DEFY: A DENIABLE FILE SYSTEM FOR FLASH MEMORY

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Timothy M. Peters

June 2014

COMMITTEE MEMBERSHIP

TITLE:	DEFY: A Deniable File System for Flash Memory
AUTHOR:	Timothy M. Peters
DATE SUBMITTED:	June 2014
COMMITTEE CHAIR:	Zachary Peterson, Ph.D. Assistant Professor of Computer Science
COMMITTEE MEMBER:	Phillip Nico, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	John Bellardo, Ph.D. Associate Professor of Computer Science

ABSTRACT

DEFY: A Deniable File System for Flash Memory

Timothy M. Peters

While solutions for file system encryption can prevent an adversary from determining the contents of files, in situations where a user wishes to hide even the existence of data, encryption alone is not enough. Indeed, encryption may draw attention to those files, as they most likely contain information the user wishes to keep secret, and coercion can be a very strong motivator for the owner of an encrypted file system to surrender their secret key.

Herein we present DEFY, a deniable file system designed to work exclusively with solid-state drives, particularly those found in mobile devices. Solid-state drives have unique properties that render previous deniable file system designs impractical or insecure. Further, DEFY provides features not offered by any single prior work, including: support for multiple layers of deniability, authenticated encryption, and an ability to quickly and securely delete data from the device. We have implemented a prototype based on the YAFFS and WhisperYaffs file systems. An evaluation shows DEFY performs comparatively with WhisperYaffs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Encrypted file systems are the typical solution to protecting sensitive data. Our current encryption mechanisms cannot be broken in a reasonable amount of time if we assume that our adversaries are confined to a brute force approach. In some cases, however, adversaries are more powerful and are able to use coercion to achieve their means. In these cases, standard encrypted file systems are insufficient because they leak the existence of encrypted data–potentially even the size of that data. This scenario is where *deniable file systems* become useful.

Deniable file systems have been a subject of research since their introduction in 1998 [15]. Since that time there have been many valuable contributions to the field. To our knowledge, however, all previous deniable file systems have been designed for magnetic disk drives–or related technologies. The assumptions that previous authors made in their deniable file system designs are not valid if those designs are applied to flash memory. Flash must be treated differently. In this work we present what we believe to be the first deniable file system for flash memory.

The Prevalence of Flash

Mobile devices are becoming increasingly ubiquitous and powerful. They collect and store large amounts of personal or sensitive information. Some users need to protect that data from unauthorized access just as they would on normal platforms. Evidence of this need can be found on the Android Play store where there are a number of privacy-enhancing technology apps. These apps include: ChatSecure [9] (secure texting), WhisperYaffs [7] (an encrypted file system), RedPhone [5] (encrypted calls), TextSecure [6] (secure texting), Orbot [10] (tor for mobile), Lookout [11] (data backups and anti-virus), and many more.

The standard method of preventing unauthorized access to information on mobile devices is the same as in general secure communication: encryption. While encryption serves to limit access to certain files, it does not attempt to hide their existence. In fact, encryption reveals the existence (and often, size) of information that the user does not want others to see.

In many environments, allowing an adversary to learn that a device contains sensitive data may be as damaging as the loss or disclosure of that data. Consider covert data collection in a hostile country, where mobile devices carrying information might be examined and imaged at border checkpoints. Inspectors may discover the presence of encrypted data, or identify changes to the encrypted file system over time, and demand that they be decrypted before allowing passage. This is not a fictional scenario. In 2012, a videographer smuggled evidence of humans rights violations out of Syria. He lacked any data protection mechanisms and instead hid a micro-SD card in a wound on his arm [28]. In another example, the human rights group Network for Human Rights Documentation - Burma (ND-Burma) collects data on hundreds of thousands of human rights violations by the Burmese government. They collect testimony from witnesses within the country that the Burmese government would not want released, putting both activists and witnesses in grave danger should the government gain access to that data [8]. In light of the control exerted by the government over the Internet within Burma [34], ND-Burma activists carry data on mobile devices, risking exposure at checkpoints and border crossings. Risk to activists and witnesses could be lessened if this group used a mechanism to hide their data such that inspectors couldn't reasonably infer its existence on devices.

Deniable File Systems

The common solution to securing data under the aforementioned situations is a class of file system known as *deniable file systems*. Deniable file systems mask all information about the stored data, and provide a user with a means to plausibly deny any storage artifacts on their device, typically by encrypting data with different keys based on the sensitivity level selected for the data. In this paper we present DEFY, the **D**eniable **E**ncrypted **F**ile System for **Y**AFFS. DEFY is specifically designed for flash-based, solid-state drives—the primary storage device found in most mobile devices. The physical properties of flash memory introduces unique challenges to plausible deniability. In particular, hardware-implemented wear leveling essentially forces DEFY to embrace a log-structured design. All known methods to ensure security and prevent data loss (*i.e.* resulting from overwriting hidden blocks) are inapplicable in this setting, as are strategies that require in-place modification of blocks. This causes DEFY to take a significant departure from previous deniable file system designs.

DEFY also provides a number of features not previously offered in prior work:

- DEFY has been designed to be resistant to a more powerful, and more realistic, adversary than previously considered by the literature.

- DEFY is supportive of an arbitrary number of user-defined, security levels;
- DEFY is the first encrypted file system for mobile devices to provide authenticated encryption;
- DEFY provides a fast and efficient mechanism to securely delete data, allowing individual files or the entire file system to be deleted in constant time.

CHAPTER 2

Background

Flash Memory

Pushed by demand for mobile devices, solid-state memory has become a popular alternative to hard disk drives, due to its low power consumption, high speed, low noise, and lack of moving mechanical parts (increasing durability). The evolution of flash memory technology has been a balancing act between cost, capacity, performance, lifespan, and granularity of access/erasure. The most recent generation of flash is NAND flash. NAND is cheaper to manufacture and many more bytes fit into a single die than in previous EEPROM and NOR technologies: current NAND chip sizes are as large as 256GB.

The distinguishing qualities of flash memory require it to be treated differently than disk drives. NAND offers random-access reads and writes at the *page* level, but erasure occurs at the *block* level. For example, an 8GB NAND device with 2^{12} blocks can write an individual page (4KB) but must erase at the granularity of a block (256KB). After erasure, pages may be programmed once, and must be erased before being programmed again. This is known as the *program-erase cycle*.

Pages in NAND flash are accompanied by an Out of Bound (OOB) area that can be used to contain metadata and error correction codes for the page's data. Since writes occur only at the page level, error correction must also occur at the page level, and is often stored in the OOB.

Flash memory has a limited number of program-erase cycles before becoming unreliable, and can range anywhere from 10,000 to 100,000 cycles. To extend their life, many solid-state drives employ a *wear leveling* strategy, whereby drivers internally distribute erasures and writes across the medium, evenly. This results in a disconnect between the logical and physical block address space—rewriting data to a logical address may result in that data being stored in two, separate physical blocks. Wear leveling can be done statically or dynamically. A dynamic implementation does not attempt to move information once it is written—or static. Instead, it allocates pages based on a least-written count. Static implementations target the static data on the device. They will move that static data if the pages it is stored on have been underutilized compared to other pages on the device. Most devices choose the dynamic implementation for simplicity and speed [2].

On Linux, flash devices may be accessed using the memory technology device (MTD) driver, which provides near "raw" access to the flash device. MTD provides no write leveling, and thus no safeties to prevent cell overuse. Unsorted block images (UBI) is a higher level flash interface that provides wear-leveling and volume management. A flash translation layer (FTL) can also be used to provide a simplified, block-level interface, in exchange for a loss of low-level control over data placement and strict overwrites [3].

Secure Deletion

Secure deletion is the process of permanently deleting data. Data is considered to be securely deleted if it has been irrevocably removed from a physical medium. Even if an adversary later gained access to that device and its decryption keys (if applicable), they should be unable to retrieve the deleted data. This can be accomplished by destroying the device, which is usually undesirable, or by making the data permanently inaccessible. The most conventional secure deletion mechanism is overwriting data after it has been erased, which was proposed by Gutman [21]. This method works well for devices that allow data to be updated in-place, like magnetic disk drives. Reardon et al. separate devices into those with in-place updates (like magnetic disk drives) and those without in-place updates (like flash devices and tape-drives) [32]. Without in-place updates, it can be impossible to know if a piece of data has been erased unless the entire device is erased. An alternative to erasing the entire drive is to zero-fill all of the space on the drive that is unused. Unfortunately, a flash controller's write leveler (discussed in section 2) may choose to ignore the zero-fill requests and instead just mark the pages as erased instead of actually zero-filling them. In short, there is no simple way to ensure secure deletion on write-leveled flash memory, which makes it extremely difficult to extend security to flash devices.

An alternative method to overwriting data is to delete it cryptographically. Swanson and Wei [41] proposed encrypting all data on a flash device with a single key. The data on the device can be securely erased by securely erasing that key. Reardon et al. [33] advanced this idea by suggesting that each page should be encrypted with its own key. This allows the granularity of secure deletion to be reduced to the page level. Obviously, this method still requires a small secondary file system where the keys can be stored, but it also brings page-level secure deletion to devices without in-place updates. This is the only way that we know of to provide secure deletion on flash devices.

Authenticated Encryption

Standard encryption methods only ensure the confidentiality of data—they keep it secret—they do not provide integrity—the ability to know if the data has been modified. The user of a standard encrypted file system would be unable to directly detect a change to the ciphertext during the decryption process, an thus such a change could go unnoticed. Authenticated encryption [16] provides both confidentiality and integrity to encrypted data. This ensures that if data is decrypted correctly, the content is what was originally encrypted. This property is particularly synergistic in environments where storage devices may be ceased for inspection.

Authenticated encryption requires message expansion—the ciphertext is larger than the original plaintext. Existing work in encrypting file systems (*e.g.* [12, 17, 42]) use only unauthenticated block ciphers, which preserve message size, to meet the alignment constraints of block-based storage devices. In practice, additional storage must be found for the bits of the ciphertext expansion.

The process of encrypting data takes a plain text and produces a ciphertext and an authentication tag. The authentication tag is generated by a Message Authentication Code (MAC) or hash function. The ciphertext and the authentication tag can be created in three ways. The most secure method of authenticated encryption is to encrypt the message data and then hash that ciphertext to generate the authentication tag. In reverse, the ciphertext is rehashed and validated against the given authentication tag. Without breaking the hash function, an adversary shouldn't be able to modify the ciphertext or the authentication tag in anyway that is undetectable.

