ranked dictionaries contain more verbs than nouns). We chose to analyze the Myphrase dictionaries for selected ebook authors and Enron

⟨NN⟩	⟨VB⟩	⟨IN⟩	⟨DT⟩	⟨NN⟩	⟨VB⟩	⟨IN⟩	⟨NN⟩
⟨DT⟩	⟨NN⟩	⟨VB⟩	⟨RB⟩	⟨VB⟩	⟨IN⟩	⟨DT⟩	⟨NN⟩
⟨NN⟩	⟨CC⟩	⟨NN⟩	⟨VB⟩	⟨RB⟩	⟨IN⟩	⟨DT⟩	⟨NN⟩
⟨IN⟩	⟨PRP⟩	⟨VB⟩	⟨RB⟩	⟨PRP$⟩	⟨NN⟩	⟨VB⟩	⟨RB⟩
⟨DT⟩	⟨JJ⟩	⟨NN⟩	⟨RB⟩	⟨VB⟩	⟨TO⟩	⟨DT⟩	⟨NN⟩
⟨RB⟩	⟨VB⟩	⟨JJ⟩	⟨NN⟩	⟨VB⟩	⟨TO⟩	⟨DT⟩	⟨NN⟩
⟨CC⟩	⟨PRP$⟩	⟨VB⟩	⟨JJ⟩	⟨NN⟩	⟨VB⟩	⟨VB⟩	⟨PRP$⟩

Table 4.1: *Myphrase 8-word connected discourse templates; where ⟨NN⟩ is a noun, ⟨VB⟩ is a verb, ⟨IN⟩ is a preposition, ⟨DT⟩ is a determiner, ⟨RB⟩ is an adverb, ⟨CC⟩ is a conjunction, ⟨JJ⟩ is an adjective, ⟨PRP⟩ is a pronoun, ⟨PRP$⟩ is a possessive pronoun, and ⟨TO⟩ is the word "to".*

employee emails. For ebooks, we choose 7–10 well-known works from ten authors. The Enron email corpus contains 150 unique user accounts. We isolate each user's personal vocabulary by parsing only sent emails after eliminating quoted text, signature blocks, etc. There were only 15 users with dictionaries of at least 4096 words (we also discarded 2 dictionaries containing several anomalies, e.g., mid-word line-breaks). We found both the Enron employees and ebook authors had similar POS breakdowns in their frequency adjusted dictionaries, validating our template selections; see Table 4.2. We also noticed earlier that some parts-of-speech (e.g., conjunctions, pronouns) were over-represented; consequently, we remove the 215 common words from those classes before frequency ranking the list.

	Composition %			
	Gutenberg authors		Enron authors	
	Avg	Stdev	Avg	Stdev
Nouns	50.29	1.94	63.31	2.84
Verbs	29.27	1.77	22.68	1.70
Adjectives	14.19	0.98	9.16	0.85
Adverbs	6.06	0.60	4.44	0.45
Others	0.19	0.12	0.41	0.08

Table 4.2: *Frequency adjusted POS classification for 4096-word popular Project Gutenberg author dictionaries (N=10) and Enron dictionaries (N=15)*

	Entropy count in bits							
	Gutenberg authors				Enron authors			
Length	Min	Max	Avg	Stdev	Min	Max	Avg	Stdev
6 words	41.5	57.5	47.8	5.4	41.2	57.0	47.0	5.4
7 words	47.7	62.6	54.9	4.7	46.5	61.5	53.7	4.4
8 words	58.1	71.0	63.9	3.9	57.2	71.2	63.1	4.4

Table 4.3: *Observed entropy for phrases from 4096-word Gutenberg and Enron dictionaries. Large deviation is a result of different templates. Each template has little variability as seen in Table 4.4.*

We then used the created Myphrase dictionaries to calculate expected entropy from each template. Entropy for RS passphrases can be directly calculated, e.g., six words from a 1024-word dictionary will result in: $log_2(1024^6) = 60$ bits. The entropy of a CD passphrase depends on the template and POS breakdown of the user's dictionary. For example, assume the chosen template is: "noun verb adverb determiner noun verb preposition noun" with POS class sizes (noun, 1975), (verb, 1209), (adverb, 86), (determiner, 21), (preposition, 86). A randomly generated passphrase from this template will provide an entropy of $log_2(1975 \times 1209 \times 86 \times 21 \times 1975 \times 1209 \times 86 \times 1975) = 70.57$. We performed the calculations on 6, 7, and 8 word phrases – see Table 4.3; on average, these CD phrases retain about 65% of the entropy compared to RS-phrases of respective lengths. The large deviations are a result of the different templates within a given phrase length. The templates provided similar results regardless of the dictionary used. This would suggest that we can further narrow the entropy estimate for a given phrase length by adjusting the templates rather than the dictionary. For example, template five for eight word phrases (see Table 4.1) performed the worst, and could be eliminated. The results of 8-word templates are shown in Table 4.4.

Template	Avg	Stdev	% of RS
1	71	0.17	72
2	66	0.32	68
3	66	0.12	68
4	61	0.83	63
5	58	0.44	60
6	63	0.64	66
7	60	0.94	62

Table 4.4: *Observed entropy for 8-word templates; combined results from both Enron and Gutenberg authors (N=25).*

