MobiPluto: File System Friendly Deniable Storage for Mobile Devices

Bing Chang[1,2,3], Zhan Wang[1,2]* , Bo Chen[4], Fengwei Zhang[5]
[1] State Key Laboratory of Information Security,
Institute of Information Engineering, Chinese Academy of Sciences, China
[2] Data Assurance and Communication Security Research Center,
Chinese Academy of Sciences, China
[3] University of Chinese Academy of Sciences, China
[4] College of Information Sciences and Technology, The Pennsylvania State University, USA
[5] Department of Computer Science, Wayne State University, USA
{changbing,wangzhan}@iie.ac.cn, bxc30@psu.edu, fengwei@wayne.edu

ABSTRACT

Mobile devices are prevalently used for processing personal private data and sometimes collecting evidence of social injustice or political oppression. The device owners may always feel reluctant to expose this type of data to undesired observers or inspectors. This usually can be achieved by encryption. However, the traditional encryption may not work when an adversary is able to coerce device owners into revealing their encrypted content. Plausibly Deniable Encryption (PDE) is thus designed to protect sensitive data against this type of powerful adversaries.

In this paper, we present *MobiPluto*, a file system friendly PDE scheme for denying the existence of sensitive data stored on mobile devices. MobiPluto achieves deniability feature as nothing but a "side-effect" of combining thin provisioning, a well-established tool in Linux kernel, with encryption. This feature makes MobiPluto more plausible for users to have such software on their mobile devices. A salient difference between MobiPluto and the existing PDE schemes is that MobiPluto is "file system friendly", i.e., any block-based file systems can be deployed on top of it. Thus, it is possible to deploy MobiPluto on most mobile devices. We provide a proof-of-concept implementation for MobiPluto in an Android phone to assess its feasibility and performance.

Keywords

Plausibly Deniable Encryption, Mobile, Thin Provisioning, LVM

1. INTRODUCTION

Mobile devices are frequently used for processing private data and sometimes collecting evidence of social injustice or political oppression. The owner of a mobile device may be reluctant to expose this type of data to undesired observers or inspectors. With today's fast-paced, multitasking lifestyle, it is possible to leave our phones lying on table with an unlocked screen incautiously [26]. In certain geopolitical areas with tensions, the border inspector may compulsively require the passengers to reveal the content on their mobile devices. This could prove detrimental and may compromise security of particular professionals such as human rights activists, who may possess evidence of violence.

Plausibly Deniable Encryption (PDE) has been adopted to protect sensitive data against powerful adversaries who can coerce users into revealing their encrypted content. This practice should not be confused with encryption, as regular encryption is overt, while PDE is covert. A variety of PDE systems have been published for PC platform, including Rubberhose [24], TrueCrypt [35], etc. StegFS [28] is a PDE solution for Ext2 file system, but its drawbacks include: (1) it is space costly for resolving overwritten issue by using multiple copies; (2) the existence of the modified Ext2 driver and the external block table may lead to compromise of deniability. Ragnarsson et al. [31] was the first to mention taking advantage of thin provisioning to achieve deniability on PC's and inspired our work. However, their proposed design requires significant modification of thin provisioning and fails to hide the metadata, thus the deniability cannot be fully realized.

As the first PDE system implemented for mobile devices, Mobiflage [33] requires a physical or emulated FAT32 SD card which is not necessarily present in some mobile devices. Recently, Mobiflage [15] is extended to support Ext4 file system by modifying the driver of Ext4. Although the extended Mobiflage no longer requires a physical or emulated FAT32 SD card, its modifications to the Ext4 driver may indicate the use of PDE and lead to compromise of deniability. MobiHydra [36] improves Mobiflage by adding support to multiple levels of deniability and mode switching without rebooting, but it also requires a physical or emulated FAT32 SD card.

All the prior solutions [15, 24, 28, 31, 33, 35, 36] are not suitable for mobile devices, due to their performance and storage costs, getting inadequate support by the mobile operating systems, or being forced to modify the associated file systems as a last resort. In this work, we propose *Mobi-*

*This author is the corresponding author.

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA
© 2015 ACM. ISBN 978-1-4503-3682-6/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2818000.2818046

Pluto, a file system friendly PDE solution, which can make the existence of sensitive data stored on mobile devices deniable. To achieve deniability, two types of volumes are used: a public volume for storing regular data, and one or multiple hidden volumes for storing sensitive data. The volumes are protected by different passwords and encrypted with associated master keys. The key features of MobiPluto include:

- **File system friendly.** As data hiding is achieved at the block level, any block-based file systems can be deployed on top of MobiPluto without modifications. To the best of our knowledge, no prior work can provide such a novel feature.

- **Deniability as a side-effect.** MobiPluto achieves deniability as nothing but a "side-effect" of combining thin provisioning with encryption. Note that thin provisioning has been a well-established tool in Linux kernel.

- **User-controlled deniability levels.** A user is able to control the number of deniability levels in the system. This can be achieved by choosing the number of hidden volumes during initialization.

- **Less storage cost.** Compared to the steganographic based schemes [16, 28, 29], MobiPluto does not require extra storage for solving the overwritten problem between public data and hidden data.

We provide a proof-of-concept implementation for Android 4.2.2 on LG Nexus 4 to assess the feasibility and performance of MobiPluto. We also discuss the best practices users should follow to restrict other known issues that may lead to compromise of deniability.

The rest of the paper is organized as follows: Section 2 presents the background. In Section 3, we discuss models and assumptions, including system model and adversarial model. In Section 4, we describe MobiPluto design. In Section 5, we discuss the implementation for Android. We present the evaluation results in Section 6, including security analysis and performance evaluation. In Section 7, we discuss our design. Section 8 presents related work and Section 9 is the conclusion.

2. BACKGROUND

2.1 Deniable Encryption

Plausibly deniable encryption (PDE) was first explored to maintain the privacy of communicated data against a coercive attacker, who can approach and coerce either the sender or the receiver into revealing the decryption keys [19]. When being applied to storage encryption, it allows a data owner to decrypt a ciphertext to a plausible and benign decoy plaintext when using a different key, such that the owner is able to deny the existence of the original sensitive data [33]. For achieving plausibility, it requires that [31], 1) the decoy plaintext can be normally found on a computer; 2) all the ciphertexts should be "accounted for", i.e., having a plausible explanation.