Encrypted File Systems

There are currently two options for encrypted file systems: encrypted file systems or full disk encryption. Encrypted file systems encrypt data at the file system level. The meta-data to those files may or may not be encrypted along with the files. Blaze presents an encrypted file system that works with the standard Unix file system interface and encrypts the associated meta-data [17]. Wright produced another such encrypted file system that works at the user level [42].

Full disk encryption schemes encrypt data at the block-device driver level. These encryption schemes usually lack visibility of actual files. Instead they ensure that each block on the device is encrypted regardless of its contents. TrueCrypt is one such full disk encryption scheme [12], WhisperYaffs is another [7].

Deniable File Systems

The unique quality of a deniable file system over an encrypted file system is that it is impossible to prove the existence of any file or files in the deniable file system–thus a stored file's existence is always deniable. In general, this deniability is achieved by making the entire file system appear to be random noise. Using a set of keys, the user can unlock parts of a deniable file system while leaving the rest as apparent noise. Even when the entire file system is unlocked, the noise persists in some areas. This prevents an adversary from being able to know when all of the files have been retrieved from the file system. Typically, the adversary to the users of these file systems has the power to force a user to give up at least one key to that file system. Thus, they usually rely on the user to separate sensitive data from normal data and only provide access keys to the non-sensitive data when coerced.

Threat Models

There are two types of adversaries that this file system was designed to protect against: the single-view adversary and the snapshot adversary. It is assumed that these adversaries will know everything about how DEFY works and will guess that it is in use if they find anything related to it on a device. In the context of adversaries, an *access* to the device provides that adversary with a complete raw copy of the device. The adversaries should be unable to prove the existence of any sensitive data if the user sets up the file system correctly and does not reveal the keys to that sensitive data.

Single-View Adversary

The *single-view adversary* is one who is able to access the file system and its user once. Among other scenarios, this models some sort of checkpoint where the file system of a device is inspected. The inspectors may ask for a password to the device and attempt to inspect a raw dump of the device. It assumes that the inspectors will not save the image and associate it with a particular device or individual.

Snapshot Adversary

The *snapshot adversary* is very similar to the single-view adversary, but the inspectors are allowed multiple accesses to the device. They keep the raw images of the device device between checkpoints. Then, by comparing two different raw dumps of the device the inspectors could determine which pages of memory have changed.

This adversary has some very real use cases. An example of a snapshot adversary is a country with multiple security checkpoints that communicate data about their visitors. A reporter would need to pass through a checkpoint on the way into the country and on the way out. That reporter may also need to pass through other checkpoints within the country. Thus the reporter would need to pass through at least two checkpoints, and would need a file system that was strong enough to protect sensitive data against that level of intrusion.

CHAPTER 3

DEFY: Overview

DEFY is a deniable file system that is specifically designed for flash memory. It was implemented as a Linux kernel module based on previous work in both YAFFS and WhisperYaffs.

Dependencies

YAFFS

YAFFS is a file system designed for use on raw NAND flash memory. Due to its simplicity, portability, and small memory footprint, YAFFS is commonly used as the default file system in many mobile devices, including the Android operating system. YAFFS is a true log-structured file system [37, 39] in that write requests are allocated sequentially within the logical address space. Its design is largely motivated by a desire to integrate device-level wear leveling. Next, we briefly summarize YAFFS's design; for a more thorough description, we direct readers to Manning [26] and other resources, such as Schmitt *et al.* [38].

Terminology and Data Structures

YAFFS is described by its designer, Charles Manning [26], using terminology that is slightly different than that of traditional file systems. Every YAFFS partition is compromised of a set of *blocks*, and each block is composed of some number of *chunks*. The unit of allocation is the chunk, which ranges in size from 512-bytes to 32KB and corresponds to one or more pages in the underlying NAND technology. The unit of erasure is the block, with a capacity of 32–128 chunks, depending on the NAND block capacity. YAFFS uses the OOB space provided by a flash device to store chunk metadata and an error correction code.

We remark that there are two versions of YAFFS: YAFFS1 and YAFFS2. The key distinctions between these are two fold: (1) YAFFS1 was designed to work with page sizes up to 1KB while YAFFS2 supports larger pages, and (2) YAFFS2 implements a true log-structured file system, performing no overwrites when new data are written. For further details on the differences of these versions, see Manning [26]. This paper refers exclusively to the YAFFS2 design, thus we use the terms YAFFS2 and YAFFS interchangeably.

Every YAFFS entity (files, directories, links, *etc.*) is maintained as an *object*, with an *object header*. Each object header stores metadata about its respective object, including the object name, size, and location of data chunks. A directory's object header contains the location of the object headers for its subdirectories and files.

Writing

Write requests are divided into chunks, allocated and written sequentially following the leading edge of the log (the last chunk written). If the leading edge is the last chunk of a block, YAFFS searches for the next block past the leading edge that is empty or available for allocation. Every chunk is assigned a *sequence number*—stored in the OOB section of memory. The sequence number is monotonically increasing—*i.e.* the last chunk written has the highest sequence number—making it the new leading edge of the log. The leading edge is the starting point for the system when searching for the next chunk to allocate.

When a file's contents are updated, the affected data chunks are rewritten to the device into new chunks. The old chunks are not affected during the rewrite process, but the new chunks that contain the most recent information are assigned higher sequence numbers. These sequence numbers allows YAFFS to find and use only the most up-to-date chunks. Chunks individually store which parent object they belong to, so the parent objects do not need to be modified each time a chunk is updated. Similar actions are triggered when any other parts of the file system are modified.

Mounting

As in LFS, YAFFS supports special objects known as *checkpoints*, which commit information about the state of the file system to the drive. On mount, YAFFS searches for the most recent checkpoint to reconstruct in-memory data structures. In the absence of a checkpoint, YAFFS scans the entire disk, creating a list of blocks, sorted by sequence number. Then, in descending order, it examines the contents of each block. Invalid chunks are ignored and every valid chunk in the block is added to a corresponding in-memory object (creating an object, if necessary).

Unlike most disk files systems (*e.g.* ext2/3/4, NTFS, HFS+), a YAFFS partition does not need to be formatted before being mounted. If no valid objects or checkpoints are found during mounting, all blocks are marked as available for allocation.

Whisper YAFFS

WhisperYaffs is a system for providing full disk encryption (confidentiality without authenticity) on flash devices [7]. A prototype version of it was released to GitHub by WhisperSystems in 2011. Currently, WhisperYaffs isn't a product offered by WhisperSystems nor does it seem to be under active development. For this project, we updated that GitHub version of WhisperYaffs to work with a more modern Linux 3.8 kernel.

The full disk encryption that WhisperYaffs provides is created using AES-XTS encryption. AES-XTS uses ciphertext stealing to enable encryption of sections of data that are not divisible by the AES block size. Ciphertext stealing takes the ciphertext from an encrypted block and uses it to pad the non-AES-sized block up to the AES block size so that it may be encrypted. This is used to encrypt the OOB section of a chunk, which varies in size, but is smaller than an AES block. WhisperYaffs first encrypts the data of a chunk and then proceeds to steal ciphertext from that encryption to encrypt the OOB section. Both the data and the OOB section of a chunk must be retrieved to decrypt either.

XTS mode requires both a key and a tweak. WhisperYaffs uses two separate tweaks, one for each encryption (chunk and OOB area). When the chunk data is encrypted, WhisperYaffs uses the logical chunk address multiplied by two. For the second encryption of the OOB data, WhisperYaffs encrypts using the logical address multiplied by two plus one. Thus, the two tweaks are related but unique for each chunk in the device.

The encryption key that WhisperYaffs uses is randomly generated when the file system is created. It is stored in the first good block of the flash device. That block is subsequently hidden from the rest of the file system and encrypted using the user password—which is expanded through PBKDF2.

Interestingly, the designers of WhisperYaffs chose not to encrypt the sequence number of the chunks. This does not leak any of the contents of the device, but it could be used to determine which chunks were changed most recently. If an adversary had two disk images he could determine which chunks had changed and thus determine an upper bound on the amount of changed data.

System Overview

Like WhisperYaffs, DEFY encrypts each chunk including the OOB areas resulting in a full disk encryption. What distinguishes DEFY from WhiserYaffs and YAFFS, is its ability to obfuscate the existance of data.

The general architecture of DEFY can be separated into a few key components. The layout of these components is presented in figure 3.1 and each component is briefly discussed below.

Operating System Interaction

DEFY is a Linux kernel module, and as such it implements the Virtual File System (VFS) interface. The VFS interface allows the operating system to ask for a plethora of information from DEFY. This can include file, directory, or general file system information. It also allows the operating system to instruct DEFY to perform certain actions–like mounting and unmounting. The VFS interface is common to all Unix file systems so we will leave a discussion of its full requirements and specifications to other literature.

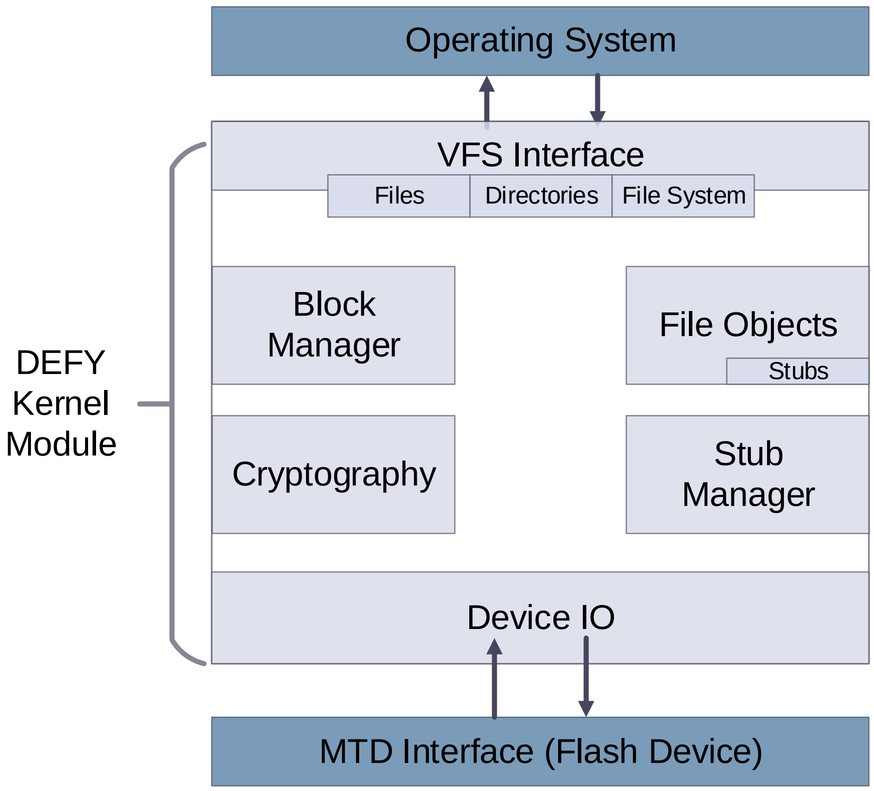

Figure 3.1: A high-level system view of the DEFY kernel module.