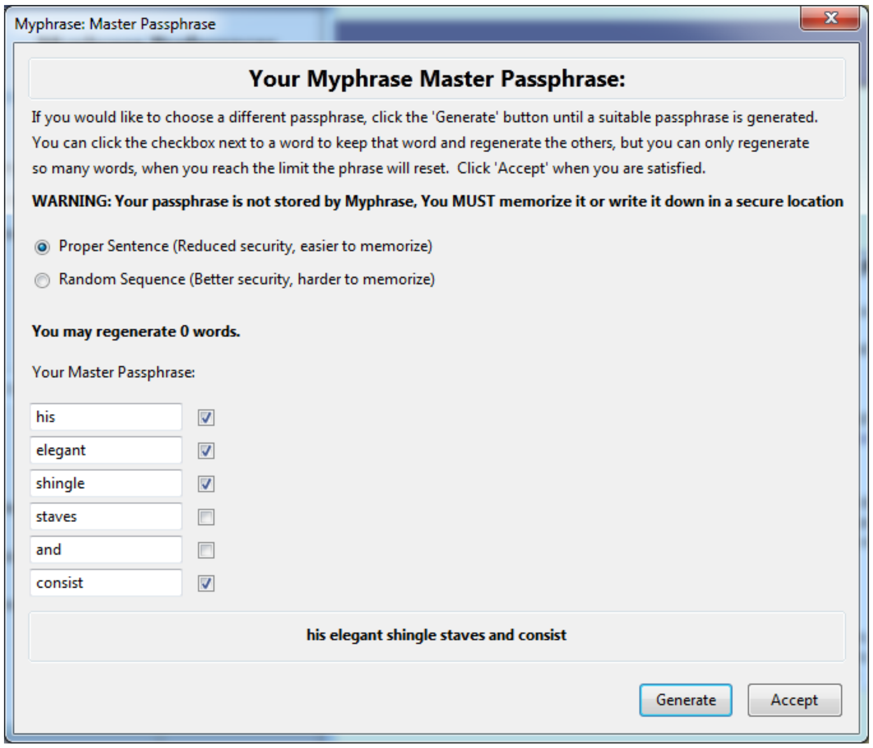
Dictionary uniqueness. We also compared the similarity between the dictionaries to measure how unique they were to each user. For every pair of user dictionaries in both Gutenberg and Enron datasets, we calculated the Jaccard index (i.e., the size of the intersection between two dictionaries divided by the size of their union); see Table 4.5. The results show that each user dictionary is relatively personal (the average similarity is between 31-43%).

4.5.2 Selective Regeneration and Brute-force Attacks

Selective regeneration of words. The user is allowed to selectively regenerate individual words in the passphrase; see Figure 4.5. Each time the user discards a word they reduce the available choices and weaken the passphrase. We restrict the number of re-selections to the number of words in the phrase (e.g., the user can regenerate 6

	Jaccard index %			
	Min	Max	Avg	Stdev
Gutenberg	34.84	49.59	42.58	3.62
Enron	23.38	41.82	30.52	3.48

Table 4.5: *Similarity of user dictionaries (measured as Jaccard index of pairs of users from both Gutenberg and Enron datasets)*


Myphrase: Master Passphrase

Your Myphrase Master Passphrase:

If you would like to choose a different passphrase, click the 'Generate' button until a suitable passphrase is generated. You can click the checkbox next to a word to keep that word and regenerate the others, but you can only regenerate so many words, when you reach the limit the phrase will reset. Click 'Accept' when you are satisfied.

WARNING: Your passphrase is not stored by Myphrase, You MUST memorize it or write it down in a secure location

● Proper Sentence (Reduced security, easier to memorize)

○ Random Sequence (Better security, harder to memorize)

You may regenerate 0 words.

Your Master Passphrase:

his	✓
elegant	✓
shingle	✓
staves	
and	
consist	✓

his elegant shingle staves and consist

Figure 4.5: *Myphrase passphrase generation with selective regeneration of individual words*

words when constructing an 6 word passphrase) in order to ensure the user does not introduce too much predictability. For example, if a RS passphrase consists of six words from a 1024-word dictionary then the entropy will be: $log_2(1024^6) = 60$. If the user selectively regenerates six words, the entropy is reduced to: $log_2(1018^6) = 59.95$. Since only a few iterations are allowed, the entropy will not suffer significantly. If the user finds many undesirable words, it would be appropriate for them to consider creating a new dictionary from different sources.

The selective regeneration feature will probably be more useful for the CD variant. Some selected words will not create coherent sentences. A user is more likely to reject these words, which decreases the available combinations for that template. For example, assume the chosen template is: "noun verb adverb determiner noun verb preposition noun" with POS class sizes (noun, 1975), (verb, 1209), (adverb, 86), (determiner, 21), (preposition, 86). A randomly generated passphrase from this template, e.g., "discomfort rang down the sunlight wait of freedom" will provide an entropy of $log_2(1975 \times 1209 \times 86 \times 21 \times 1975 \times 1209 \times 86 \times 1975) = 70.57$ bits. Assume the user regenerates the following words: (discomfort$\rightarrow$clamor), (the$\rightarrow$a), (sunlight$\rightarrow$heap$\rightarrow$yelp$\rightarrow$spirit) and (wait$\rightarrow$answering$\rightarrow$alter$\rightarrow$contrived). Now the sentence becomes: "clamor rang down a spirit contrived of freedom" and the modified entropy is: $log_2(1971 \times 1206 \times 86 \times 20 \times 1971 \times 1206 \times 86 \times 1971) = 70.48$ bits. As appears from this example, after a few iterations, the sentence may converge to something acceptable to the user, without losing much entropy. However, if an attacker uses sophisticated natural language processing, they may be able to determine which words are more likely to be rejected, thereby narrowing their search space. A larger dictionary can help offset this loss of complexity.

Iteration count and brute-forcing Myphrase passwords. We experimented with different hash iteration counts, and found that 32768 caused an acceptable delay for our PC (Firefox addon) and smartphone (Android app) implementations. Running on a 2.5GHz Intel i7-2860 CPU, it required 2.84 ± 0.02 seconds to complete for the addon. On a 1.2GHz ARM Cortex-A8 HTC smartphone it required 3.23 ± 0.06 seconds. With assembler and hardware crypto accelerated instructions (e.g., OpenSSL) the time to iterate PBKDF2 reduced to 0.3 seconds on average. Note that, the extension adds almost 10 times more inefficiency in the PBKDF2 calculation, which benefits the attacker. Native code execution within the browser, e.g., Google Native Client [110], can be used to reduce this inefficiency. An attacker can also parallelize the computations to brute-force the dictionary. With a 60-bit passphrase, the attacker will need to perform 2^{59} calculations on average. On a 4-core PC this will require: $\frac{2^{59}\times 0.3}{60\times 60\times 24\times 365.25\times 4} = 1370020420$ years to search this space. If the attacker has access to a grid of 1 million such 4-core CPUs (e.g., a million-node botnet), it will still require 1370 years; without the hash iteration, the required time is only about 15 days (on average). For a 40-bit passphrase, the required times on the million-node grid are about 12 hours (with hash iteration) and 1.25 seconds (without hash iteration) on average. An attacker may be able to reduce the time using custom hardware (e.g., FPGAs, ASICs, or GPU arrays) or efficient hashing algorithms (cf. transferable state attack [100]). On the other hand, the attacker's workload to brute-force a target password will increase significantly if the server-side stores only a one-way mapping of the password obtained through the use of iterated hash/PBKDF2 functions with unique salt values per account. If the adversary attempts to brute-force the passphrase without the dictionary, he will need to attempt lower-case strings. Since the passphrases are combinations of words, the adversary can take advantage of the fact that these strings will not be random (e.g., using Markov chains or a weighted prefix trie). In the worst case scenario, the passphrase could be the (unlikely) combination of 6 one-character words. The workload will then be at most $log_2(26^6) = 28.2$. The search space for brute-forcing an individual site password, under Myphrase's default settings, is $log_2(62^{12}) = 71.45$