2.2 Full Disk Encryption

To protect sensitive data, it can either encrypt individual files/directories (i.e., file system level encryption) or encrypt the entire disk (i.e., full disk encryption). Compared to file system level encryption, full disk encryption (FDE) has several benefits: 1) it transparently encrypts almost everything including the swap space and temporary files, by which the users do not need to bother about which files to be encrypted; 2) it allows to immediately destruct data by simply destroying a small number of keys for FDE. Popular FDE tools include BitLocker [4] (for Windows) and FileVault [12] (for Mac OS). FDE on Android has been an option since Android 3.0 [21] and it is implemented with dm-crypt [7].

To deny the existence of sensitive data protected by FDE, the device owner can claim that he/she does not possess the secret keys any more (e.g., he/she has not used the device for a long time, and has forgotten the password), and is not able to decrypt the disk. This however, only provides very weak deniability as the device owner may not be able to prove the aforementioned statement [31].

2.3 Steganographic File Systems vs. Hidden Volumes

To build practical PDE systems, it typically relies on either steganography or hidden volumes. Multiple steganographic file systems [16, 28, 29] have been designed in the literature to hide data in regular file systems. However, all of them seem to hide deniable data among regular file data. This may result in data loss of hidden files as they may be overwritten by the regular file data. To mitigate the risk of data loss, they usually need to maintain a large amount of redundant data which will lead to inefficient use of disk space. The hidden-volume mechanism (e.g., TrueCrypt [35]) can mitigate the risk of data loss by intelligently placing all the deniable files toward the end of the disk. In this way, the redundant data required for data loss can be significantly reduced. Consequently, we choose to use hidden volumes to build MobiPluto.

The hidden-volume mechanism works as follows. The disk is initially filled with random data. Two volumes are created on the disk, a public volume and a hidden volume. The public volume is encrypted (i.e., FDE) by a decoy key and is placed on the entire disk. The hidden volume is encrypted by a hidden key and is placed towards the end of the disk, starting from a secret offset. Upon a coercive attack, the device owner can disclose the decoy key to the attacker, to deny the existence of the hidden volume, as the attacker cannot differentiate the encrypted hidden volume data and the initial randomness embedded on the disk. Note that when using the hidden-volume mechanism to achieve deniability, the data written to the public volume should be placed sequentially (or approximately sequentially), to reduce the risk of over-writing the sensitive data stored in the hidden volume. This explains why the PDE systems [33, 36] based on hidden volumes prefer to use FAT [11] file systems.

2.4 Thin Provisioning

To avoid any potential failures caused by inadequate storage, storage administrators usually need to plan ahead and install more storage capacity than required (i.e., "thick provisioning"). This thick provisioning usually leads to significant waste, as a lot of storage capacity may remain unused over time. Thin provisioning has been designed to optimize storage utilization by eliminating the need for installing unnecessary storage capacity. With thin provisioning, a storage administrator only allocates logical storage space to an application and the system will not release the physical storage capacity until it is actually required. This "on-demand" storage avoids pre-allocating physical storage capacity, eliminating the waste caused by unused capacity.

Thin provisioning has been implemented by the dm-thin-pool module, which works with two devices, a data device and a metadata device. The data device contains blocks of the various volumes, allocated sequentially from the beginning, while the metadata device contains the block mappings. The dm-thin module provides two device mapper targets, thin-pool and thin. The thin-pool target maps the data and the metadata device to a pool device, while the thin target maps this pool device to multiple thin volume devices.

Logical Volume Manager (LVM [1]) has gained popularity on Android for being able to flexibly handle internal and external storage [2, 5]. Thin provisioning is added to LVM and can provide much more flexible storage management. As mobile devices become more and more powerful and are expected to be equipped with more and more storage capacity, using thin provisioning to manage mobile storage will become extremely helpful and popular.

In this work, we aim to adapt thin provisioning to build "file system friendly" deniable storage for mobile devices, for the following reasons: 1) The dm-thin-pool module has been added to the kernel, and we can simply rely on the existing kernel features to build PDE systems for mobile devices; 2) A thin volume can be used to build any block-based file systems, and thin provisioning can transform the non-sequential allocation on the thin volume to sequential allocation on the underlying storage. This makes it possible to combine both thin provisioning and hidden volumes to build "file system friendly" PDE systems.

3. MODELS AND ASSUMPTIONS

3.1 System Model

We mainly consider mobile devices equipped with storage media that expose a block-based access interface. Such block-based storage media are used extensively as the internal storage for mobile devices nowadays [33], including eMMC [14], etc.

For mobile devices equipped with raw flash, MobiPluto cannot directly work, as raw flash does not expose a block interface due to its nature [23] (e.g., flash memory has a limited number of program-erase cycle, and cannot be overwritten before being erased, etc.).

3.2 Adversarial Model

We consider a computationally bounded adversary, who can fully control a mobile device after having captured the device's owner. The adversary can get root privilege of the device, and fully control over the device's internal and external storage, etc. Additionally, the adversary can coerce the device's owner to surrender keys, in order to decrypt the storage and obtain the sensitive data stored in the device. As mobile devices usually communicate with the external environment, the adversary may also collude with the wireless carrier or the ISP to collect the network activity logs of suspected devices.

However, we do not consider an adversary who can continuously monitor a suspicious device, and can stealthily take periodic snapshots of the device's storage. In other words, our adversary can only have "one-time", rather than "multiple-time" storage snapshots. This would be practical as the adversary usually can have access to a mobile device only after seizing the user [33].

3.3 Assumptions

Our MobiPluto relies on multiple assumptions, as summarized in the following:

- The adversary cannot capture a mobile device which is working in the PDE mode. Otherwise, it can trivially retrieve the sensitive data from the PDE mode.
- MobiPluto needs to be merged with Android code stream, such that the PDE capability is widespread, and the availability of PDE will not be a red flag.
- The adversary will know the design of MobiPluto. However, it does not have any knowledge on the keys and passwords for PDE mode as well as the offset of the hidden volume.
- The adversary will stop coercing the device's owner once it is convinced that the decryption keys have been revealed.
- For a mobile device that uses MobiPluto, we assume the mobile OS, the bootloader, as well as the firmware and the baseband OS are all malware-free (i.e., trusted). Especially in the PDE model, the user will not use the malicious apps controlled by the adversary.

Compared to Mobiflage [33] and MobiHydra [36], MobiPluto does not require any assumptions on the file systems, and is thus "file system friendly". This will be advantageous in practice, as it allows the deployment of any block-based file systems in MobiPluto PDE systems.

4. MOBIPLUTO DESIGN

In this section, we present the design of MobiPluto. Our MobiPluto provides a file system friendly PDE solution for mobile devices by utilizing hidden volumes and thin provisioning. MobiPluto is named after the *Helmet of Pluto*, which according to classic mythology, is capable of turning its wearer invisible [13].