File Objects

Each file, directory, and link that is contained in DEFY is mapped to a specific file system object. Just like YAFFS, all of these file objects are held in memory while DEFY is mounted. The in-memory object contains all pertinent information about itself, its attached data blocks, and each of its data chunks' stubs (discussed below). Objects also contain links to their children and parents, these can be used to traverse the file object tree. Each file object can be found using its unique name or associated object number.

Block Manager

The block manager tracks the state of each erase block on the device. The state of each of these blocks is determined on mount and then updated as changes occur. DEFY's list of possible block states is shorter than YAFFS (see [26]). It allows for blocks to be: allocating, dead, assumed empty, or used. DEFY can only assume that blocks are empty because all blocks are technically used at all times. This is part of the deniability framework that ensures that no block looks different from any other if no keys are present.

Stub Manager

The encryption process for each chunk generates a matching stub that is required to decrypt that chunk. The stub manager manages these stubs and provides an interface to access them in the system. It does not generate stubs, instead it simply arranges them. Given either a logical chunk index or a file object pointer, the stub manager can find the correct stub or stubs to decrypt the associated data from the device. It can also store stubs, delete stubs, and iterate through stubs.

Cryptography

Everything that is encrypted or hashed must be sent through the cryptography module. It also provides functions to generate more secure random numbers and expand passwords based on PBKDF2. Note that, like WhisperYaffs, we did not implement our own cryptographic primitives. We rely on the Linux kernel to provide the actual randomization, hashing, and encryption functions.

Device IO

The device IO module deals with reading and writing data to the flash device–which includes the OOB data. Like YAFFS, DEFY is designed to work with the MTD interface. At this level DEFY may write and read directly from specific pages on the flash device. Corrupted and damaged pages are not hidden from DEFY, the MTD interface will return read or write errors that DEFY notes in the block manager.

DEFY could be ported to work on top of a UBI or a FTL as long as another device was present that could support secure deletion, but we leave this to the future work discussion in chapter 6.

CHAPTER 4

DEFY: Design

DEFY is designed as an extension to the YAFFS file system, with security features inspired by WhisperYaffs. We chose YAFFS because it is designed to operate on raw NAND flash, handles wear-leveling, is widely-deployed, and is open-source. To YAFFS we add authenticated encryption, cryptographic secure deletion, and support for multiple deniability levels that are resistant to strong adversaries. A comparison of DEFY's features with existing work appears in Chapter 7. The following provides a high-level description of DEFY's main design features.

This is our second implementation of DEFY. Our first deniable file system design was implemented using dummy blocks. We, however, found a vulnerability in that design and discarded it. For interest, a discussion of that design is included in the appendix (see 8). Our second design appears to be strong against the single-visit adversary and the snapshot adversary.

Privilege Levels

Like many previous deniable file systems, DEFY separates files into one or more privilege *levels*. The exact number of levels created is specified by the user. The system maintains no record of how many levels exist in the system; it can only know which levels are currently open. When a user reveals a level, all lower levels are also revealed. This is a convenience that helps to minimize the chance of overwriting (see 4) and follows the conventions of previous work.

When mounted, privilege levels appear as directories under the root directory of the file system. Each of these *top-level* directories is associated with a unique name (like level_0, level_1, .., level_n). All files for a deniability level are located below its top-level directory. Assigning deniability to directories at the root level is strategic and provides a number of advantages. Level directories allow for easy inheritance of deniability levels. Objects created within a directory will, by default, inherit the level of that directory, *i.e.* be correctly encrypted at the appropriate level. We believe this behavior to be quite natural, following the tradition of other security semantics (*e.g.* file system permissions), and frees users of the burden of assigning deniability levels to individual files. Separating deniability level namespaces through level directories, also forces users to be more thoughtful, and perhaps, careful about how they categories the sensitivity of their data.

Each of these levels are maintained like separate file systems but they exist in the same logical address space on the device. This is necessary for our deniable encryption scheme that is discussed in the following sections.

Authenticated Encryption

The two key challenges associated in implementing authenticated encryption in DEFY are: (1) designing a file system that can accommodate the data expansion that results from authentication and, (2) designing an encryption scheme that is supportive of efficient and granular secure deletion. Here, we focus our discussion on the former, leaving a discussion of the latter for the next section (4).

DEFY's encryption scheme is presented in figure 4.1. The algorithm takes as input a data chunk, broken into n, 128-bit messages $(p_1, \ldots, p_n)$, the OOB data (oob), a unique identifier for the chunk (id), a unique global counter (t), a level encryption key (K_ℓ) and a level MAC key (M_ℓ). The algorithm implements an encrypt-then-MAC scheme: first encrypting the data and OOB using AES in cbc mode (AES-CBC), then MAC-ing the resulting ciphertext of the data using a SHA256-based message authentication code (HMAC-SHA256).

The encryption of the OOB area is treated specially due to its inconsistent size. The exact size of the OOB area depends on the flash device, but the majority of the time it is not an even multiple of the AES block size–which is required to use the AES encryption algorithm. To fix this problem, we perform ciphertext stealing from c_1: this takes ciphertext from c_1 and appends to oob until there is a full AES block size worth of data to encrypt. The result of the encryption operation is then stored back in oob and c_1–over the previously stolen ciphertext.

An additional cbc mode encryption is performed using the authenticator as its key to complete an all-or-nothing transform (described later). A *stub* (s) is created by XOR-ing all the ciphertext message blocks $(x_1, \ldots, x_n)$ with the authenticator (σ). The resulting tag is small and not secret, rather, it is an expansion of the encrypted data and is subject to the all-or-nothing property. The ciphertext $(x_1, \ldots, x_n)$ is written to disk as data, the encrypted OOB (x_{oob}) is written to the OOB area, and

```
**Input:** Chunk Data $p_1, \ldots, p_n$, OOB Data $oob$,
Chunk ID $id$, Level Constant $t$, Level Encryption Key
$K_\ell$, MAC Key $H_\ell$
1: $ctr_1 \leftarrow id||t||1||0^{128-|t|-|id|-1}$
2: $c_1, \ldots, c_n \leftarrow \text{AES-CBC}_{K_\ell}^{ctr_1}(p_1, \ldots, p_n)$
3: $x_{oob}, c_1 \leftarrow \text{AES-CBC}_{K_\ell}^{ctr_1+n}(oob, c_1)$
4: $\sigma \leftarrow \text{HMAC-SHA256}_H(c_1, ..., c_n)$
5: $ctr_2 \leftarrow id||t||0||0^{128-|t|-|id|-1}$
6: $x_1, \ldots, x_n \leftarrow \text{AES-CBC}_{\sigma}^{ctr2}(c_1, ..., c_n)$
7: $s \leftarrow \sigma \oplus x_1 ... \oplus x_n$
**Output:** Stub $s$, Ciphertext $x_1, \ldots, x_n$,
Encrypted OOB Data $x_{oob}$
```

Figure 4.1: Authenticated encryption for a single chunk in DEFY

the tag (t) is stored as metadata in the parent object.

The decryption algorithm is the exact opposite of this process and is presented in figure 4.2. Note that due to the ciphertext stealing the encrypted OOB data must be decrypted before the ciphertext blocks ($c_1, \ldots, c_n$) can be decrypted. The decryption process of OOB returns both the decryption of OOB and the stolen bytes of c_1.

The same counter and key pair should never be used for encryption more than once, so we use a chunk's physical disk address for id and a global sequence counter t; both characteristics exist within a DEFY object and, by policy, are non-repeatable in a file system. The encryption key and MAC key are also distinct between levels.

We remark that other constructions for achieving all-or-nothing encryption, leveraging other cryptographic modes and algorithms, may provide better performance or a more elegant design. For example, Steps 1,2, and 4 of Figure 4.1 may be combined

```
**Input:** Stub $s$, Ciphertext $x_1, \ldots, x_n$, Chunk ID $id$, Encrypted OOB Data $x_{oob}$, Encryption key $K_\ell$, MAC key $M_\ell$, Level Constant $t$
1: $ctr_2 \leftarrow id||t||0||0^{128-|t|-|id|-1}$
2: $\sigma \leftarrow s \oplus x_1 ... \oplus x_n$
3: $c_1, ..., c_n \leftarrow \text{AES-CBC}_{\sigma}^{ctr2}(x_1, ..., x_n)$
4: $\sigma' \leftarrow \text{HMAC-SHA256}_{M_\ell}(c_1, ..., c_n)$
5: **if** $\sigma' \neq \sigma$ **then** return Error
6: $ctr_1 \leftarrow id||t||1||0^{128-|t|-|id|-1}$
7: $oob, c_1 \leftarrow \text{AES-CBC}_{K_\ell}^{ctr_1+n}(x_{oob}, c_1)$
8: $p_1, ..., p_n \leftarrow \text{AES-CBC}_{K}^{ctr_1}(c_1, ..., c_n)$
**Output:** Chunk Data $p_1, \ldots, p_n$, OOB Data $oob$
```

Figure 4.2: Authenticated decryption for a single chunk in DEFY

into a single call of OCB mode [36], which requires only one pass over the data and is fully parallelizable. Our construction acts as proof-of-concept and an exemplar for achieving our design goals.