4.5.3 UDS Evaluation of Myphrase

We now provide an analytical evaluation of Myphrase using the recently-proposed UDS (usability, deployability, security) framework [10]. We modified the ratings slightly, from the original three point scale, to include a fourth *partial-benefit* rating. The *partial-benefit* rating exists between the original *no-benefit* and *quasi-benefit* ratings, and indicates that a given benefit is weakly, or only partially met. The *quasi-benefit* rating still indicates that a given benefit is almost fully met. For context, we also include the ratings for regular user-chosen passwords, the Firefox password manager, and the online password manager LastPass[4] from the original UDS evaluation. We also evaluate some similar schemes based on NLP, multi-word secrets, or site-specific passwords generated from a master secret. Due to space limitation, we refer readers to the UDS paper [10] for details of the framework and feature definitions. We have not conducted any formal user-testing yet; to help better design such tests, we would like to get expert feedback and comments on our publicly available prototype. Thus we would like to emphasize that our usability ratings for Myphrase within the UDS framework are only best guesses, given the lack of empirical data on regular users at this point. See Table 4.6 for the summary of our evaluation.

Myphrase UDS Ratings Explanation.

We use *Quasi* to refer to "almost full benefit" and *Partially* to indicate "partial benefit only." We rate Myphrase as *Quasi-Memorywise-Effortless*/U1: users must remember at least one secret; *Scalable-for-Users*/U2: site-specific passwords are generated from the master passphrase; *Quasi-Nothing-to-Carry*/U3: having the dictionary aids usability (less typing and less error in typing), but passwords can be generated from memorized passphrases; we do not grant *Physically-Effortless*/U4 since, if the dictionary is unavailable, the user is likely to type more characters (for most passphrases) than a regular password; Myphrase is *Partially-Efficient-to-Use*/U6: typing the same passphrase should become easier with repeated use, however, un-

[4]www.lastpass.com/

	Usability								Deployability						Security										
	U1: Memorywise-Effortless	U2: Scalable-for-Users	U3: Nothing-to-Carry	U4: Physically-Effortless	U5: Easy-to-Learn	U6: Efficient-to-Use	U7: Infrequent-Errors	U8: Easy-Recovery-from-Loss	D1: Accessible	D2: Negligible-Cost-per-User	D3: Server-Compatible	D4: Browser-Compatible	D5: Mature	D6: Non-Proprietary	S1: Resilient-to-Physical-Observation	S2: Resilient-to-Targeted-Impersonation	S3: Resilient-to-Throttled-Guessing	S4: Resilient-to-Unthrottled-Guessing	S5: Resilient-to-Internal-Observation	S6: Resilient-to-Leaks-from-Other-Verifiers	S7: Resilient-to-Phishing	S8: Resilient-to-Theft	S9: No-Trusted-Third-Party	S10: Requiring-Explicit-Consent	S11: Unlinkable
Text passwords			●		●	●	◐	●	●	●	●	●	●	●		◐						●	●	●	●
Firefox	◐	●	◐	◐	●	●	●		●	●	●		●	●	◐	◐					●	●	●	●	●
LastPass	◐	●	◐	◐	●	●	●	◐	●	◐	●		●		◐	◐	◐	◐		◐	●	●		●	●
Myphrase	◐	●	◐		●	○	◐	○	●	●	●	○		●		●	●	◐		●	●	●	●	●	●
Fastwords [47]			●		●	○	◐	●	●	●	○					◐						●	●	●	●
Topkara [95]	◐	●			●	○	◐		◐	●	●	●		●		●	●	◐		●		●	●	●	●
Cheswick [16]			◐		●	○	◐	●	●	●	○			●		●	●	●				●	●	●	●

Table 4.6: *UDS evaluation of Myphrase. Key: ● (offers the benefit); ◐ (almost offers the benefit); ○ (offers partial benefit); blank (benefit not offered).*

like a traditional password manager, the master passphrase must be entered for each authentication; *Quasi-Infrequent-Errors*/U7: use of the same passphrase and auto-fill words from a drop-down menu may result in less typing errors; *Partially-Easy-Recovery-from-Loss*/U8: losing the master secret requires re-setting all site passwords, although the user can browse their dictionary in an attempt to jog their memory; *Partially-Browser-Compatible*/D4: the web interface can be used when software tools are unavailable; *Non-Proprietary*/D6: no known patents as we are aware of. Myphrase is not *Resilient-to-Physical-Observation*/S1: all password input techniques are vulnerable to physical observation, and inspecting the user's auto-complete selections may allow an adversary to learn the master secret more easily; is *Resilient-*

to-Targeted-Impersonation/S2: words in the passphrase are chosen randomly—so having access to a user's preference to certain words or even the user dictionary will not help in targeted guessing; *Resilient-to-Throttled-Guessing*/S3 and *Quasi-Resilient-to-Unthrottled-Guessing*/S4: assuming at least 60 bits of entropy, the generated site-specific password is resilient against online guessing and to some extent, against offline attacks; *Resilient-to-Leaks-from-Other-Verifiers*/S6 and *Resilient-to-Phishing*/S7: each password is site-specific (i.e., salted by the site's domain) and retrieving the master passphrase from a compromised password requires non-trivial computation power (on average, 2^{59} password trials in the default setting; additionally each trial needs 2^{15} iterations of a hash function). This feature also restricts malicious sites from replaying a user's password to get access to another web account of the user (perhaps to a more valuable account).