4.1 Overview

MobiPluto is able to deny the existence of sensitive data by hiding volumes (storing sensitive data) in the empty space of the storage medium. For simplicity of presentation, we consider a simple case which has only two volumes: a public volume created for storing regular data, and a hidden volume created for storing sensitive data. The data stored in the hidden volume are those whose existence the owner wants to deny. If multi-level deniability is required, the number of hidden volumes can be varied accordingly [36]. By utilizing the interesting properties offered by thin provisioning (Sec. 2.4), we build thin logical volumes ("thin volumes" for short) to achieve a file system friendly deniable storage solution for mobile devices.

In MobiPluto, the public volume is protected by a decoy password and the hidden volume is protected by a hidden password. Specifically, we use a randomly generated decoy key to encrypt the public volume and use the decoy password to encrypt the decoy key, which will be stored in the encryption footer. We use a randomly generated hidden key to encrypt the hidden volume, and use the hidden password to encrypt the hidden key. The encrypted hidden key will be stored at a secret offset which is generated by the hidden password. To achieve "file system friendly" PDE, we create thin volumes in the public volume and the hidden volume respectively. Upon booting, the password will be used to decrypt the key and the decrypted key will be used to decrypt the corresponding volume. Specifically, if the owner provides the decoy password, the system will boot into public mode, in which the content of the public volume will be present. The hidden volume part of the storage looks no difference from random data. In this fashion, an ordinary observer will be convinced no sensitive data exists. To process sensitive data, the owner should use the hidden password and boot into PDE mode, in which the hidden volume will be located (i.e., the offset of the hidden volume can be generated based on the hidden password) and decrypted by the hidden key.

A more sophisticated adversary may try to examine the utilization of the entire storage space and its availability. Thus, the entire storage space should remain allocatable and usable in the public mode. This can be handled in MobiPluto, as thin provisioning allows dynamically scaling, i.e., we can specify the size of a thin volume that exceeds its allocated capacity.

Note that in the public mode, the system can allocate the entire storage and the storage area that contains the hidden volume just looks like random noise. The adversary cannot infer the existence of hidden volume without knowing the hidden key.

Compared to the existing deniable encryption schemes for mobile devices [15, 33, 36], MobiPluto has a salient advantage: it is file system friendly, i.e., any block-based file systems can be deployed on top of each thin volume. Thus, we can deploy MobiPluto on the internal storage of most mobile devices.

4.2 File System Friendly Deniability

Different file systems may use different allocation strategies (e.g., FAT32 prefers linear allocation [11] and Ext4 prefers random allocation [15]). FAT32 has the nature of supporting hidden-volume mechanism due to its concentrated metadata and sequential allocation. For a FAT32 formatted device, we can simply place the hidden volume somewhere in the second half of the storage medium, by which the data in the hidden volume will not be easily overwritten by the data in the public volume, as the public volume grows sequentially from the beginning of the storage medium. However, it is problematic to place hidden volumes in an Ext4 formatted device, because: Firstly, the data in the hidden volumes may overwrite all or part of the public volume's metadata. By observing this abnormal overwrite, the adversary may suspect the existence of hidden volumes; Secondly, the data from the public volume may easily overwrite the data in the hidden volumes. In general, if the device is formatted with a file system having similar characteristics like Ext4, we cannot simply create hidden volumes within it.

Thin provisioning can help address the aforementioned concern. By using thin provisioning, we can first create a thin pool and then create thin volumes in the thin pool. The thin pool ties together a small metadata device and a data device, and the latter occupies most of the storage space. Interestingly, the data device in thin provisioning is used linearly (i.e., the space maps allocate space linearly from front to back). To avoid fragmenting free space, the allocation always goes back and fills the gaps in the data device. In this way, thin provisioning can transform the possibly non-sequential allocation on the thin volume to sequential allocation on the data device. This renders it possible to combine hidden volumes and thin provisioning to achieve a PDE solution that allows any block-based file systems to be deployed.

Thin provisioning provides a logical block device (a thin volume), which allows to deploy an arbitrary block-based file system. No matter how the file system uses the logical device, the changes on the data device are linear (see Figure 2). On the thin volume, the non-sequential addresses are mapped to sequential addresses on the data device through mappings in the metadata device. For example, the metadata of an Ext4 file system is evenly distributed on the thin volume [10], but through mapping, the metadata is written on the data device sequentially. As a result, we can create hidden volumes on the second half of the public volume's data device. Since the data in the public volume are always written sequentially to the data device (regardless of the allocation strategies of the deployed file system on the thin volume), it is very unlikely that they will overwrite the hidden volumes. In other words, the "file system friendly" feature can be achieved. Note that we write random noise to the storage when initializing and encrypting the hidden volume with dm-crypt and thus in the public mode, the adversary cannot differentiate the hidden volume from the random noise.

Another benefit offered by thin provisioning is that the size of a thin volume can be set larger than or equal to the entire storage space. Since the label, the metadata of LVM and the metadata device of thin pool will occupy space, the thin volume usually cannot use the entire space. However, in the public mode, we can set the thin volume size equal to the size of the entire storage space, by which the adversary will observe the entire space is usable and will not suspect the existence of PDE. Additionally, as thin provisioning is a kernel feature, its existence will not become a clue of the existence of PDE.

4.3 Storage Layout

For the public volume, we use the decoy key to create an encrypted block device over the entire disk. We then use LVM to create a physical volume on the encrypted block device. A physical volume label will be placed in the second 512-byte sector and the first 512-byte sector remains unused according to the default LVM configuration [9]. We further create a volume group within this physical volume. The metadata of the volume group will be placed right after the physical volume label (see Figure 1).

Next, we create a thin volume (used for storing the regular data) in this volume group. We first create a small ordinary logical volume, which is used as the metadata device for the thin pool. We then create a large ordinary logical volume in the remaining space of the volume group. This volume is used as the data device for the thin pool, on which we can create thin volumes. LVM allocates the space in the data device to thin volumes according to the mappings in the metadata device. On thin volumes, we can deploy any block-based file systems (e.g., Ext4).