Encryption-Based Deletion

The same AON transform that provides authenticated encryption, also provides for a means for efficient and granular secure deletion. The original AON transform, due to Rivest [35], is a cryptographic function that, given only a partial output, reveals nothing about its input. No single message of a ciphertext can be decrypted in isolation without decrypting the entire ciphertext. The original intention of the transform was to provide additional complexity to exhaustive search attacks, by requiring an attacker to decrypt an entire message for each key guess. AON has been proposed to make secure an RSA padding scheme [16], to make efficient smart-card transactions [18, 19, 22], message authentication [20], and threshold-type cryptosystems using symmetric primitives [14].

Our design implements a encryption-based secure deletion scheme that is inspired by Peterson *et al.*'s AON technique for secure deletion of versioned data [31]. The all-or-nothing transform allows for any subset of a ciphertext block to be deleted (*e.g.* through overwriting) in order to delete the entire ciphertext; without all the ciphertext blocks, the chunk can never be decrypted. When combined with authenticated encryption, the AON transform creates a message expansion that is bound to the same all-or-nothing property. This small expansion becomes the *stub* (from section 4) and can be efficiently overwritten to securely delete the corresponding chunk. Indeed, message expansion is fundamental to our deletion model and the AON transform is a natural construct for providing efficient secure deletion for DEFY, as it minimizes the amount of data needed to be overwritten, does not complicate key management, and conforms to our hierarchical deletion model.

Stub Management

As previously discussed (4 & 4), stubs are used in DEFY to support authenticated encryption and secure deletion. They are stored both in the metadata of objects and in a separate *stub storage area*, as shown in figure 4.3.

The metadata of an object is used to store stubs for its child objects: data chunks in the case of a file object, or file objects in the case of a directory object. When a child object is modified, the parent object is updated with a new stub, overwriting the previous stub and securely deleting the old version of that information. As a result of storing a new stub, the parent object is modified. Thus, creating, deleting or modifying any object in DEFY will trigger a *stub rotation* for all the tree objects in the direct lineage of that object up to the top-level directory.

The objects that reside in the top-level directory are still secured using stubs, those stubs are stored in the separate *stub storage area* that is written to another file system. That separate file system must support secure deletion. Some examples of just such a flash file system are presented by Reardon *et al.* [33] and Lee *et al.* [25]–which is also built on top of YAFFS. It would be possible to implement a similar secure deletion scheme in DEFY, but we will leave that for future work.

The stub storage area contains a map of chunk numbers and their associated decryption stubs. It is stored in a stub file of fixed size. This imposes an upper bound on the number of chunks in the top-level directory of the file system–other directories lack this bound. The stub file's size = the number of bytes per stub $\times$ the number of allowed top-level chunks $\times$ the maximum number of levels. Each section of the stub file that corresponds to a particular privilege level is then encrypted using the appropriate key for that level of the file system. The encryption step hides the plain text chunk locations that are used by a particular level. It would be difficult to maintain deniability if the adversary knew exactly which chunks were used by which level of the file system–including the unused levels. When chunk location and matching stub data is encrypted, even changing a single byte in the stub list will result in an entirely different output that is saved to disk.

The user may or may not use all of the available levels in DEFY. If the user chooses not to use all of the levels, they may instruct DEFY to randomize those sections of the stub storage area, to prevent sections of the stub storage area from remaining constant (thus revealing their lack of use). Once the user randomizes the storage area, it contents will be irrecoverable even with the correct key–this is also a quick way to erase a level. Without a key, the snapshotting adversary would only be able to determine that one or more changes took place in the stub storage area. More automatic constructions of this operation are possible, but are also less desirable due to the destructive power of this operation.

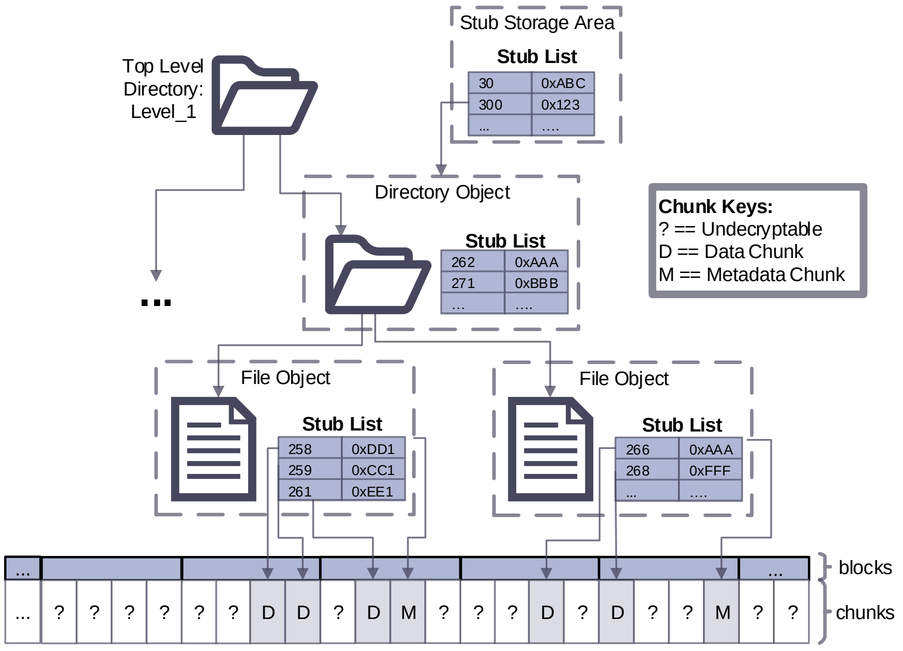

Figure 4.3: An example of how metadata (stubs) and data are stored in DEFY. Note that the stub lists contain the chunk address (left) and the stub value (right).

DEFY's hierarchy of stubs architecture has a number of advantages to achieving fine-grained and efficient secure deletion. Individual objects, be they chunks, files, or directories, may be securely deleted by overwriting their corresponding stubs and performing a stub rotation. This granularity extends to the top-level level directories, allowing a user to securely delete an entire level or the entire file system by overwriting the stub storage area as discussed. And because YAFFS, and thus DEFY, stores all metadata objects in memory, stub rotations only affect in-memory structures in the short term. This behavior limits the system's standard performance overhead to the computation of new stubs. Eventually, however, the memory structures are written to disk and do require additional I/O.

Placeholder Blocks

Placeholder blocks serve two purposes in DEFY: (1) they mark the current head of the log for a particular level and (2) they aid in augmenting deniability. Placeholder blocks were inspired by checkpoints, which are common in log-structured file systems [37, 39]–including YAFFS.

Each time DEFY is unmounted it writes a placeholder block for each level. The placeholder blocks for each level are written in order of greatest privilege to least privilege (see figure 4.4). This places the least privileged level's block last. Thanks to DEFY's convention of unlocking all levels in order, data from the least privileged level will always be available if DEFY is mounted. By placing the least privileged level's block last, we can ensure that all file systems will know where the head of the log is, and subsequently where to begin writing without overwriting data.

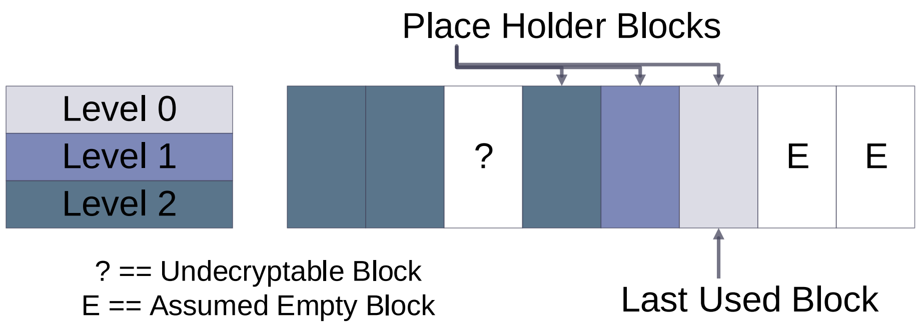

Figure 4.4: A multi-level view of the placeholder block order. The highest privilege level's block is written first and the lowest privilege level's block is written last.

The placeholder block is not a distinct block type–like a checkpoint block. Instead, it is simply a block that was selected by the normal allocator (4) at the head of the file system and filled with file objects from the top-level directory. The objects that are written to the placeholder block follow the normal update cycle–where their previous chunks are deleted. The benefit of using top-level objects is that the stub rotation process is brief.

The ordering of each level's placeholder block ensures that the file system has a consistent size no matter which levels are unlocked. It also ensures that a snapshot adversary cannot determine anything about the existence of higher levels. For example, if the order were reversed and the highest placeholder block were written last, then the snapshot adversary could determine that seemingly empty blocks past the end of the log had changed. That sort of pattern would lead the snapshot adversary to realize there were unlocked levels.

If the file system is used without all levels unlocked then only the unlocked levels will be written to placeholder blocks. Again, this is not problematic given that DEFY loads the least privileged level first and writes that placeholder blocks last. This protection scheme allows DEFY to avoid overwriting in most cases.

Chunk Allocation

The chunk allocator in DEFY has been designed to provide support for distinct privilege levels and reduced data collisions. Its allocation scheme also eliminates the need for a distinct garbage collector.

Unlike YAFFS, chunks and blocks are associated to a particular level in DEFY (as shown in figure 4.5). As such, the chunk allocator will only assign chunks from a particular level's current chunk allocation block. When an allocation block is fully written another block must be selected. The choice of block is the next block that is undecrytable or assumed to be empty. The next block is usually the next physical block, but may wrap around the end of the device.