UDS comparison with Fastwords. We include Fastwords [47] as an example of multi-word passwords where words are user-chosen. We rate it as not offering U1 and U2: users are required to remember several phrases (similar to regular passwords). Fastwords are independent of any particular dictionary (except the widely-available built-in English dictionary in current mobile platforms); we rate it to offer U3. We rate it as offering *Quasi*-D3: it may require server-side changes as many websites currently disallow the space character in passwords. Fastwords do not offer D6: as mentioned at fastword.me, the technology is patent-pending. Fastwords' security features are rated similar to text passwords; we believe it is unlikely to achieve significant improvement in this area, as long as user-choice is involved (cf. [11]).

UDS comparison with Topkara's scheme. The Topkara [95] scheme uses NLP-generated master passphrases for use as mnemonics. Their scheme also constructs site-specific passwords, using a pen-and-paper table-lookup. We rate it *Quasi*-U1: since there is at least one secret to memorize; we grant U2: the cognitive load does not become more difficult for additional accounts; we do not grant U3 since the user must carry the helper-cards, and additional accounts will require more cards; we do not grant U4 since the user must perform a table-lookup and type in their site-password; we rate *Quasi*-U6: the table-lookup and helper card construction require a non-negligible amount of time; we do not grant U8 since losing the helper cards will require resetting all site passwords; *Quasi*-D1: the scheme requires sight (although Braille helper cards may mitigate this issue); we grant S2: card lookups are generated from a random seed; *Quasi*-S4: the resulting site-passwords are at least 8 characters long, including digits and special characters, which should exhibit $log_2(94^8) = 52$ bits of entropy at minimum; we grant S8: access to the card does not give an adversary any significant advantage.

UDS comparison with Cheswick's scheme. Cheswick's scheme [16] generates a passphrase by randomly selecting words from a fixed dictionary. We rate it as not offering U1 and U2 since, like Fastwords, users must memorize several phrases; *Quasi*-U3: similar to Myphrase, the fixed dictionary helps easy recall and typing; *Quasi*-D3: server-side changes may be necessary to handle the space character.

4.5.4 Memorability of Myphrase Passphrases

We believe the memorability of a Myphrase passphrase is enhanced by the following factors. (a) Frequent repetition: repeated use reduces the user's working memory load from potentially dozens of site passwords to one. (b) Semantic and syntactic structures: words are apparently more memorable than random character strings, and sentences more memorable than random sequences of words. (*c*) Personally meaningful words: familiarity with the passphrase components should make recall easier. (*d*) Recognition of words: the auto-complete suggestions allow the user to recognize their passphrase words from a list.

Our hypothesis is that Myphrase passphrases may retain security advantages of random phrases without being too difficult to remember. We would like to test this hypothesis through a user study in the future. Below we discuss some related work supporting the idea that personally meaningful words could be more memorable.

An fMRI study performed by Saykin et al. [82] shows that brain activity is much greater, and takes place in more regions, when a subject is shown a familiar word as compared to an unaccustomed word. Gregg [35] also suggests that recall of common words is higher than uncommon words. (Note that frequency ranked words in Myphrase are by definition "common" to the user.) Hulme et al. [42] performed two experiments: in the first, non-word sequences were found to be less memorable than words. The second experiment compared memorability of Italian and English words on English-speaking participants. Users remembered English words better, but memory span for Italian words increased after learning the English translations. This also supports the idea that a user will better remember words for which their semantics are well known.

The auto-suggest feature of Myphrase reduces typing and may also jog the user's memory (i.e., the task of recall is partially reduced to recognition). However, it is unclear to what extent this will help for overall memorization of the user's passphrase, including sequence. Early research by Gregg [35] suggests that recognition of words is actually higher for uncommon words. In a recent recognition study [108], in the context of text based passwords, it was found that recognition of words was no better than free-recall.

Our scheme can be thought of as a type of cued-recall in which the user must recall the first letter of the word, and then a cue is given (in this case a list of words, or the word itself). It is usually agreed upon that cued-recall is much easier than free-recall (e.g., [98]).

The connected discourse variant also leverages the semantic and syntactic structure of a sentence. Previous research (e.g., [63, 25, 60, 29, 61]) suggests that such high-order patterns are more memorable than random sequences of words. A recent study on the memorability of random passphrases performed by Shay et al. [87] contradicts this notion, and indicates that there is little difference between recollection of random passwords versus passphrases. However, in this study (also in [108]) personally meaningful words were not used. Furthermore, users were not provided any recognition cues, and did not benefit from the repeated use of the phrase (one Myphrase passphrase is expected to be used repeatedly for all or most web logins, reducing the memory load and combating the the multiple password interference problem [17]).

Words in a Myphrase passphrase should be more familiar to users; however, users must memorize the sequence of these words—e.g., six words from a 1024-word dictionary. Memorability of a phrase of such length appears to be feasible (albeit non-trivial), especially when this phrase can potentially replace all site passwords for a user.

4.5.5 Limitations

Major limitations include: ***(a)*** Myphrase's approach of using a master secret is similar to several existing techniques—expecting that users will remember one strong secret and derive all other site passwords from it. However, users most likely would not change all their existing passwords to Myphrase at the outset (cf. [7]). Therefore, the Myphrase passphrase would be "one more secret" to remember and will benefit users only in the long-run. However, users can gradually migrate their accounts under a Myphrase password, and keep using regular text passwords along with Myphrase. ***(b)*** A forgotten or compromised (e.g., via PC malware) Myphrase master passphrase will incur selecting a new passphrase and resetting all site passwords—a major inconvenience for users. Users may write down the passphrase and store it in a place not accessible to others (cf. [111, 40]). Users may browse the dictionary to attempt to recognize the forgotten words in the passphrase. ***(c)*** The dictionary may be lost or unavailable to users (e.g., when using a new device). In such cases, typing errors may increase as users must recall the exact words in the phrase without any cue. Posting the dictionary to a public or semi-public website (e.g., Facebook) may enable access-from-anywhere. The dictionary may also be re-created from the original sources used. ***(d)*** The Myphrase tool is required to convert the master passphrase to a site password. When the tool is unavailable (e.g., in a friend's device), a website for this conversion is available at: http://users.encs.concordia.ca/~a_skil/myphrase/myp-web/. The web tool uses locally-executed JavaScript, and does not interface with any 3rd party web services. The user may even download the script and execute it locally (offline or even in a virtual machine sandbox). The user enters their passphrase and the URL of the site they wish to log into. The site password is then generated, and the user can copy–paste the site password into the login field of the web service. ***(e)*** We anticipate the use of Myphrase may increase the login time (e.g., we observed the time to be close to 20 seconds for a 8-word phrase from our own experience on a PC–a formal user study will result in a more accurate estimation). We expect that the repeated use of the same passphrase may reduce the login time in the long-run.