For the hidden volume, we first calculate an offset using the hidden password and create another encrypted block de-

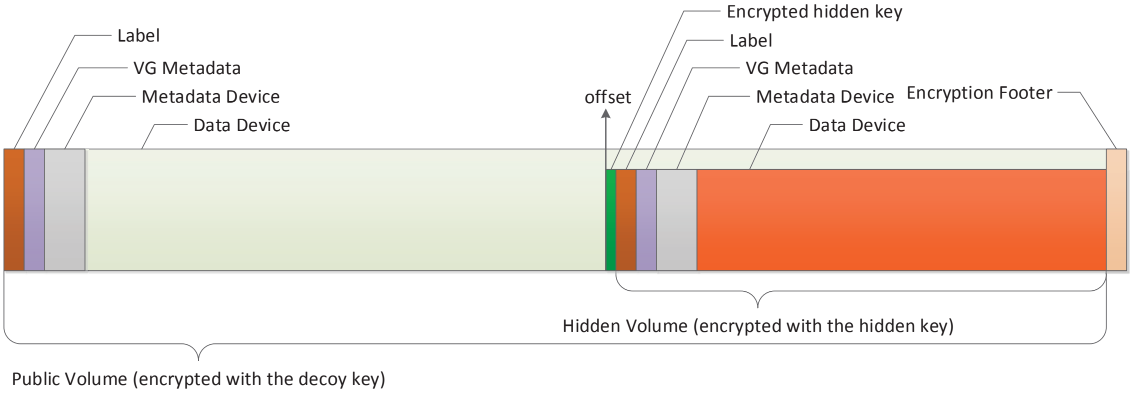

Figure 1: MobiPluto Storage Layout

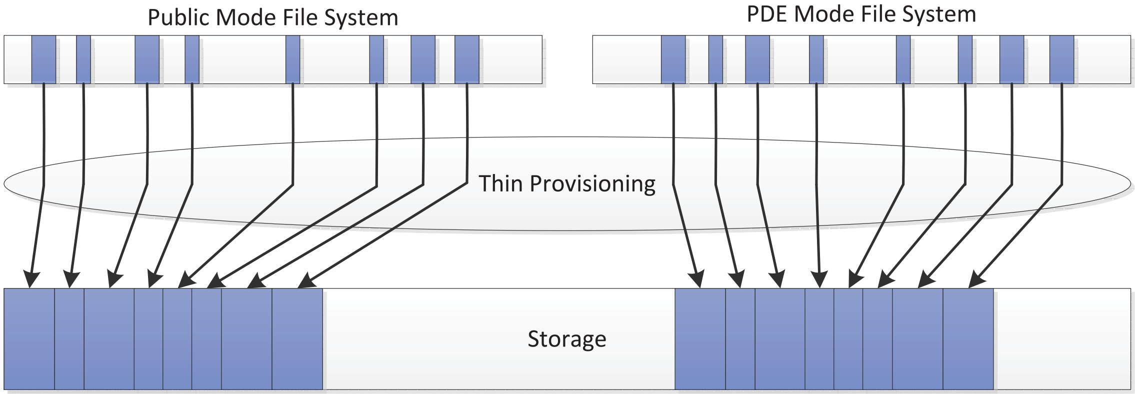

Figure 2: Thin Provisioning and the Hidden Volume

vice within the space between the offset and the end of the storage medium (Figure 1). We then create a thin volume (used for storing the sensitive data) following the aforementioned steps. Note that any block-based file systems can also be deployed on this thin volume.

4.4 Size Calculation

In this section, we describe how to calculate the size of the hidden volume, the metadata device and the data device. For deniability purpose, the thin volume size should be set the same as the total disk size in the public mode.

The hidden volume starts at a specific offset on the storage medium. MobiPluto generates this offset using hidden password in the following way [33]:

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

Here, $vlen$ denotes the number of 512-byte sectors on the physical block device; H is a PBKDF2 iterated hash function [17]; pwd is the hidden password; $salt$ is a random salt value for PBKDF2 and it is the same as the one stored in the encryption footer. Thus, we do not need to store another salt.

The hidden volume is stored after the encrypted hidden key (stored at the offset). The hidden volume size can be calculated as follows:

$$S_{hid} = vlen - \text{offset} - S_{key} - S_{footer}$$

Here, S_{key} and S_{footer} denote the size of the encrypted hidden key and of the encryption footer respectively.

The amount of metadata will vary according to the block size of the thin provisioned devices, the size of the thin provisioning pool and the maximum number of overall thin provisioned devices and snapshots. There is a tool named "thin_metadata_size" in the thin provisioning tools and it can return the size of thin metadata according to the above input. However, it is space-consuming to add this tool to "boot.img" and our scenario is very simple as we only have one thin volume and no snapshots. We give a calculation of thin metadata device size:

$$S_{meta} = S_{req} \times vlen/S_{block}$$

Here, S_{block} is the block size of the thin provisioned devices. S_{req} denotes the average space needed by each data device block in thin metadata device. After the thin metadata device is created, the free space in the volume group is used to create the data device. Note that the size of thin metadata device is the same in both the public volume and the hidden volume.

5. IMPLEMENTATION

We implement a prototype of MobiPluto on a LG Nexus 4 device and its Android version is 4.2.2. The source code of the implementation has been released on GitHub[1]. We add about 1000 lines of C code into the Android volume mounting daemon (VOLD). We also change some of the default kernel configurations so that we could use the required features (e.g., thin provisioning in kernel). In addition, we

[1] https://github.com/FengweiZhang/MobiPluto

compile Logical Volume Manager (LVM) and thin provisioning tools [34] for Android and put them in a boot image.

5.1 Thin Provisioning on Android

In this section, we describe how we run thin provisioning on Android. Thin provisioning is available in the Linux kernel since version 3.2. Android 4.2.2 uses the Linux kernel 3.4, but the default configuration disables this feature. Therefore, we have to enable it and recompile the kernel. In addition, since we use AES-XTS, the `xts` and `gf128mul` kernel crypto modules should be enabled, too.

It is not enough to enable only the thin provisioning feature in the kernel. We have to use LVM to setup logical volumes. Furthermore, we use thin provisioning tools to activate the thin volumes. Thus, we compile LVM and thin provisioning tools for Android. The compiling process requires a specific environment for Android. Besides `gcc` and `g++` tool chains for Android, both tools need to be statically linked. For static compiling, we add "–enable-static_link" when configuring LVM and we add "LDFLAGS= -static" to the makefile of thin provisioning tools. Furthermore, the default LVM configuration does not enable the thin provisioning, so we add "–with-thin = internal" in the LVM configuration for that.

Next, we add both tools to the `boot.img` using `unpack-bootimg` and `mkbootimg` which are provided by AOSP [8]. Note that we modify the access permissions of these files by adding "chmod" command to `mako.init` in the `boot.img`. Otherwise, we are not able to use them. After enabling thin provisioning feature in the kernel and adding the tools to the `boot.img`, we can use thin provisioning on Android.

