DEFY: A Deniable, Encrypted File System for Log-Structured Storage

Timothy M. Peters
Cal Poly, San Luis Obispo
tipeters@calpoly.edu

Mark A. Gondree
Naval Postgraduate School
mgondree@nps.edu

Zachary N. J. Peterson
Cal Poly, San Luis Obispo
znjp@calpoly.edu

Abstract—While solutions for file system encryption can prevent an adversary from determining the contents of files, in situations where a user wishes to hide the *existence* of data, encryption alone is not sufficient. Indeed, encryption may draw attention to those files, as they may likely contain information the user wishes to keep secret. Consequently, adversarial coercion may motivate the owner to surrender their encryption keys, under duress. This paper presents DEFY, a deniable file system following a log-structured design. Maintaining a log-structure is motivated by the technical constraints imposed by solid-state drives, such as those found in mobile devices. These devices have consequential properties that previous work largely ignores. Further, DEFY provides features not offered by prior work, including: authenticated encryption, fast secure deletion, and support for multiple layers of deniability. We consider security against a snapshot adversary, the strongest deniable filesystem adversary considered by prior literature. We have implemented a prototype based on YAFFS and an evaluation shows DEFY exhibits performance degradation comparable to the encrypted file system for flash, WhisperYAFFS.

I. INTRODUCTION

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 Google Play store where there are a number of privacy-enhancing technology apps, including: ChatSecure [3] (secure texting), WhisperYAFFS [51] (an encrypted file system), RedPhone [49] (encrypted calls), TextSecure [50] (secure texting), Orbot [4] (tor for mobile), Lookout [5] (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 human rights violations out of Syria. He lacked any data protection mechanisms and instead hid a micro-SD card in a wound on his arm [29]. 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 these data [2]. In light of the control exerted by the government over the Internet within Burma [37], ND-Burma activists carry data on mobile devices, risking exposure at checkpoints and border crossings. Using a way to hide the encrypted data such that inspectors cannot reasonably infer sensitive data exist on the device, risk to activists and witnesses can be lessened.

A promising approach to securing data under these conditions is to employ a class of file system known as *deniable file systems*. Deniable file systems mask information about stored data, and allow a user to plausibly deny any storage artifacts on their device, typically by encrypting data with different keys based on the sensitivity of the data. Unfortunately, all known methods to provide deniability from previous designs are inapplicable when applied to flash-based storage devices. In particular, for flash media, strategies that require in-place modification of blocks are unavailable, due to wear-leveling requirements and the special handling required for erasures and writes in NAND flash.

In this paper we present DEFY, the **D**eniable **E**ncrypted **F**ile System from **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 introduce unique challenges to plausible deniability, requiring us to depart non-trivially from the designs of prior deniable file systems. In particular, hardware-implemented wear leveling essentially forces DEFY to embrace a log-structured design. The DEFY file system provides a number of features not offered by prior work:

- it features a generic design, adaptable to other settings requiring deniability while maintaining a log-structure;

- it supports an arbitrary number of user-defined deniability levels that can be created or removed from the system, dynamically;

- it is the first encrypted file system for mobile devices providing authenticated encryption;

- it provides a fast and efficient mechanism to securely delete data, allowing individual files or the entire file system to be deleted in bounded time. What's more, it is the first file system to provide secure deletion of prior allocations by policy;

- it is designed to be resistant against the most powerful adversary considered by prior work, a snapshotting adversary.

DEFY's design is a significant departure from previous deniable file systems, which require strict control over block placement on the device, and whose security guarantees do not hold when the underlying media re-maps their writes. For example, for storage using a hardware flash translation layer (FTL), prior designs are only secure under the additional assumption that the media is effectively tamperproof, *i.e.* the hardware controller implementing the FTL cannot be bypassed during adversarial analysis (revealing past writes). Alternatively, when no FTL is performed, those systems ignore the constraints of the underlying media, *e.g.*, they do not wear-level appropriately. In contrast, DEFY embraces a logical log-structure so that its security guarantees hold, especially when the underlying media is also written in a log-structure. Indeed, DEFY solution is generalizable, and can be used with devices employing hardware FTL (without additional assumptions), with those requiring FTL logic be implemented by the file system, or with those requiring no FTL logic at all.