4.6 Concluding Remarks

Myphrase takes advantage of the already existing tendency towards choosing familiar words as passwords. Users are generally frowned upon by security advocates for making such choices, as these words can easily be subjected to dictionary attacks. In contrast, Myphrase allows users to generate stronger passwords from a dictionary of words they are familiar with or use in their daily communications. In the context of FDE, regular user-chosen passwords do not provide adequate protection. For PDE-enabled storage, a weak password can have dire consequences. Offline guessing is easily performed on an encrypted volume image or partial snapshot. Myphrase is much more suitable for protecting encryption keys, given the relatively high entropy attained by the master passphrase and context-specific passwords.

As discussed, Myphrase has several potential limitations, e.g., longer login times, and memorizing a sequence of several words as the master passphrase. However, the use of personal words may help user-acceptance[5] — a major obstacle for any new password scheme. The auto-suggest feature reduces typing, which may also make Myphrase more suitable for mobile devices than regular passwords. However, we

[5]See e.g., the user study [8] of object-based password: users browse their personal images/music files as part of the login mechanism; some users reportedly *enjoyed* interacting with such objects.

would like to emphasize that no formal user-testing has been conducted yet. We introduce Myphrase here to promote discussion on authentication schemes that can sustain site-password leakage and are suitable for both desktop and mobile platforms. Our prototype implementation is available at:

http://users.encs.concordia.ca/~a_skil/myphrase/.

Chapter 5

Conclusions and Future Work

In certain situations, users require a level of protection beyond the *semantic security* that is offered by encryption. Deniable encryption techniques can be used to augment standard encryption, to contend with a coercive adversary. This dissertation examined the feasibility and efficacy of deniable storage encryption for mobile devices. The *Mobiflage* tool was designed and prototyped to assess the effective security and usability of the mobile deniable storage concept. The results are promising, as Mobiflage addresses several leakage vectors while incurring a tolerable impact on performance and usability. The implementation relies on a conscientious user that will adhere to usage guidelines devised to prevent leakage or compromise through inappropriate behaviour. One such directive is to choose a high entropy password to protect the volume encryption keys. This dissertation also discussed a new password scheme for that specific purpose.

The *Myphrase* design aims to facilitate passwords that are suitably strong for protection of encryption keys, easier to input on mobile devices, and alleviate the memory burden on the user. By building on research in cognitive psychology, Myphrase passwords are constructed to offer better memorability and security: familiar words and structures are randomly combined to create user-specific multi-word passphrases. The Myphrase scheme was designed to generate passwords with at least 60 bits of entropy, plus 15 bits through PBKDF2. While this falls short of the current 80-bit infeasibility boundary, it still provides at least twice the average complexity of current user-chosen passwords. Myphrase was implemented for PCs and mobile devices. The Android implementation is suitable for use with the Mobiflage pre-boot authenticator, demonstrating the utility for protection of encryption keys on mobile devices.

5.1 Future Research Goals

The research on deniable storage encryption and key-protection passwords has identified additional avenues that should be explored:

1. Flash storage wear-levelling – Without access to the raw flash storage cells (as with SD and eMMC), leakage through partial snapshots cannot be prevented. It would be prudent to determine how easily LBAs can be correlated to hidden volume data. This may inhibit the effective security of PDE on mobile devices.

2. Generalized deniable storage encryption – Newer Android devices do not contain SD cards, precluding the current Mobiflage implementation. Likewise, other platforms (e.g., iOS) use the MTP protocol and share internal/external storage. The Mobiflage design can be adapted, and implemented for use with MTP devices to provide a general solution for mobile devices.

3. Usability study of Myphrase – The memorability of Myphrase remains conjecture at this point. There is strong support from existing research in cognitive psychology, however, until a formal user study is conducted, the memorability aspects cannot be confirmed.

Bibliography

[1] Ross Anderson, Roger Needham, and Adi 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] Julian Assange, Ralf-Philipp Weinmann, and Suelette Dreyfus. Rubberhose: Cryptographically deniable transparent disk encryption system. Project website: http://marutukku.org/.

[4] Mikhail J. Atallah, Craig J. McDonough, Victor Raskin, and Sergei Nirenburg. Natural language processing for information assurance and security: an overview and implementations. In *NSPW'00*, Ballycotton, County Cork, Ireland, 2000.

[5] Ben F. Barton and Marthalee S. Barton. User-friendly password methods for computer-mediated information systems. *Computers and Security*, 3(3):186–195, August 1984.

[6] BBC News. LinkedIn passwords leaked by hackers. News article (June 7, 2012). http://www.bbc.co.uk/news/technology-18338956.

[7] Kemal Bicakci, Nart Bedin Atalay, Mustafa Yuceel, and Paul C. van Oorschot. Exploration and field study of a browser-based password manager using icon-

based passwords. In *Workshop on Real-Life Cryptographic Protocols and Standardization*, March 2011.

[8] Robert Biddle, Mohammad Mannan, Paul C. van Oorschot, and Tara Whalen. User study, analysis, and usable security of passwords based on digital objects. *IEEE TIFS*, 6(3):970–979, September 2011.

[9] Joseph Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *IEEE Symp. on Security and Privacy*, May 2012.

[10] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symp. on Security and Privacy*, May 2012.

[11] Joseph Bonneau and Ekaterina Shutova. Linguistic properties of multi-word passphrases. In *Workshop on Usable Security (USEC'12)*, Bonaire, Netherlands, March 2012.

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

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

[14] William Burr, Donna Dodson, and W. Polk. Electronic authentication guidelines (NIST SP 800-63), April 2006.

[15] Ran Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In *CRYPTO'97*, Santa Barbara, CA, USA, 1997.