5.2 User Interface and Pre-boot Authentication

Users can use a command-line utility, `vdc`, to activate MobiPluto PDE; the command is as follows: "vdc cryptfs pde <wipe> <decoy_pwd> <hidden_pwd>". The default Android shell does not maintain history, thus the commands or the passwords cannot be retrieved from a captured Android device.

To make the hidden volume indistinguishable, we first wipe the entire internal storage with random noise. For the public volume of MobiPluto, we use a random key to create an encrypted block device and store the encrypted key and the salt in the encryption footer. We then create a thin volume on the encrypted block device and create an Ext4 file system on the thin volume. We have described the procedure of initializing a public volume and a hidden volume in Section 4.3. The size of the metadata device is calculated according to Section 4.4. S_{req} is chosen as 50 for now, and a more accurate value will be investigated in our future work.

When the device is booted but fails to find a valid Ext4 file system on the userdata partition, the system will ask the user to enter a password. The default Android will use this password to decrypt the key in the footer and decrypt the storage medium with this key. If a valid Ext4 file system can be mounted, the system will continue to boot. However, MobiPluto creates a thin volume instead of an Ext4 file system on the encrypted block device. It would be time consuming if MobiPluto enables the thin volume to check the existence of Ext4 file system by mounting it. To reduce the time of checking, we use a Message Authentication Code (MAC) in the following way: 1) We calculate a master secret S with the corresponding password and the salt in the encryption footer using PKCS5_PBKDF2_HMAC_SHA1 function. 2) The encryption key of the volume key is derived from S and a character string "encryption_key". 3) The MAC key is derived from S and another character string "mac_key". 4) We use the MAC key to compute a MAC for the entire volume header (this header is stored in the second 512-byte sector of the volume). 5) We store the MAC in the first 512-byte sector of the volume, which remains unused when we use LVM. When the user enters a password, the system will check the MAC of the public volume. If it is matched, the password is the decoy password and the system will boot into the public mode. Otherwise, the system will calculate an offset and check the MAC of the hidden volume. If this MAC is matched, the password is the hidden password and the system will boot into the PDE mode. Otherwise, the system asks for another password.

6. EVALUATION RESULTS

6.1 Security Analysis

Deniability provided by hidden volumes. In general, storage units are not filled with random data when coming from the manufacturers. In addition, the operating system installation procedures do not fill the entire storage with random data. Thus, the adversary may suspect the existence of PDE after decrypting the disk with a decoy key, as it can find out random data which is not "accounted for" (Sec. 2.1). A plausible explanation from the device owner can be, he/she always fills the disk with random data before putting file systems on it. Although the adversary has the full knowledge of MobiPluto design (Sec. 3.3), without knowing the secrets, it cannot prove the existence of hidden volumes, as they are encrypted by FDE and are indistinguishable from the initially filled random data (Note that MobiPluto uses the encryption function for FDE as the pseudorandom number generator).

To prevent the adversary from identifying hidden volumes without recovering any hidden plaintexts, MobiPluto uses XTS as the block cipher mode, which has been designed for disk encryption, and is able to prevent attacks such as ciphertext manipulation and cut-and-paste [27].

Thin provisioning/LVM specific security issues. MobiPluto uses both thin provisioning and LVM tools. Thin provisioning/LVM in either the public mode or the hidden mode will have its own label, VG metadata, metadata device and data device (Sec. 4), which are stored in its own userdata partition, encrypted by dm-crypt with different keys (i.e., decoy key and hidden key). For the hidden mode specifically, the location of the userdata partition is secret and can only be derived when knowing the hidden password. Thus, when the adversary looks into the public volume (i.e., in the public mode), it will not have any clues of the data related to thin provisioning/LVM in the hidden volumes.

Other security issues. We require device owners to choose strong passwords resilient to guessing. The data and existence leakage of hidden files into temporary files, swap space, or OS logs can be mitigated by the two modes of MobiPluto [33]. MobiPluto is built for mobile devices, which usually use flash storage. An analysis of leakage from flash storage can be found in Mobiflage [33]. However, it is still not clear

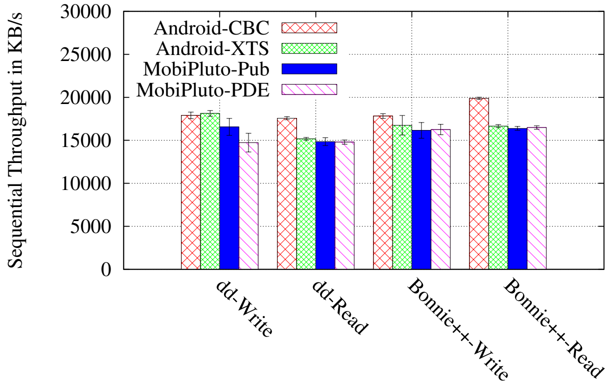

Figure 3: Sequential Throughput test of dd and Bonnie++ in KB/s

how this leakage can affect deniability and how to avoid this leakage. It is the subject of future work to further understand this.

6.2 Performance Evaluation

In this section, we describe experimental results of MobiPluto prototype and explain the performance impact on the device. We summarize our findings and provide conclusions.

The main difference between MobiPluto and the default Android is that MobiPluto uses 1) AES-XTS and 2) thin volumes, so we intend to understand how these two points impact the performance. We use three experiments to understand the performance differences among 1) the default Android FDE, 2) the XTS Android FDE (i.e., only replace the AES-CBC of Android FDE with AES-XTS), 3) the public mode of MobiPluto, 4) and the PDE mode of MobiPluto. We conduct the experiments on the internal storage of a LG Nexus 4.

First of all, we use a popular Linux command tool, "dd", to measure the storage performance of the four systems. For the write speed, we execute following command, "time dd if=/dev/zero of=test.dbf bs=400M count=1 conv= fdatasync". It measures the time for writing a 400MB file to the storage. Note that "conv=fdatasync" ensures the data is written to the disk instead of a RAM buffer. In addition, we use "time dd if=234.mp4 of=/dev/null bs=400M" to measure the read speed. Here "234.mp4" is a multimedia file and its size is 3 GB. Note that "dd" command tests the sequential I/O performance. Additionally, we use a popular benchmark, Bonnie++ [20], to test the sequential I/O operations. We conduct each experiment 10 times, and the average results and standard deviations are shown in Figure 3. We can see that the AES-XTS has a small impact on the read speed, and the use of thin volumes has little influence on the performance.