Once the file system has wrapped around the end of the device it will continue to choose blocks that appear to be empty—or are undecryptable. It may choose blocks

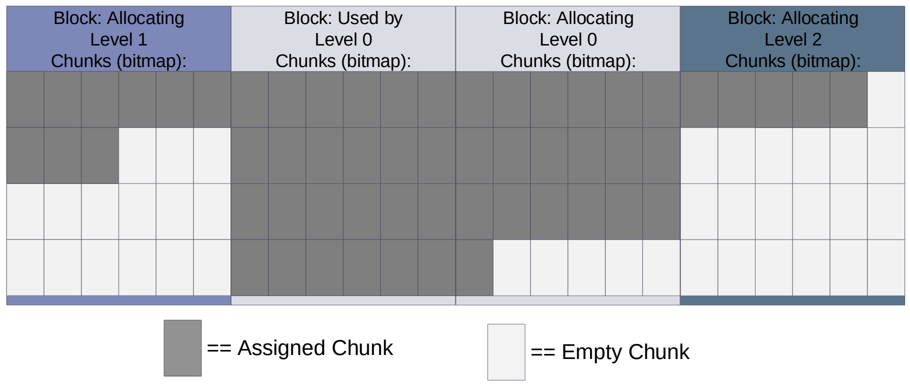
Block: Allocating Level 1 Chunks (bitmap):
Block: Used by Level 0 Chunks (bitmap):
Block: Allocating Level 0 Chunks (bitmap):
Block: Allocating Level 2 Chunks (bitmap):
== Assigned Chunk
== Empty Chunk

Figure 4.5: This is an example of current allocation blocks from various levels. It shows that three levels are currently in use and have open allocation blocks.

that were previously erased—replacing the behavior of a garbage collector—or blocks that are currently in use by unrevealed levels of the file system. The latter is the only case where DEFY will overwrite data. For this reason, the user should only write data when all levels of the file system have been revealed—thus guaranteeing that DEFY will not overwrite data.

Due to flash hardware, a block must be erased before its individual chunks can be rewritten. When a block is selected to be used for allocation it is erased. If that block has not been fully utilized when the file system is closed, it is filled with random data that is again encrypted using ephemeral stubs. This behavior mimics the strategy that is used when the file system is initialized.

File System Operations

Initialization

When the user creates a new DEFY file system, he must provide the number of levels he desires and a unique password for each. Each password is expanded using

PBKDF2 using an appropriate number of iterations [23]. DEFY uses each password to protect the randomly generated encryption key for that level. It also protects the randomly generated hash key and level constant that are used in the encryption process. All of this information is stored in a single erasable block at the beginning of the file system. This is known as a *key block*. One key block is allocated per level even if that level is not used. If the user chooses to use less than the maximum number of levels allowed by the system, those blocks are effectively wasted. This waste is necessary to prevent the adversary from being able to determine if a block has been used or not and thus if a level exists or not. The unused blocks are encrypted with the same procedure as the used blocks. The only difference is that a randomly generated ephemeral key is used. DEFY ignores the existence of these key blocks for all other file system operations. The effective 0th block for the file system is the first block after the key blocks. An example layout of these key blocks can be seen in figure 4.6.

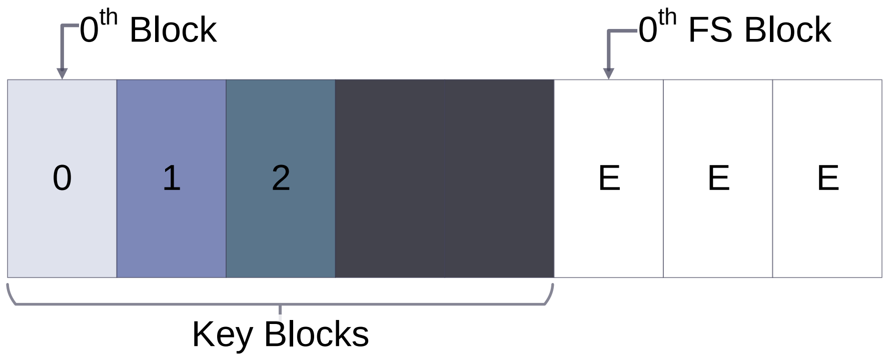

Figure 4.6: A view of the key blocks of a 5 level file file system with only the first 3 levels used. All file system operations and the block allocator treat the 5th block (white) as the 0th block.

Note that DEFY will not write keys to bad key blocks. If a bad block is detected in the key block area, that block will be skipped and the keys will be written to the next block. This will result in holes in the key block area which leads to the 0th file system block being offset further than the maximum number of key blocks. When the file system is mounted, DEFY compensates for these potential holes by iterating forward from the first block on the device looking for key blocks.

The remaining blocks in the file system are erased, and then written with random data. The actual data that we write to each chunk is randomly generated and then encrypted with a random key. The data, its generated stub, and the key are all thrown away between chunks. This method ensures that adversaries can only compare the output of one encryption to the output of another encryption. If data were generated randomly and then written directly to disk, the adversary would be allowed to compare the output of a random number generator with the output of an encryption function, which is less desirable.

The stub file is created at this time with enough space for every levels' top-level stubs. As discussed in 4, it is created on another file system that must support secure deletion. The exact location of the file is user-definable. Its contents, even when blank are encrypted using the level key that is stored in the matching key block.

The number of levels in the system could be changed if desired. To remove a level, its key block and all of its associated file system blocks could be wiped and filled with encrypted random data as previously discussed. Adding a level would be as simple as replacing the next unused key block if space was available. If space wasn't available, the user could choose to remove a level and then re-use the old key block. This feature does not currently exist in our implementation.

The total number of levels in our implementation is set to 30. This is an arbitrary number, that can be changed easily if desired. A smaller maximum number of levels results in reduced memory requirements but also reduces the potential for additional levels.

Mount

The actions to mount the DEFY file system are significantly different from those of YAFFS [26]. YAFFS relies on sequence numbers to determine which chunks are most up-to-date. Those sequence numbers are written to every chunk's OOB area regardless of their status as an object, data, or other type of chunk. When YAFFS is mounted it can either use a checkpoint or scan through the sequence numbers on the device. Using a checkpoint is preferable due to speed, but valid checkpoints are not always available on the device. The scanning alternative consists of a number of steps. First it pre-scans all of the chunks on the device looking for the highest sequence number. As each chunk is scanned, it is added to a sorted list that is ordered by sequence number. Second, that list is scanned in reverse. This guarantees that the objects that are built reflect the most up to date information. If duplicates objects are found later in the list, they are assumed to be out of date. This scanning process is slow, especially with larger devices.

DEFY does not need to rely on sequence number scanning or checkpoints. Instead, it takes advantage of its stub file and the fact that there are no out-of-date chunks in in the file system. The stub file contains the roots–both chunk indexes and stub keys–of each file object in the file system. DEFY uses that stub file as a starting point and walks down through each key tree to build all of the necessary in-memory information.

Placeholder blocks are used by DEFY, but they are mostly used for security reasons as discussed in section 4. The mounting process treats placeholder blocks like any other blocks. DEFY still iterates down the file tree and loads objects as their keys are found. Objects that are part of the placeholder block are merely close to the head of the file system. Thus, placeholder blocks have no effect on load-time performance in DEFY.

Read

A file read operation requires DEFY to determine which chunks contain the desired information. Those chunks are then read in, decrypted, and verified using the data chunk stubs that are stored in the DEFY file object for this file. This operation does not change any of the contents of the device.

Write

To write new information, or update a portion of the file, DEFY needs to write new chunks to the device. This is done by requesting new chunks from the chunk allocator as discussed in section 4. Key stubs are then generated as the data to be written to the newly allocated chunks is encrypted as discussed in section 4. Those stubs are then stored in the active file's stub map and the encrypted data is written to the device. The file object is then re-written with the new stub map, which requires a stub rotation for all objects who have this object as a descendent.

Links

DEFY supports both hard and soft links. They are represented by file system objects in DEFY, which are created and encrypted just like a normal file object. For security reasons, we limited links to only being able to point at content within a specific level of the file system. They are not allowed to point between levels or outside DEFY.

Delete

File deletion simply removes that file's object from memory and removes the stub needed to decrypt that file from the disk. This triggers as stub rotation just like the write operation.

Create

File creation creates a new DEFY file object. That object is written to disk using a chunk that is assigned from the chunk allocator as discussed in 4. The decryption stub for that chunk is then added to this object's parent object which triggers a stub rotation.

Discussion

We believe that DEFY is strong against both the single-visit and the snapshotting adversary. In this section we discuss how DEFY is strong against both adversaries and what is expected of the user to maintain their data security.

Assumptions

The first assumption that we make about DEFY is that the user will be well informed. They should know generally how DEFY works and how to use it correctly. For example, the user of DEFY must use strong passwords. All of the defense mechanisms of DEFY assume that the adversary cannot simply break the keys. It would be far easier for an attacker to brute-force a set of weak passwords than determine if the user of the file system is being truthful. Thus, we assume that a user of DEFY will use strong passwords to protect their obviously valuable data.

We also assume that the device is free of malware. If the device is in anyway compromised, the security of everything on it will also be compromised–including DEFY. Malware could record all of the users passwords or even read data out of the deniable file system once the user opened it.

Creating Deniability

The first and foremost goal of DEFY is to offer plausible deniability in the snapshot adversary model—the strongest model considered in this setting. DEFY gains this feature by using the computational indistinguishability provided by our encryption scheme. Specifically, it should be hard for an adversary to distinguish file system blocks that contain data from an unrevealed level, contain old/deleted data from a revealed level, or contain random data written as part of the initialization process. As a result, the adversary has no cryptographic rationale for compelling additional level revelations. Indeed, it should be impossible to prove (for either party) if the final level is ever revealed.

The basic idea behind our plausible deniability mechanism is to introduce undecryptable blocks into DEFY at all levels through normal file system use. These undecryptable blocks create the necessary obscurity that allows the user to plausibly deny the existence of additional data stored in unrevealed levels. DEFY creates this obscurity through the forward-writing nature of log-structured file systems and its forced delete-on-update policy. Each time a object is modified, a new chunk must be written to reflect that change. If the change invalidates a previous versions of the chunk, that chunk is deleted as part of the stub rotation. This creates randomly placed "holes" in the log that can be explained by normal file system use (see Figure 4.7).