[16] William Cheswick. Rethinking passwords. Invited talk at USENIX LISA 2010. http://www.usenix.org/event/lisa10/tech/slides/cheswick.pdf. See summary in ;login: The USENIX Magazine, 36(2):68-69, Apr. 2011.

[17] Sonia Chiasson, Alain Forget, Elizabeth Stobert, Paul C. van Oorschot, and Robert Biddle. Multiple password interference in text and click-based graphical passwords. In *ACM CCS'09*, Chicago, IL, USA, November 2009.

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

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

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

[21] Alexei Czeskis, David J. St. Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce 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.

[22] 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).

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

[24] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[25] D. James Dooling and Roy Lachman. Effects of comprehension on retention of prose. *Journal of Experimental Psychology*, 88(2):216 – 222, 1971.

[26] Radu Dragusin. Data breach at IEEE.org: 100k plaintext passwords. Online article (Sept. 18, 2012). http://ieeelog.com/.

[27] Markus Dürmuth and David Freeman. Deniable encryption with negligible detection probability: An interactive construction. In *Eurocrypt*, Tallinn, Estonia, 2011.

[28] 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.

[29] William Epstein. The influence of syntactical structure on learning. *American journal of psychology*, 74(1):80 – 85, 1961.

[30] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications.* Wiley Publishing, March 2010.

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

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

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

[34] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences (JCSS)*, 28(2):270–299, April 1984.

[35] Vernon H. Gregg. *Recall and Recognition*, chapter Word Frequency, Recognition, and Recall. John Wiley & Sons, Inc., 1976.

[36] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, San Jose, CA, USA, 2008.

[37] J. Alex Halderman, Brent Waters, and Edward W. Felten. A convenient method for securely managing passwords. In *Conference on World Wide Web (WWW'05)*, May 2005.

[38] Jin Han, Meng Pan, Debin Gao, and HweeHwa Pang. A multi-user steganographic file system on untrusted shared storage. In *Annual Computer Security Applications Conference (ACSAC'10)*, Orlando, Florida, USA, 2010.

[39] Cormac Herley. So long, and no thanks for the externalities: The rational rejection of security advice by users. In *NSPW'09*, Oxford, UK, September 2009.

[40] Cormac Herley and Paul C. van Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 10(1):28–36, 2012.

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

[42] Charles Hulme, Sarah Maughan, and Gordon D.A Brown. Memory for familiar and unfamiliar words: Evidence for a long-term memory contribution to short-term memory span. *Journal of Memory and Language*, 30(6):685–701, 1991.

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

[44] Imperva.com. Consumer password worst practices. Imperva white paper on Rockyou.com's 32 million leaked passwords (Jan. 2010). http://www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf.

[45] Philip Inglesant and M. Angela Sasse. The true cost of unusable password policies: Password use in the wild. In *CHI'10*, Atlanta, GA, USA, April 2010.

[46] 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/.

[47] Markus Jakobsson and Ruj Akavipat. Rethinking passwords to adapt to constrained keyboards. In *Mobile Security Technologies (MoST) Workshop*, May 2012.

[48] Markus Jakobsson, Elaine Shi, Philippe Golle, and Richard Chow. Implicit authentication for mobile devices. In *USENIX HotSec'09*, Montreal, Canada, August 2009.

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

[50] Sundararaman Jeyaraman and Umut Topkara. Have the cake and eat it too - infusing usability into text-password based authentication systems. In *ACSAC'05*, 2005.

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

[52] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. Secure applications of low-entropy keys. In *Information Security Workshop (ISW'97)*, Ishikawa, Japan, September 1998.

[53] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, 9, 1883.

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

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

[56] Marek Klonowski, Przemysław Kubiak, and Mirosław Kutyłowski. Practical deniable encryption. In *Theory and Practice of Computer Science (SOFSEM'08)*, Novy Smokovec, Slovakia, 2008.

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

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

[59] Cynthia Kuo, Sasha Romanosky, and Lorrie Faith Cranor. Human selection of mnemonic phrase-based passwords. In *SOUPS'06*, Pittsburgh, PA, USA, July 2006.

[60] Linda Lombardi and Mary C. Potter. The regeneration of syntax in short term memory. *Journal of Memory and Language*, 31(6):713 – 733, 1992.

[61] Lawrence E. Marks and George A. Miller. The role of semantic and syntactic constraints in the memorization of english sentences. *Journal of Verbal Learning and Verbal Behavior*, 3(1):1 – 5, 1964.

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

[63] George Miller. Human memory and the storage of information. *Information Theory, IRE Transactions on*, 2(3):129 –137, September 1956.

[64] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, March 1956.

[65] George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine Miller. WordNet: An on-line lexical database. *International Journal of Lexicography*, 3:235–244, 1990.

[66] George Edward Moore. Progress in digital integrated electronics. In *International Electron Devices Meeting, (IEDM 1975)*, Washington, DC, USA, 1975.

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

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

[69] Adam ONeill, Chris Peikert, and Brent Waters. Bi-deniable public-key encryption. In *CRYPTO'11*, Santa Barbara, CA, USA, 2011.

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

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

[72] Sigmund N. Porter. A password extension for improved human factors. *Computers and Security*, 1(1):54–56, January 1982.

[73] Roger Price and Leonard Stern. *The Original Mad Libs 1.* Price Stern Sloan, February 1974.

[74] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *USENIX Security Symposium*, Bellevue, WA, USA, 2012.

[75] Rede Globo. Not even FBI was able to decrypt files on Daniel Dantas. News article (Jun. 25, 2010). http://g1.globo.com/English/noticia/2010/06/not-even-fbi-can-de-crypt-files-daniel-dantas.html.

[76] Arnold Reinhold. Diceware passphrase. http://world.std.com/~reinhold/diceware.html.

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

[78] Oriana Riva, Chuan Qin, Karin Strauss, and Dimitrios Lymberopoulos. Progressive authentication: deciding when to authenticate on mobile phones. In *USENIX Security Symposium*, Bellevue, WA, USA, 2012.

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

[80] Blake Ross, Collin Jackson, Nicholas Miyake, Dan Boneh, and John C. Mitchell. Stronger password authentication using browser extensions. In *USENIX Security Symposium*, 2005.