We use AndroBench [25], a popular storage benchmark for Android-based mobile devices, to conduct the second experiment. AndroBench measures the sequential and random I/O operations and SQLite transactions. We repeat the experiment 10 times. Figure 4 shows the I/O access speed that includes sequential and random read/write. In Figure 4, for sequential I/O performance, we can get a similar conclusion to "dd" and Bonnie++ tests. For random I/O access, the AES-XTS also has a small impact on the access speed. Additionally, Figure 5 shows throughputs of three SQLite transactions, which are SQLite-Insert, SQLite-Update, and SQLite-Delete. Though AES-XTS decreases the throughput of I/O access, it improves the storage performance of SQLite, compared with AES-CBC. However, we can see that from the experiment results, the use of thin volumes decreases the performance. On the whole, the storage performance is not significantly affected.

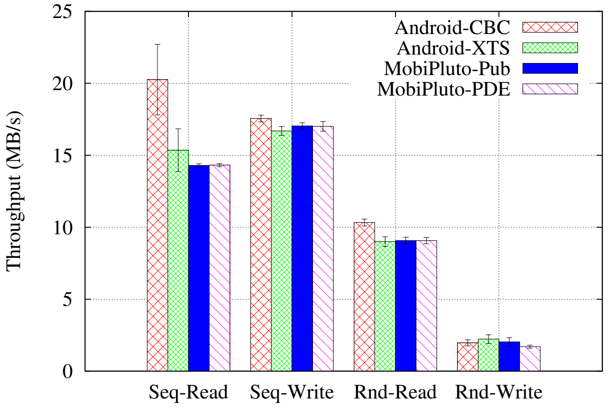

Figure 4: I/O Throughput test of AndroBench in MB/s (Seq: Sequential, Rnd: Random)

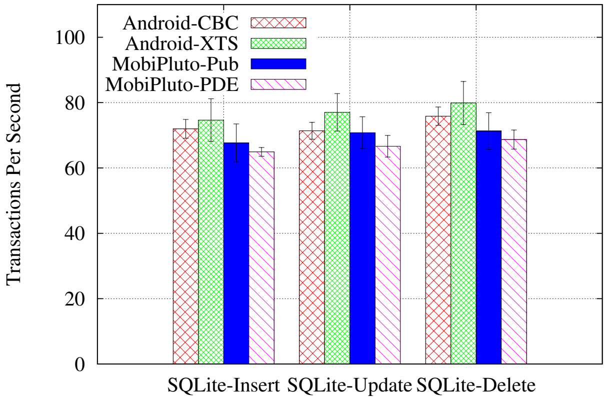

Figure 5: The SQLite performance test of AndroBench in transactions per second

We use Bonnie++ [20], a benchmark suit conducting tests on hard drives and file systems, for the third experiment. Figure 6 shows the number of file system operations can be done in one second. We can see from Figure 6 that both AES-XTS and thin provisioning have a low performance overhead on file system operations, and it gives us a good result that MobiPluto is competitive with the default Android FDE in I/O performance.

Additionally, Bonnie++ shows us the CPU overhead, which indicates the power consumption difference. The CPU overhead shows the CPU requirement of encryption and space allocation, so we can get the power consumption result from it, as shown in Figure 7. We can see that the AES-XTS decreases the power consumption and the use of thin volumes has little impact on it. Note that we repeat 10 times for each experiment with Bonnie++, and we show the averages and standard deviations of the experiment results.

The initialization time and the booting time are two factors that affect the user experience. If a user wants to encrypt the phone, the default Android FDE performs an inplace encryption to the internal storage but MobiPluto per-

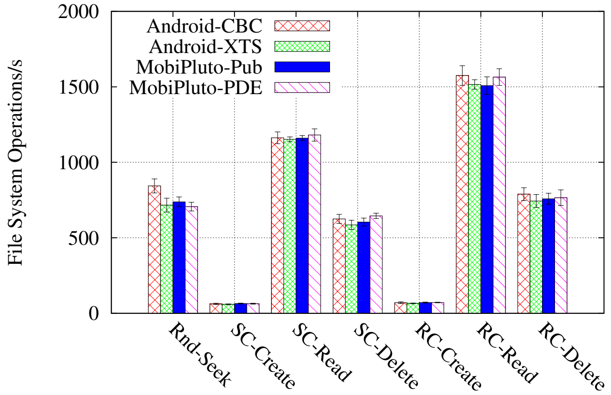

Figure 6: File system operations per second measured with Bonnie++ (Rnd: Random; SC: Sequential Create; RC: Random Create)

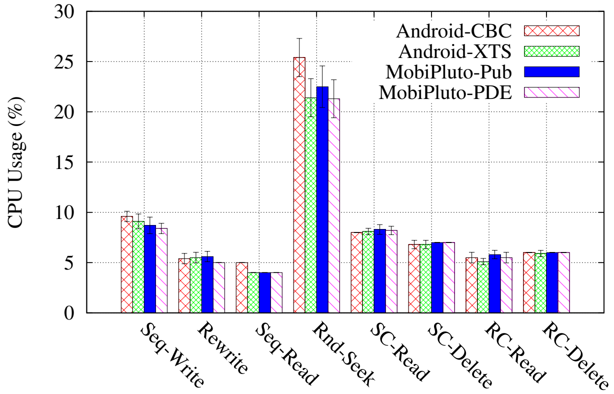

Figure 7: CPU Usage test of Bonnie++ in percentage (Seq: Sequential; Rnd: Random, SC: Sequential Create; RC: Random Create)

forms a two pass random-wipe. Thus, MobiPluto takes twice as long to encrypt the storage. However, the initialization is a one-time procedure which will not be repeated. To test the initialization time, we use a timer to record the time interval between the moment when the passwords are entered through the vdc command, and the moment when the screen shows up a user interface for password entering. We analyze the booting time by reading the logs of the system. The experimental results of initialization time and booting time are shown in Table 1.

7. DISCUSSION

Precautions against collusion between the adversary and carriers/service providers. The collusion between the adversary and carriers may disclose the existence of hidden volumes. If a user connects to external networks when working in the PDE mode, a malicious carrier may provide the adversary this user's connection activity logs. As the logs from the carrier may be different from the logs in the public mode of the captured device, the adversary may obtain evidence for the existence of PDE mode. Thus, when working in the PDE mode, we recommend the user open "Airplane Mode" and remove the SIM card. If the user feels the necessity to communicate with the external networks, anonymity should be used. Similarly, to prevent the collusion of the adversary and the service providers, we recommend they use a secondary account with a pseudonym when using any web services in the PDE mode [33].