When DEFY is used with only a single deniability level, it acts like an encrypted file system with additional features (secure deletion, authenticated encryption) appropriate for mobile devices, making it attractive beyond deniability. We provide a prototype implementation of DEFY and remark on the system's performance, showing performance comparable to WhisperYAFFS, the encrypted file system for flash storage.

II. RELATED WORK

Anderson *et al.* propose the first file system with the security property of plausible deniability [9]. They present schemes demonstrating two alternate approaches: hiding blocks within valid-looking data ("cover files"), and hiding blocks within random data. DEFY follows this second basic approach. Writing new data has the possibility of over-writing data at unrevealed levels. Anderson *et al.* use block replication and random block placement to reduce the probability of overwriting,

McDonald and Kuhn describe StegFS, an implementation based on adapting Anderson *et al.*'s construction to the $ext2$ file system [28]. They use a block allocation table to track files, rather than random block placement.

Pang, Tan, and Zhou describe a different implementation, also called StegFS [33]. Their implementation uses an unencrypted global bitmap to ensure blocks are not accidentally overwritten. To ensure deniability, "dummy blocks" are occasionally written; these explain blocks apparently in-use but otherwise unreadable by the file system.

Gasti *et al.* describe DenFS, a FUSE leveraging cloud storage and providing deniability in the event the cloud service becomes compromised [22]. DenFS uses cover files and deniable encryption [16] to protect remote data. In DenFS, the adversary may intercept messages and request remote files be revealed, but it cannot seize arbitrary snapshots of local storage (which may reveal local caches, the database of pre-generated cover files, *etc.*). Their model is not appropriate for threats associated with seizure of mobile devices, as considered here and in other prior work.

Skillen and Mannan describe Mobiflage, a deniable filesystem for Android devices [46]. Their system hides a drive image in the standard encrypted file system, placing it at a random location, somewhere in the third quarter of the drive's address space. This is similar to the design of a "hidden volume" under TrueCrypt [6]. Efforts to port TrueCrypt to mobile platforms also follow this pattern. Each of these systems work at the block device layer or higher, ignoring the unique properties of flash storage; thus, the log-structure below this layer may potentially undermine the deniability of the hidden filesystem above it, revealing recent activity on the hidden portion of the device. Further, these systems lack support for more than one deniability level, and are not trivially extensible to handle this feature.

WhisperYAFFS is a system providing full disk encryption on flash devices [51]. It provides only confidentiality without authenticity and, unlike DEFY and other deniable file systems, does not provide plausible deniability; in particular, its use of plaintext block sequence numbers trivially leaks the history of block updates.

III. BACKGROUND

Pushed by demand from the growing mobile device market, solid-state memory has become a popular alternative to hard disk drives, due to its small power footprint, lack of moving mechanical parts and high speed. The evolution of flash has largely 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 denser (bytes per die) than it predecessors, EEPROM and NOR technologies. Current NAND chip sizes are as large as 256GB.

NAND offers random-access reads and writes at the *page* level, while erasure occurs at the *block* level. For example, an 8GB NAND device with 2^{12} blocks can write a 4KB page but must erase at the granularity of a 256KB block. Once pages are programmed, they must be erased before they are written again; this is called the *program-erase cycle*. A per-page Out-of-Bound (OOB) area holds metadata and error correction codes for the page's data.

Flash memory degrades after many program-erase cycles, becoming unreliable after 10,000–100,000 cycles. Many solid-state drives employ *wear leveling* to extend their lifespan

within this constraint: drivers attempt to disperse erase/write traffic to avoid media wear. In *dynamic wear leveling*, data is written to locations based on availability and a least-written count. In *static wear leveling*, some existing, under-utilized (static) block may be moved to distribute wear on the device during a page write request. Most devices implement dynamic wear leveling, for its simplicity and speed.