[81] Markku-Juhani Saarinen. Encrypted watermarks and Linux laptop security. In *International Workshop on Information Security Applications (WISA'05)*, Jeju Island, Korea, 2005.

[82] Andrew J. Saykin, Sterling C. Johnson, Laura A. Flashman, Thomas W. McAllister, Molly Sparling, Terrance M. Darcey, Chad H. Moritz, Stephen J. Guerin, John Weaver, and Alexander Mamourian. Functional differentiation of medial temporal and frontal regions involved in processing novel and familiar words: an fMRI study. *Brain*, 122(10):1963–1971, 1999.

[83] J. Schaad and R. Housley. Advanced Encryption Standard (AES) key wrap algorithm, September 2002. IETF RFC 3394.

[84] Stuart Schechter, Cormac Herley, and Michael Mitzenmacher. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *HotSec'10*.

[85] Norbert Schmitz. Improved guessing of composite passwords. Master's thesis, Ruhr University Bochum, March 2012.

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

[87] Richard Shay, Patrick Gage Kelley, Saranga Komanduri, Michelle L. Mazurek, Blase Ur, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Correct horse battery staple: exploring the usability of system-assigned passphrases. In *SOUPS'12*, Washington, DC, USA, 2012.

[88] SkullSecurity. Leaked passwords (database). http://www.skullsecurity.org/wiki/index.php/Passwords.

[89] Sidney L. Smith. Authenticating users by word association. *Computers and Security*, 6(6):464–470, 1987.

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

[91] Techcrunch. 370 passwords you shouldn't (and can't) use on Twitter. Dec. 27, 2009. http://techcrunch.com/2009/12/27/twitter-banned-passwords/.

[92] telega and TechnalXS. Simple Mail: Mail client (POP3/IMAP/SMTP) for Firefox. https://addons.mozilla.org/en-us/firefox/addon/simple-mail/.

[93] 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/.

[94] 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/.

[95] Umut Topkara, Mikhail J. Atallah, and Mercan Topkara. Passwords decay, words endure: secure and re-usable multiple password mnemonics. In *ACM Symposium on Applied computing (SAC'07)*, Seoul, Korea, 2007.

[96] 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.

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

[98] Endel Tulving and Zena Pearlstone. Availability versus accessibility of information in memory for words. *Journal of Verbal Learning and Verbal Behavior*, 5(4):381 – 391, 1966.

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

[100] Brian Wallace. Transferable state attack on iterated hashing functions. Tech. document (July 2012). https://firebwall.com/research/TransferableStateAttackonIteratedHashingFunctions.pdf.

[101] Percy Wegmann. jspos - Javascript part of speech tagger. Javascript port of Mark Watson's FastTag. https://code.google.com/p/jspos/.

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

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

[104] Charles Matthew Weir. *Using probabilistic techniques to aid in password cracking attacks.* PhD thesis, Florida State University, Tallahassee, FL, USA, March 2010.

[105] Matt Weir, Sudhir Aggarwal, Michael Collins, and Henry Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *ACM CCS'10*.

[106] Matt Weir, Sudhir Aggarwal, Breno de Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2009.

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

[108] Nicholas Wright, Andrew S. Patrick, and Robert Biddle. Do you see your password? applying recognition to textual passwords. In *SOUPS'12*, Washington, DC, USA, 2012.

[109] Jeff Yan, Alan Blackwell, Ross Anderson, and Alasdair Grant. Password memorability and security: Empirical results. *IEEE Security & Privacy*, 2(5), 2004.

[110] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, May 2009.

[111] ZDNet. Microsoft: Write down your passwords. News article (May 23, 2005). http://www.zdnet.com/microsoft-write-down-your-passwords-1139193117/.

[112] Xuan Zhou, HweeHwa Pang, and Kian-Lee Tan. Hiding data accesses in steganographic file system. In *International Conference on Data Engineering (ICDE'03)*, Bangalore, India, 2003.

Appendix A

Crypto Primitives and Deniability

Certain crypto primitives, such as ciphers and pseudorandom number generators (PRNGs), may leak information that can compromise deniability. For example, attacks have been discovered against the CBC mode of operation, when applied to disk encryption (e.g., [32]). This chapter gives details about possible attacks against CBC, when used for data-at-rest, that may compromise deniability. These attacks justify the use of XTS-AES in Mobiflage (the default Android encryption implementation uses the CBC mode).

Plaintext difference attack. A plaintext difference attack has been demonstrated (e.g., by Fruhwirth [32]). If any two ciphertext blocks within a volume sector, $C_m = E_k(P_m \oplus C_{m-1})$ and $C_n = E_k(P_n \oplus C_{n-1})$, are identical, the adversary can learn the difference between the original plaintexts. The preceding blocks, C_{m-1} and C_{n-1}, are also known to the adversary since they exist in the sector immediately preceding the blocks in question. Since $C_m = C_n$, it holds that:

$$P_m \oplus C_{m-1} = P_n \oplus C_{n-1}$$

Which may be expressed as:

$$C_{m-1} \oplus C_{n-1} = P_n \oplus P_m$$

The adversary can now deduce the difference in plaintexts. In the case where either P_n or P_m is zero, the adversary can recover the original plaintext of the other block. In Mobiflage, the storage is filled with random bytes, however some regions may have been zeroed (e.g., the inode table and block bitmaps in an Ext4 block group are zeroed during formatting). This may allow the adversary to reveal fragments of hidden data if the CBC mode is used.