Block-based file systems. MobiPluto relies on dm-crypt, a kernel feature that can encrypt whole disks, partitions, software RAID volumes, logical volumes, as well as files. dm-crypt provides a logical block device interface to the upper level, such that any block-based file systems can be built on top of it. Flash file systems (e.g., YAFFS [30]) however, are specifically designed to accommodate the special nature of raw flash memory [23]. They may not be used on top of block devices, and thus may not be used in MobiPluto. Note that block devices are broadly used as the internal storage for mobile devices nowadays. For example, eMMC [32] is used in a large number of Android phones (e.g., Samsung Galaxy S 5, Samsung Galaxy Note 4, Google Nexus 6, LG G3, HTC One M9). Thus, MobiPluto can be used extensively in mobile devices. The raw flash is used as the internal storage in a few early Android devices like Google Nexus One. However, it is rarely used in the latest Android devices. Deploying MobiPluto on the raw flash is possible by modifying the encryption layer (e.g., dm-crypt) to accommodate the special characteristics of raw flash (e.g., wear leveling). This may require moderate engineering work on dm-crypt without affecting other components of MobiPluto framework.

Supporting multi-level deniability. There are debates over the effectiveness of multi-level deniability [33]. Extending MobiPluto to support multi-level deniability can be achieved by adapting the multi-level deniability solution introduced in MobiHydra [36]. To support multi-level deniability, each deniability level is corresponding to a different hidden volume. Each hidden volume starts at a different secret offset, and extends toward the end of the storage medium. The hidden volumes will thus overlap, and this overlap leads to a situation that the data from different hidden volumes may overwrite each other. This is a common issue in all the prior PDE solutions [15, 28, 29, 33, 35, 36]. MobiPluto mitigates this "overwrite" issue in the following aspects: 1) Each secret offset is derived from the hidden password corresponding to that deniability level [36]. When generating offsets, we usually ensure a minimum separation distance between offsets [36]. As the data written to a hidden volume are usually placed sequentially (starts from the offset), overwrite will not happen until the total amount of data written to a certain hidden volume exceeds the minimum separation distance; 2) In practice, the sensitive data stored in each deniability level is usually small in size; 3) Users will be carefully guided when working in the PDE mode. Specifically, when a user is working in the PDE mode and using a hidden volume, a daemon program should be used to monitor the storage usage in the corresponding hidden volume, and the user will be notified if the total amount of writes to this hidden volume is approaching the minimum separation distance.

Supporting other operating systems. To use MobiPluto in other operating systems (mobile or desktop), we have to ensure that the OS supports both thin provisioning and block-layer encryption. Thin provisioning helps transform the non-sequential allocation on the thin volume to sequential allocation on the physical volume. The block-layer encryption is also required, as MobiPluto needs to be built on top of encrypted block devices.

Table 1: Initialization time and booting time

	Initialization	booting time (wrong pwd)	booting time (decoy pwd)	booting time (hidden pwd)
Android FDE	18min23s±1s	0.19±0.02s	0.29±0.02s	N/A
MobiPluto	37min2s±2s	1.98±0.03s	1.36±0.02s	2.35±0.03s

8. RELATED WORK

Deniable encryption is an emerging security paradigm in network communications [19], disk storage, cloud storage [22], etc. In disk storage, most of the existing work relies on either steganography or hidden volumes to achieve deniability.

Steganography-based. Anderson et al. [16] propose the first file encryption scheme with PDE support. They present two solutions: Hiding blocks within cover files and hiding blocks within random data. However, both solutions are not suitable for performance-sensitive mobile devices due to high storage and I/O overheads. StegFS [28] is a deniable-encryption version of the work of Anderson et al. [16]. It uses the second approach in [16] to work on Ext2 file system. However, the existence of the modified Ext2 driver and the external block table may make the system suspicious. In addition, the disk usage rate is low due to the collision avoidance. Pang et al. [29] propose a different design that blocks used by hidden files are marked as occupied in the bitmap, and it uses "abandoned blocks" and "dummy blocks" to achieve deniability. Unfortunately, their design is disk space inefficient.

Hidden volumes-based. TrueCrypt [35] and FreeOTFE [6] are two well-known PDE tools relying on hidden volumes. Compared to TrueCrypt, MobiPluto decouples file system from the underlying storage medium, achieving "file-system friendly" feature. We summarize the differences between TrueCrypt and MobiPluto as follows: 1) TrueCrypt is sensitive to file systems and its hidden volume can only be deployed on top of the storage using FAT or NTFS [3]. If the storage medium is using a file system with distributed metadata or non-sequential block assignment (e.g., Ext4), TrueCrypt may not work. However, MobiPluto works with any block-based file systems due to its "file system friendly" design. 2) TrueCrypt uses a special boot loader to obtain the user's password before the OS is loaded, but using such a special boot loader may make the system suspicious and may lead to compromise of deniability. To handle the password, MobiPluto uses thin provisioning and regular FDE which should be standard in Android. Thus, the deniability offered by MobiPluto is stronger than that offered by TrueCrypt. 3) TrueCrypt is not a default module in desktop, and the adversary can tell the difference between a TrueCrypt volume and a regular volume (TrueCrypt volume header contains either random data (i.e., salt) or encrypted fields, and regular volume header contains meaningful plaintext fields), so he/she can easily identify the existence of TrueCrypt and may suspect the existence of hidden volumes. This may lead to compromise of deniability. However, FDE has been a default module in Android since version 3.0 and the adversary cannot tell the difference between MobiPluto and Android FDE by disk analysis (the MobiPluto footer cannot be differentiated from FDE footer), so he/she cannot identify the existence of MobiPluto, which is good for deniability. Mobiflage [33, 15] builds the first PDE scheme for mobile devices. It is implemented in two versions: one for FAT32 file system in external storage [33], and the other for Ext4 file system in internal storage [15]. The FAT32 version is not suitable for mobile devices without external storage; the Ext4 version needs to significantly modify Ext4 file system that introduces a large attack surface against PDE. MobiHydra [36] improves Mobiflage by addressing a new booting-time attack. In addition, it introduces multi-level deniability and supports mode switching without rebooting. Blass et al. [18] present HIVE, a desktop PDE scheme that can defend against a multiple-snapshot adversary. HIVE relies on write-only oblivious RAM, which suffers from a high performance overhead.