This same mechanism allows multiple levels to be combined in the same logical

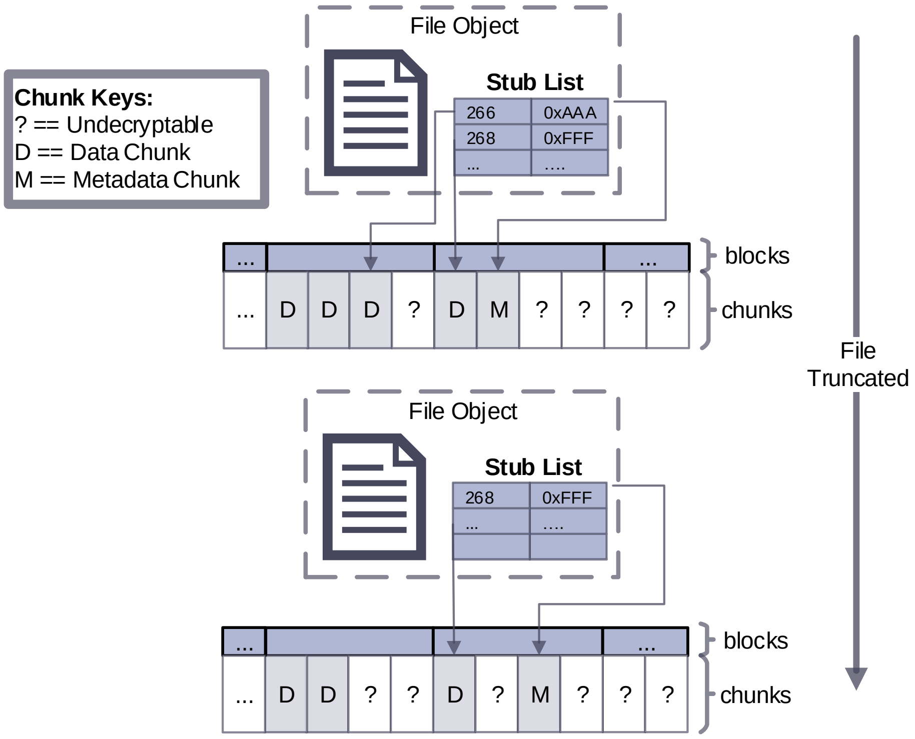

Figure 4.7: A chunk-level view of a file being truncated. As soon as chunk-266 is deleted, a new metadata chunk is written to the head of the log. Both of these operations create holes in the log. In this example blocks are equal to 4 chunks

space. An example of how the blocks could be physically laid out in flash is shown in Figure 4.8. An adversary cannot gain any information about the unrevealed levels, even when a subset of levels have been revealed. Undecryptable blocks may be attributed to old versions of data or metadata. The frequency of which, would rely entirely on the user's usage patterns, the types of files, and the programs used to access those files.

Stub Storage Area

To maintain deniability against the strongest type of adversary, the user needs to periodically randomize the unused sections of the stub storage area. As previously

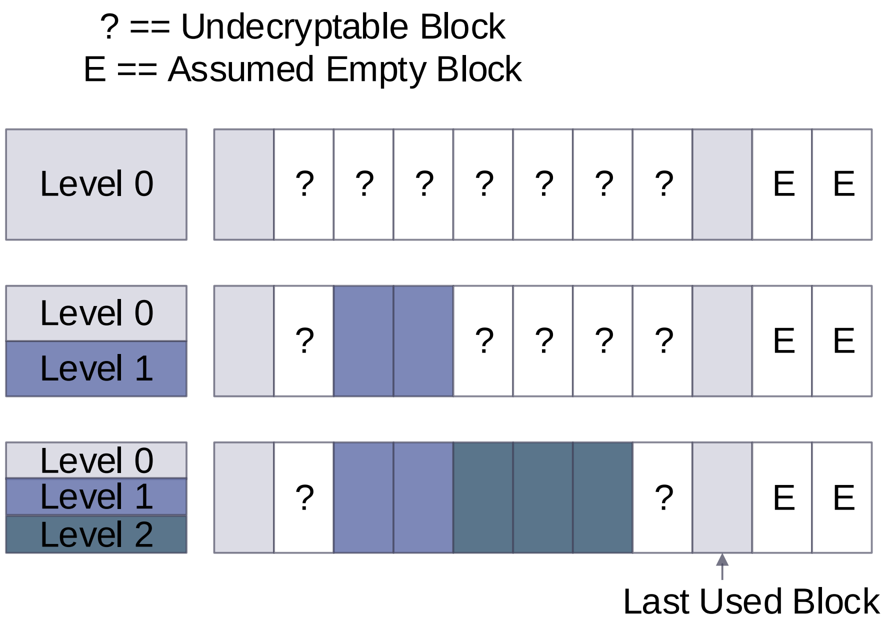

Figure 4.8: A multi-level view of blocks in DEFY.

discussed, an attacker could infer information if sections of the stub storage area remain static. Our recommendation is for the user to randomize the stub storage area after each suspected device inspection. Such a randomization schedule would ensure that even a snapshotting adversary would be unable to determine which levels of the file system were static and which were active. This operation is not required for the single-visit adversary.

A user could employ the alternative strategy of writing data to a high level and letting it remain static thereafter. The snapshot adversary without drive images before the high level writes could be more easily convinced that the file system contained a subset of the actual set of privilege levels.

CHAPTER 5

Evaluation

Testing

We wrote DEFY on top of the YAFFS and WhisperYaffs source code. DEFY was built and tested on a Ubuntu 13.04 machine. That machine was run in a virtual machine with 4GB of memory and a single emulated processor. We used Oracle's VM VirtualBox 4.3.6 to handle the emulation. The host machine ran Windows 8 with an Intel 2.8 GHz quad-core processor. The hosted machine's Linux kernel was modified from Linux 3.8 to allow the loading of unsigned kernel modules–like DEFY. DEFY was tested with the MTD device simulator nandsim [3]. We used nandsim to test DEFY with a number of different NAND configurations. For the testing results below we emulated a 64 megabyte device with 2048 byte pages.

The same setup was used to test WhisperYaffs [7], YAFFS [13], ext3 and ext4. The version of YAFFS that we used lacks a specific version number, but it was committed on the 10th of August, 2013 to the YAFFS repository. Our version of WhisperYaffs was created by merging the alpha version of WhisperYaffs from GitHub with our version of YAFFS. Finally, we used the versions of ext3 and ext4 that were included with the 3.8 kernel.

The version of DEFY that we test here is limited to keeping all of the chunk stubs in memory at all times. As a result, we found that the total file size that DEFY could store was limited to about 2 megabytes. In future we plan to modify DEFY to allow it to remove keys from memory and then intelligently re-fetch them from disk when needed.

Basic Tests

We first ran defy through the Connectathon tool [4]. Successfully passing Connectathon shows that DEFY complies with all standard file system operations. This verification step was not run on the other comparison file systems.

Performance Tests

Performance was not the focus of this file system, but it was a consideration. We chose to test DEFY's performance using IOZone and FFSB. Each tool tests the file system differently. FFSB uses time as the input metric, where IOZone uses data as the input metric.

FFSB

FFSB benchmarks file systems by holding time constant and measuring the volume of disk accesses that are possible in that time [24]. That is, it reads and writes data until a timer expires, instead of reading and writing a finite amount of data. We limited our runs to 1 second due to the higher speed file systems. If the time was set any longer the emulated drive would run out of space. Our results from FFSB are shown in figure 5.1.

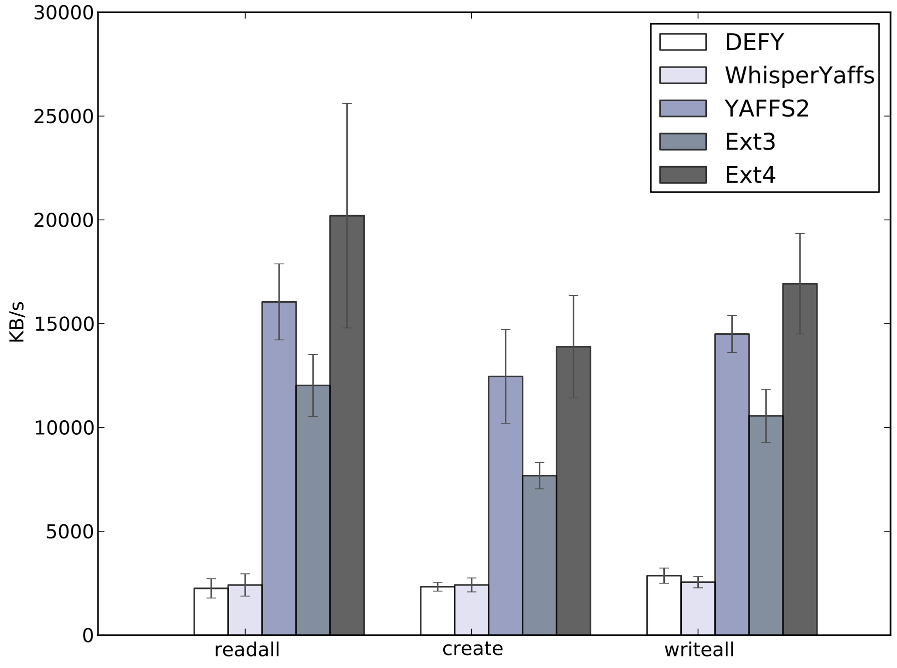

Figure 5.1: The performance of a number of file systems as measured by FFSB.

IOZone

Unlike FFSB, IOZone tests how long it takes to write a set amount of data to a device–perhaps the more conventional benchmarking method. Again, note that we limited the tests to 2 megabytes in size. The outputs from these tests can be seen in figure 5.2.

For each test category, IOZone attempts that operation on a number of uniformly sized files up to some maximum size. For example, for a 64 kilobyte sized file test it would try to write sixteen 4 kilobyte files, eight 8 kilobyte files, and so on up to one 64 kilobyte file. For each category, we averaged all of the transaction speeds within each file system. An example of one of IOZone's outputs is shown in table 5.1.

The write category measures both the speed at which a new file is created and written to. This is dissimilar to FFSB, which measures those indicators separately.

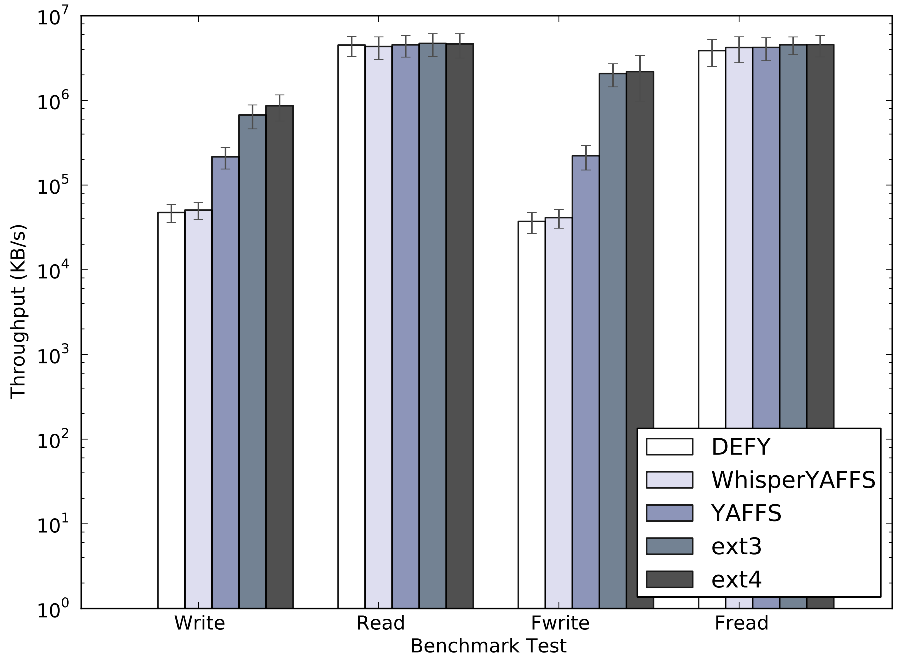

Figure 5.2: The performance of a number of file systems as measured by IOZone (note the log scale).