This is mostly a theoretical attack, as the probability of two identical blocks in a sector is very low: treating the cipher output as a random distribution (a relaxed assumption), for the AES block size of 128 bits, and a disk sector size of 512 Bytes, then

$$\begin{aligned} P(\text{two identical blocks in a sector}) &= \binom{32}{2} / ((2^{128})^{32}) \\ &\approx 496/1.0 \times 10^{1233} \\ &\approx 2.1 \times 10^{-1230} \end{aligned}$$

Furthermore, assuming an equally likely distribution of ciphertext blocks, the probability that any given block in a sector will be all zeros is:

$$P(\text{one block in a sector is all zeros}) = 32/(2^{128}) = 9.4039548 \times 10^{-38}$$

For a per-sector probability of 1.98×10^{-1269} that the attack conditions will be met. There are 2097152 sectors per GB, so even for a 64GB disk there is only a negligible probability that suitable attack conditions will exist:

$$\frac{1.34 \times 10^{8}}{1.98 \times 10^{-1269}}$$

Watermarking attack. Under certain schemes, the sector IVs are predictable (e.g., the index of the disk sector). This gives rise to a chosen plaintext attack known as *watermarking*, as demonstrated by the authors of [32, 81]. Watermarking manifests itself as a data existence leak, and can defeat deniable storage encryption. The attack is mounted by encrypting a special plaintext file to produce two sectors where the first ciphertext block of each sector is identical:

$$IV_1 \oplus P_1 = IV_2 \oplus P_2$$

Equivalently:

$$IV_1 \oplus IV_2 = P_1 \oplus P_2$$

This pattern, or watermark, can be detected in the encrypted data even when the key and cipher are unknown. An adversary may trick a user into storing such a file in their hidden volume, then seize the device and search for the watermark, proving the existence of the hidden volume.

***Copy-and-paste* attack.** An adversary can move pairs of ciphertext blocks from one area of the disk to another. Since the decryption of a plaintext block P_i depends only on the ciphertext blocks C_i and C_{i-1}, the block will decrypt properly no matter where it exists on the disk. This has serious implications for deniability: a pair of ciphertext blocks can be moved outside of the deniable region of a volume. The block will decrypt to unintelligible plaintext since a different key is being used. Unless empty regions of the disk were filled with random data before encryption, the adversary would expect zero sectors or old file fragments, proving the existence of hidden data and additional keys. Most PDE storage schemes, including Mobiflage, fill vacant storage with random bytes which should prevent the discovery of a hidden volume. In addition, Mobiflage uses the XEX tweaked-codebook with ciphertext stealing (XTS) mode which is not susceptible to the copy-and-paste attack, since the tweak operation depends on a block's sector index and will produce a different result when moved.

Tweakable block ciphers. Special *tweakable* cipher modes, such as LRW and XTS, were created specifically for disk encryption, to prevent or mitigate known attacks. IEEE Std 1619-2007 [43] defines the XTS-AES mode of operation. It has been approved by NIST [68] as the preferred disk encryption block cipher. XTS mode is a code book mode (i.e., no block chaining) which uses a secondary *tweak* key to make unpredictable use of the disk sector index. The IEEE standard rates the security of the system as equivalent to using ECB mode with a different key for each block. It is important to note that the security of XTS rests on the security of AES. The XTS-AES cipher mode works as follows:

$$C_i = E_{K1}(P_i \oplus (E_{K2}(n) \otimes a^i)) \oplus (E_{K2}(n) \otimes a^i) \tag{A.1}$$

Where:

$\oplus$ is the exclusive OR operation,

$\otimes$ is multiplication over the finite field GF(2^{128}) modulo $x^{128} + x^7 + x^2 + x + 1$,

$K1$ is the AES encryption key,

$K2$ is the tweak key,

i is the cipher block index within a 512-byte sector,

n is the sector index on the disk, and

a is a primitive element of Galois Field (2^{128}) that corresponds to the polynomial x.

The IEEE standard does warn against encryption of more than a few hundred terabytes with the same key. This may introduce the possibility of certain chosen ciphertext attacks discussed in the standard. The attacks are not unique to XTS. They are the result of the 128-bit AES block size and the birthday paradox (i.e., use of a larger key space will not mitigate these attacks). This is not a problem for modern mobile devices which have relatively limited storage capacity (e.g., on the order of 64GB).

Appendix B

Attacking Android Encryption

In this chapter we discuss a dictionary attack against the default Android encryption scheme. As a result of the implementation choices in the Android security model, this brute-force attack, if successful, will also recover the device-unlock secret.

The Android FDE subsystem reuses the screen-unlock secret, to protect the encryption key, with 2000 iterations of PBKDF2. It is the user's responsibility to choose a suitably strong password, to prevent a dictionary attack. The pre-boot authentication prompt will time-out for 30 seconds after 10 failed password attempts. In order to mount an online attack, attempting all 4 and 5 digit PIN codes (110,000 total), the device will be unresponsive for almost 3 days and 20 hours. The actual attack will take longer, as there is a non-negligible delay to test each PIN with PBKDF2. Since the adversary can, on average, obtain the correct PIN after exhausting 50% of the search space, this online attack is feasible. If the user chooses a longer PIN however, and the adversary has to attempt all possible PINs between 4 and 9 digits (1,111,110,000 total), the device would be timed-out for almost 106 years.

An offline attack against the encryption system is also possible, if the adversary can obtain an image of the device's encrypted storage. Physical storage acquisition techniques, such as JTAG and "chip-off", can be used to obtain such an image (see e.g., Hoog [41, pp. 266–284]). Logical acquisition techniques, such as installing a custom recovery image with root privileges, may also be possible.

The PBKDF2-protected volume key is contained in the encrypted volume's footer. A brute-force attack can be mounted to attempt to decrypt the volume key. When the correct screen-unlock secret is found, the resulting volume key will decrypt a valid filesystem. A tool, `droidcrack`, was created to test the feasibility of such a dictionary attack against an encrypted Android disk image (source code available at http://users.encs.concordia.ca/~a_skil/droidcrack/). The experiments were conducted on a single core of a 3.4GHz Intel Core i7-2600, running Ubuntu 12.04 with the OpenSSL 1.0.0 library. To perform the offline attack against all 4 and 5 digit PINs required 9 minutes and 2 seconds. To test all PINs between 4 and 9 digits required 63 days 7 hours and 35 minutes. On average the correct PIN would be identified after one month, making this attack feasible, even on commodity hardware. These experiments demonstrate that weak passwords cannot be relied upon to protect encryption keys. The eight-character lower-case search space (26^8) would require 32.6 years on this hardware, but may be feasible with parallelization. For comparison, the search space for passwords generated by Myphrase is, by default, 62^{12} and would require 1.0×10^{14} years to exhaust on this hardware. If the Myphrase dictionary is available to the adversary, then the search space is reduced to 2^{60} and would require 1.8×10^{8} years. Both of these scenarios can be considered infeasible. Increasing the PBKDF2 iteration count, as discussed in Section 3.7, can slow a brute-force attack, but is insufficient alone. A strong password is necessary to ensure an exhaustive search is infeasible.