Others. Ragnarsson et al. [31] propose to use thin provisioning to provide deniability. However, their solution requires significant modifications of thin provisioning. In addition, they do not provide any proof-of-concept implementation. Peters et al. [30] introduce DEFY, a deniable encrypted file system based on YAFFS. DEFY is the first deniable file system specifically designed for flash-based, solid-state drives. It follows a log-structured design, motivated by the technical constraints of flash memory.

9. CONCLUSION

In this paper, we have proposed MobiPluto, a file system friendly PDE solution for mobile devices. MobiPluto achieves the deniability feature as nothing but a "side-effect" of equipping thin provisioning, which is a well-established tool in Linux kernel. Most significantly, MobiPluto utilizes thin provisioning to build an additional layer that can transform the non-sequential allocation on the thin volumes to sequential allocation on the underlying storage medium. This renders it feasible to achieve "file system friendly" PDE using hidden volumes. We have implemented a prototype of MobiPluto on a LG Nexus 4 device and our extensive evaluations have shown that MobiPluto only introduces a small performance overhead.

10. ACKNOWLEDGMENTS

We would like to thank our shepherd, Ariel Feldman, and anonymous ACSAC reviewers for their insightful suggestions and advice. This work is partially supported by National 973 Program under Award No. 2014CB340603, and Strategy Pilot Project of Chinese Academy of Sciences under Award No. XDA06010702. Bo Chen is currently supported by US ARO grant ARO W911NF-15-1-0576.

11. REFERENCES

[1] LVM Administrator's Guide. https://www.centos.org/docs/5/html/Cluster_Logical_Volume_Manager/, 2007.

[2] "partitioning" your Nexus S using LVM. http://forum.xda-developers.com/nexus-s/general/howto-partitioning-nexus-s-using-lvm-t1656794, May 2012.

[3] TrueCrypt User's Guide. https://www.grc.com/misc/truecrypt/TrueCrypt%20User%20Guide.pdf, 2012.

[4] BitLocker Overview. https://technet.microsoft.com/en-us/library/hh831713.aspx, 2013.

[5] Consider LVM on Android. *http://forum.cyanogenmod.org/topic/4226-has-anyone-considered-lvm-on-android/*, 2013.

[6] FreeOTFE - Free disk encryption software for PCs and PDAs. version 5.21. *Project website: http://sourceforge.net/projects/freeotfe.mirror/*, 2014.

[7] Android encryption. https://source.android.com/devices/tech/security/encryption/, 2015.

[8] AOSP: Android open source project. http://source.android.com/, 2015.

[9] Appendix E. LVM Volume Group Metadata. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Logical_Volume_Manager_Administration/lvm_metadata.html, 2015.

[10] Ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout, 2015.

[11] FAT file system. https://technet.microsoft.com/en-us/library/cc938438.aspx, 2015.

[12] OS X Yosemite: Encrypt the contents of your Mac with FileVault. https://support.apple.com/kb/PH18637?locale=en_US&viewlocale=en_US, 2015.

[13] Pluto-King of the Underworld. http://www.crystalinks.com/plutorome.html, 2015.

[14] Samsung eMMC memory. http://www.samsung.com/global/business/semiconductor/product/flash-emmc/overview, 2015.

[15] Adam Skillen and Mohammad Mannan. Mobiflage: Deniable storage encryption for mobile devices. *IEEE Trans. Dependable Sec. Comput.*, 11(3):224–237, 2014.

[16] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *Information Hiding*, pages 73–82. Springer, 1998.

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

[18] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu. Toward robust hidden volumes using write-only oblivious RAM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214. ACM, 2014.

[19] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable encryption. In *Advances in Cryptology-CRYPTO'97*, pages 90–104. Springer, 1997.

[20] R. Coker. Bonnie++ file system benchmark suite. http://www.coker.com.au/bonnie++/, 2009.

[21] B. Donohue. Android 5.0 data better protected with new crypto system. https://blog.kaspersky.com/full-disk-encryption-android-5/, 2014.

[22] P. Gasti, G. Ateniese, and M. Blanton. Deniable cloud storage: sharing files via public-key deniability. In *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, pages 31–42. ACM, 2010.

[23] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 2–2. USENIX Association, 2012.

[24] J. Assange, R.P. Weinmann, and S. Dreyfus. Rubberhose Filesystem. *Archive available at: http://web.archive.org/web/20120716034441/http://marutukku.org/*, 2001.

[25] J.-M. Kim and J.-S. Kim. AndroBench: Benchmarking the storage performance of Android-based mobile devices. In *Frontiers in Computer Education*, pages 667–674. Springer, 2012.

[26] A. Levin. The 10 Dumbest Risks People Take With Their Smartphones. *http://blog.credit.com/2013/01/the-10-dumbest-risks-people-take-on-their-smartphones-64384/*, 2013.

[27] L. Martin. XTS: A mode of AES for encrypting hard disks. *IEEE Security & Privacy*, (3):68–69, 2010.

[28] A. D. McDonald and M. G. Kuhn. StegFS: A steganographic file system for Linux. In *Information Hiding*, pages 463–477. Springer, 2000.

[29] H. Pang, K.-L. Tan, and X. Zhou. StegFS: A steganographic file system. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 657–667. IEEE, 2003.

[30] T. M. Peters, M. A. Gondree, and Z. N. Peterson. DEFY: A deniable, encrypted file system for log-structured storage. In *22th Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11*, 2015.

[31] B. Ragnarsson, G. Toth, H. Bagheri, and W. Minnaard. Desirable features for plausibly deniable encryption. *https://www.os3.nl/_media/2012-2013/courses/ssn/desirable_features_for_plausibly_deniable_encryption.pdf*, 2012.

[32] E. Silverstein. 2013 Was a Year to Remember for NAND eMMC Memory. http://www.mobilitytechzone.com/topics/4g-wirelessevolution/articles/2014/02/28/371835-2013-a-year-remember-nand-emmc-memory.htm, 2014.

[33] A. Skillen and M. Mannan. On implementing deniable storage encryption for mobile devices. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27*, 2013.

[34] J. Thornber. Thin Provisioning Tools. https://github.com/jthornber/thin-provisioning-tools, 2015.

[35] TrueCrypt. Free open source on-the-fly disk encryption software.version 7.1a. *Project website: http://www.truecrypt.org/*, 2012.

[36] X. Yu, B. Chen, Z. Wang, B. Chang, W. T. Zhu, and J. Jing. MobiHydra: Pragmatic and multi-level plausibly deniable encryption storage for mobile devices. In *Information Security*, pages 555–567. Springer, 2014.