The read category is simply the speed at which files were read. Fwrite and Fread are the same as write and read except that they use buffered library calls. Each of these categories is discussed in more depth on IOZone's website and in their documentation [29].

Discussion

We tested DEFY against the other aforementioned file systems assuming that it would be slower. What we were interested in was how much introducing deniability would slow it down. In particular, we were interested in how DEFY would compare to WhisperYaffs, its most realistic competitor. Both WhisperYaffs and DEFY encrypt each chunk as it is written to disk and decrypt it after it is read from disk. DEFY then adds on the additional tasks of stub creation and management. YAFFS is obviously

Write Size File Size	4	8	16	32	64	128	256	512
64	28021	28382	26501	33559	38116			
128	26084	26772	34820	26348	28809	31794		
256	29297	29439	27482	30527	28070	28315	27853	
512	27563	19444	18056	26590	26924	29377	26788	27938
1024	27368	25742	19303	28090	27816	27804	28331	26732
2048	28195	27689	24718	27290	28114	23215	24843	24302

Average Speed = 27.4 KB/s and Standard Deviation = 3.42 KB/s

Table 5.1: IOZone write benchmark result for the DEFY file system

the root of both WhisperYaffs and DEFY, but it does not include any encryption so it is understandably much faster. We also included results from the Ext3 and Ext4 journaling file systems. Ext4 is used with increasing frequency on mobile devices despite its lack of sensitivity for flash memory.

Emulation both on the behalf of the machine and the flash device doubtlessly introduced inconsistencies in our performance data. Nonetheless, we compared all of the file systems on the same system with the same amount of load so that they would all be similarly encumbered.

FFSB's output matches what we would expect from WhisperYaffs, YAFFS, ext3, and ext4. We expected YAFFS, ext3, and ext4 to be faster than either WhisperYaffs or DEFY. We also expected WhisperYaffs and DEFY to be comparable given that they both have an encryption step for every chunk that is written to the device. Finally, DEFY should be slightly slower than WhisperYaffs due to its in-memory stub operations.

IOZone, however, produced less agreeable results. Due to our file-size restrictions, IOZone was operating at its minimum accepted test size. It also produced a warning that some of its output would be unreliable. Indeed, the read tests seemed to lack resolution as they were unable to distinguish between ext3 and ext4. Because of this inconsistency, we chose to discard IOZone's read tests from our final evaluation, but we still included them here for the sake of completeness. The write tests, however, were closer to FFSB's output. They ranked DEFY as being twice as slow as WhisperYaffs on average. They also ranked both WhisperYaffs and DEFY as being much slower than the other three file systems, which was expected.

Combining the data from all of FFSB's tests and IOZone's write tests we can say that FFSB runs one to two times slower than WhisperYaffs. In our opinion this is a more than acceptable trade off for the added benefits of deniability.

CHAPTER 6

Future Work

There are a number of ways that DEFY could be modified for future use. We discuss a few of them below.

Stub Caching

We maintained the YAFFS design of storing all metadata in memory while adding greatly to the size of that metadata with stubs. As a result, DEFY can only handle files up to 2 megabytes in size. This number could be drastically increased by decreasing the number of stubs in memory. DEFY already stores all of the needed stubs on the device and in the stub storage area, so the only necessary modification is adding a better memory manager that manages which stubs are in memory at all times. Unfortunately, such a change would decrease the performance of DEFY, but it would also make it much more practical for today's larger file sizes.

Internal Deniable File System

We would have liked to remove the need for an external file system that supported secure deletion. This modification is straightforward and was left out due to time constraints. It would require that the top-level stub file was written to special stub blocks on the file system. Each time a stub was changed, its containing block should be erased and re-written in a new location. The old block would then need to be re-written with encrypted random data, just like any other unused block in the file system. This would ensure that old key data was destroyed. The choice of the next block to use for a stub block could simply align with the next available block on the device—provided by the allocator. This would align with our current deniability model and maintain even device wear.

One drawback of this system is that the blocks that were used to store top-level stubs would experience slightly more wear than normal blocks. Each time the stub block is rotated forward, it must be erased and rewritten with random data, which is one additional write cycle compared to normal data blocks. We must rewrite the stub blocks because they are encrypted with the level key only.

Other Device Types

The assumptions necessary for a deniable flash file system are more strict than those for a block-device file system. Indeed, DEFY could be modified to work with block devices. Such a file system would not receive the benefit of even wear, but it would be deniable. This modification would be less beneficial if the underlying device was a disk drive. Disk drives are not designed for random access so the large number of holes that DEFY creates would be detrimental to read times. This modification would be useful if the underlying device was still flash memory. An example use case is USB sticks, which use a block device layer, but are still flash memory. Note that in such a situation, DEFY would still need a file system with secure deletion to store the key file.

CHAPTER 7

Related Work

Other Deniable File Systems

Anderson, Needham, Shamir

The idea for a plausibly deniable file system was introduced by Anderson *et al.* [15] in 1998. They presented two competing constructions. The first was steganographic, it hid data inside of other valid data. The second hid data inside of a large amount of random data. Time has shown the second option to be more popular, it is emulated by the majority of deniable file systems–including DEFY.

Their first design hid data within a valid-looking set of ("cover files"). Each file in the file system could be reconstructed by XORing certain parts of the cover files together. How the cover files were superimposed on each other was determined by a matrix key. For this to work properly, the cover files needed to be much larger than the hidden files.

The second design placed valid data inside of a large volume of random data. Each block of data was encrypted before it was added to the device so that it would blend in with the random data. The exact block number where data is stored is dependent on a pseudo-random number generator that was seeded with the file name of the data. Each block of data is then encrypted with a level key. Thus, to access a block of data the user must provide a file name and the level key.

Each new block that is added to the device could overwrite existing data. The probability of collision is the same as the probability of a collision in the birthday problem [1]. To mitigate the problem, Anderson *et al.* duplicate data into multiple blocks.

McDonald and Kuhn

McDonald and Kuhn released a file system called StegFS. It was based closely on Anderson's second construction and the ext2 file system [27]. They did away with purely pseudo-random block placement and added a block allocation table. This allowed users to access their files without needing to store file names. The potential for overwriting still exists in this file system and the authors again chose to mitigate it using duplicated blocks.

Pang, Tan, and Zhou

Pang, Tan, and Zhou, introduced a deniable file system with an unencrypted global bitmap of used blocks in the file system [30]. They also named it StegFS. Their bitmap does leak information about the maximum amount of data that could be in the file system even if no keys are available. In return, this file system ensures that files cannot be overwritten. The authors augment their deniability using dummy blocks and abandoned blocks. As the block names suggest, the file system never accesses abandoned blocks after they are allocated and it randomly updates dummy blocks with random data.

Skillen and Mannan

Mobiflage is a deniable file system for Android devices that was introduced by Skillen and Mannan [40]. Their file system hides the deniable drive in a standard encrypted file system. They introduced deniability by placing the start to the deniable drive somewhere in the third quarter of the drive's address space. The exact location was based on a hash of the encryption key.

The system works at the block device layer; thus, when used with flash storage, the write leveling systems may potentially undermine the deniability of the hidden filesystem, revealing recent activity on the hidden portion of the drive. Further, their system only supports one deniability level and cannot be trivially extended to provide additional levels.

Comparison

It is our belief that DEFY provides a number of features that have not previously been explored in deniable file systems. We compare the features of DEFY to the aforementioned other deniable file systems in table 7. We also discuss the features that we are comparing against, except for those that we already discussed in the background section.

Number of Deniable Levels

The concept of a deniability level was introduced in previous deniable file systems implementations [15, 27, 30]. A deniability level is a collection of files that form a

	DEFY	Skillen [40]	Pang [30]	McDonald [27]	Anderson: 1 [15]	2 [15]
Single-Visit Strong	✓	✓	✓	✓	✓	✓
Snapshot Strong	✓		✓			
Arbitrary No. of Levels	✓		✓	✓	✓	✓
Authenticated Encryption	✓					
Zero Data Loss	&	✓	✓	*	*	✓
Wear Leveling	✓					

(*) these filesystems experience low but probabilistic data loss

(&) DEFY may experience data loss in specific circumstances (see 4)

Table 7.1: Feature comparison between DEFY and previous deniable file systems.

sensitivity equivalence class (*e.g.* love letters vs. trade secrets). Here, as in previous work, deniability levels form a total order: $\ell_0 \leq \ell_1 \leq \ldots \leq \ell_h$. A user has some secret password to reveal all files at a chosen deniability level. Following a convenience established in previous work, when revealing a level, all lower levels should also revealed. The system should support an arbitrary number of named deniability levels that can be created dynamically, rather a fixed number of levels or levels created exclusively during initialization–both of which impose an artificial restriction on the system's use and leave a user vulnerable to coercion by an intelligent adversary.

Overwriting

Data loss occurs when hidden data (unrevealed data at a high deniability level) is overwritten because the file system is mounted at a lower level—an unfortunate, but unavoidable characteristic of any deniable file system. The ability or the probability of a file system overwriting data is entirely dependent on its design. One strategy to prevent overwriting is to maintain a global list of memory blocks that are free for writing (not in use by any higher or lower levels); a strategy similar to this is employed by Pang *et al.* [30]. Alone, this strategy undermines plausible deniability: a single-view adversary learns which blocks are in-use across the system, revealing if hidden levels exist. The remedy in Pang is to create abandoned blocks, or blocks that are falsely marked as in-use. This creates plausible deniability, at the expense of *permanently* sacrificing capacity. Anderson *et al.* [15] prevent data loss in their system through block replication, similarly suffering a significant overhead to prevent data loss. While the capacity of NAND drives is increasing and prices decreasing, the cost-per-byte for flash memory still almost double that of hard disk devices, limiting the appeal of solutions with high storage overheads. What's more, storage devices that employ wear-leveling preclude file systems from modifying data in place or at completely random locations. This entirely excludes data recovery strategies based on random placement of replicas, or using recovering overwritten blocks from n-out-of-m threshold-based error correction codes.

Wear Leveling

NAND flash has a limit to the number of times data can be written to a block before it fails. To delay failure, many devices implement *wear leveling*, in which all writes are systematically written to new locations, preventing some blocks from failing far earlier than others. This has implications for both encrypting and deniable file systems: wear-leveling mechanisms may persist old version of encrypted data, providing an adversary with a time-line of changes made to disk, and thus, an ability to differentiate between claimed and actual disk activity. Wear-leveling undermines any file system whose security is predicated on the ability to overwrite data. Any secure file system designed for flash-based storage should be secure and compatible with drives that either do or do not manage their own wear leveling.

Easily Deployable

To have the broadest impact, a deniable file system should be easily distributable and compatible with popular operating systems (*e.g.* Android and Linux). Using a loadable kernel module to extend the existing kernel allows for systems enhancements without rebuilding from source.

CHAPTER 8

Conclusion

In this work we proposed the design for a new deniable file system dubbed DEFY. DEFY is strong against both the snapshot and the single-view adversaries. It provides wear-leveling, secure deletion, authenticated encryption, multiple deniable levels, and mitigated data loss. The deniability of DEFY resides in the undecryptable blocks that are created by the combination of the forward-writing nature of a log-structured file system and normal user interactions. We built a prototype version of DEFY and found that it performed comparably to WhisperYaffs—its closest competitor—when benchmarked with both the FFSB and the IOZone test suites. We believe that this is a valuable contribution to the existing collection of deniable file systems, and we propose that it is even strong enough to defend against the snapshot adversary. To our knowledge there is only one other deniable file system that reaches this level of security.

BIBLIOGRAPHY

[1] Birthday problem. http://mathworld.wolfram.com/BirthdayProblem.html.

[2] Usb flash wear-leveling and life span: Frequently asked questions, 2007. http://web.archive.org/web/20071013150729/http://www.corsair.com/_faq/FAQ_flash_dri

[3] UBIFS - A UBI File System, October 2008. http://www.linux-mtd.infradead.org/doc/ubifs.html.

[4] Connectathon, 2010. http://www.nfsv4.org/.

[5] GitHub: WhisperSystems/RedPhone. `http://goo.gl/Mmz9s`, 2012.

[6] GitHub: WhisperSystems/TextSecure. `http://goo.gl/3qoV8`, 2012.

[7] GitHub: WhisperSystems/WhisperYAFFS: Wiki. `http://goo.gl/Qsku4`, 2012. https://github.com/WhisperSystems/WhisperYAFFS/wiki.

[8] Martus case studies: The global human rights abuse reporting system, 2012. https://www.martus.org/resources/case_studies.shtml.

[9] Guardianproject/chatsecure:android, 2014. https://guardianproject.info/apps/chatsecure.

[10] Guardianproject/orbot, 2014. https://guardianproject.info/apps/orbot/.

[11] Lookout/lookout mobile security, 2014. https://www.lookout.com/.

[12] Truecrypt - free open-source disk encryption software for windows, mac os x, and linux, January 2014. http://www.truecrypt.org/.

[13] Yaffs (yet another flash file system), 2014. http://yaffs.net/.

[14] R. Anderson. The dancing bear – a new way of composing ciphers. In *Proceedings of the International Workshop on Security Protocols*, April 2004.

[15] R. J. Anderson, R. M. Needham, and A. Shamir. The Steganographic File System. In *Information Hiding*, pages 73–82, 1998.

[16] M. Bellare and C. Namprempre. Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology - Asiacrypt'00 Proceedings*, volume 1976. Springer-Verlag, 2000. Lecture Notes in Computer Science.

[17] M. Blaze. A cryptographic file system for UNIX. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[18] M. Blaze. High-bandwidth encryption with low-bandwidth smartcards. In *Fast Software Encryption*, volume 1039, pages 33–40, 1996. Lecture Notes in Computer Science.

[19] M. Blaze, J. Feigenbaum, and M. Naor. A formal treatment of remotely keyed encryption. In *Advances in Cryptology – Eurocrypt'98 Proceedings*, volume 1403, pages 251–265, 1998. Lecture Notes in Computer Science.

[20] Y. Dodis and J. An. Concealment and its applications to authenticated encryption. In *Advances in Cryptology – Eurocrypt'03 Proceedings*, volume 2656, 2003. Lecture Notes in Computer Science.

[21] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security Symposium*, 1996.

[22] M. Jakobsson, J. Stern, and M. Yung. Scramble all. Encrypt small. In *Fast Software Encryption*, volume 1636, 1999. Lecture Notes in Computer Science.

[23] B. Kaliski. Pkcs #5: Password-based cryptography specification version 2.0, 2000.

[24] G. T. R. Keshava Munegowda, Sourav Poddar. Ffsb and io-zone: File system benchmarking tools, features and internals, 2012. http://elinux.org/images/f/f7/FFSB_and_IOzone-_File_system_Benchmarking_Tools,_Features_and_Internals.pdf.

[25] J. Lee, S. Yi, J. Heo, S. Y. Shin, and Y. Cho. An Efficient Secure Deletion Scheme for Flash File Systems. *Journal of Information Science and Engineering*, 26:27–38, 2010.

[26] C. Manning. How YAFFS works, 23 May 2012. Available at http://goo.gl/OMdja.

[27] A. D. Mcdonald and M. G. Kuhn. StegFS: A steganographic file system for Linux. In *Information Hiding*, 1999.

[28] J. Mull. How a syrian refugee risked his life to bear witness to atrocities, March 2012. Toronto Star Online; posted 14-March-2012.

[29] W. Norcott and D. Capps. IOzone filesystem benchmark. http://www.iozone.org/.

[30] H. Pang, K. lee Tan, and X. Zhou. StegFS: A Steganographic File System. In *Proceedings of the International Conference on Data Engineering*, 2003.

[31] Z. N. J. Peterson, R. C. Burns, J. Herring, A. Stubblefield, and A. D. Rubin. Secure Deletion for a Versioning File System. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 11–11, 2005.

[32] J. Reardon, D. Basin, and S. Capkun. Sok: Secure data deletion. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 301–315, Washington, DC, USA, 2013. IEEE Computer Society.

[33] J. Reardon, S. Capkun, and D. Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the USENIX Security Symposium*, pages 333–348, Berkeley, CA, 2012. USENIX.

[34] Reporters Without Borders. Internet enemies, 12 March 2012. Available at http://goo.gl/x6zZ1.

[35] R. L. Rivest. *All-or-Nothing Encryption and the Package Transform.* 1997.

[36] P. Rogaway, M. Bellare, J. Black, and T. Krovet. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 196–205, November 2001.

[37] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *Operating Systems Review*, 25:1–15, 1991.

[38] S. Schmitt, M. Spreitzenbarth, and C. Zimmermann. Reverse engineering of the Android file system (YAFFS2). Technical Report CS-2011-06, Friedrich-Alexander-Universität Eriangen-Nürnberg, 2011.

[39] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the Winter USENIX Technical Conference*, pages 307–326, January 1993.

[40] A. Skillen and M. Mannan. On implementing deniable storage encryption for mobile devices. In *Proceedings of the Network and Distributed System Security Symposium*, February 2013.

[41] S. Swanson and M. Wei. Safe: Fast, verifiable sanitization for ssds, 2010.

[42] C. P. Wright, M. C. Martino, and E. Zadok. Ncryptfs: A secure and convenient cryptographic file system. In *Proceedings of the USENIX Technical Conference*, pages 197–210. USENIX Association, 2003.

A: Old Design

We document our original vulnerable idea for DEFY below with the hope that it will not be reattempted in the future.

Our initial idea for DEFY was to use randomly placed dummy blocks to conseal data. The idea of a dummy block was inspired by the work of Pang et al. [30]. Apart from that, this version of DEFY is unique. The idea is detailed below along with its vulnerability.

Design

The chunk allocator in YAFFS assigns writable chunks out of a single block that is currently in the "allocating" state. In this design, each time a block was allocated a random number (including zero) of chunks would be assigned to each active–or currently opened–level of the file system, including a dummy level. Any chunks assigned to the dummy level were filled with random data. We allowed the over levels to use chunks from that block based on their random allocation. The allocator ordered chunks in the block based on which level they were assigned to, see figure 8.1. If a level needed more chunks, then a new block had to be allocated, which led to multiple blocks being open for allocation at the same time.

Vulnerability

We believed that randomizing and ordering blocks would create the required obscurity and security for a deniable file system. The user and adversary would always expect to find a random number of dummy chunks in any one allocation block, privileged data could then hide in that margin of expected dummy chunks.

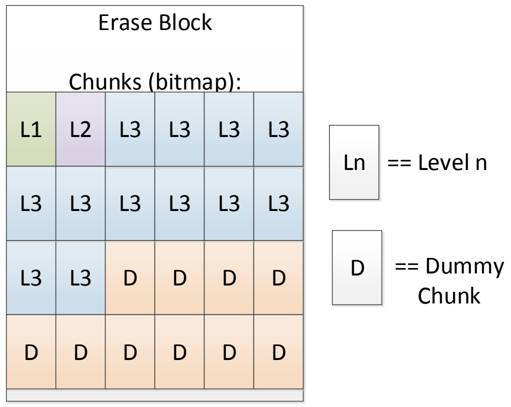

Figure 8.1: A possible allocation of chunks in an erase block.

The problem is, if an adversary had a sufficient number of blocks she can look at the average distribution of blocks to any one level. The adversary would expect to find an even distribution of the known levels in the file system, if something else was found, it would be unpleasant for the user. A potential solution to this is to allocate chunks for all possible levels, but that would severely diminish the available disk space. We thus determined that this problem could be better solved through other deniability mechanisms.