Flash devices can be accessed using Linux's memory technology device (MTD) subsystem interface, essentially providing a "raw" interface to NAND flash devices. An MTD provides a consistent mapping from logical blocks to physical blocks, it provides no write leveling, and thus nothing to prevent cell overuse. The unsorted block images (UBI) interface builds on MTD, providing an abstraction comparable to the Linux Logical Volume Manager for flash storage. UBI tracks logical blocks in a data structure, deciding to re-map logical blocks based on use, implementing wear leveling. A flash translation layer (FTL) can be built on top of UBI, providing a simplified, block-level interface for flash, in exchange for a loss of low-level control over data placement and strict overwrites.

A. YAFFS Overview

YAFFS is a file system designed for use with NAND flash memory. Due to its simplicity, portability, and small memory footprint, YAFFS has been used as the default file system in many mobile devices, including the Android operating system. YAFFS is a true log-structured file system [41], [44] in that write requests are allocated sequentially within the logical address space. Its design is largely motivated by a desire to integrate with device-level wear leveling. Next, we briefly summarize YAFFS's design; for a more thorough description, we direct readers to Manning [27] and related resources, *e.g.*, Schmitt *et al.* [42].

The unit of allocation is the page (called a *chunk* in YAFFS terminology), ranging from 512-bytes to 32KB in size. The unit of erasure is the block, each block being composed of 32–128 pages, depending on the NAND block capacity. YAFFS uses the OOB space provided by a flash device to store page metadata and an error correction code.

There are two versions of YAFFS: YAFFS1 and YAFFS2. The key distinctions between these are: (1) YAFFS1 is 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. This paper refers to the YAFFS2 design, and 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 associated object, including its name, its size, and location its pages. A directory's header contains the location of headers for its children (files and subdirectories).

B. Writing in YAFFS

Write requests are divided into pages, allocated and written sequentially following the leading edge of the log (the last page written). If the leading edge is the last page of a block, YAFFS searches for the next block past the leading edge that is not full.

Every page is assigned a *sequence number*, stored in the OOB section of memory. The sequence number is monotonically increasing, *i.e.*, the last page written has the highest value, making it the new leading edge of the log. The leading edge marks the starting point for the system when searching for the next page to allocate.

When a page is updated, its corresponding object header is updated to reference the new page. When an object header is updated, it too will be written to a new page, and the object containing it (*e.g.* a directory) will also be updated. Therefore, when a page is modified, the directory path above its object, up to and including the root, is modified on disk.

C. Mounting in YAFFS

YAFFS supports special objects known as *checkpoints*. These commit information about the state of the file system to the drive. On mount, YAFFS searches for the most recent checkpoint, using it to reconstruct in-memory data structures. In the absence of a checkpoint, YAFFS will scan the entire disk, creating a list of blocks and sorting these by sequence number. Then, in descending order, it examines the contents of each block: invalid pages are ignored and valid pages are added to an associated 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.

D. Garbage Collection in YAFFS

Since YAFFS is a log-structured file system, a page is never updated in place. Thus, when a page is updated, an older version of the page likely exists elsewhere on disk. Since NAND requires a page be erased before it can be written and offers only block-level erasure granularity, a block may contain many obsolete pages that cannot be reclaimed until all pages in the block are obsolete.

YAFFS supports two, heuristic modes of garbage collection: *passive* and *aggressive* garbage collection. In general, YAFFS garbage collection proceeds in the following fashion. The system scans the disk looking for a "dirty" block, *i.e.* a block with "few" valid pages. The definition of "few" depends on the garbage collection mode. During normal operation, the collector considers a block to be "dirty" if the number of valid (active) pages in the block is below some threshold. On startup, during *passive garbage collection*, this threshold is lowest: if no more than four pages are valid, then the block is dirty. On each unsuccessful scan, the threshold is increased, beginning with four but never going beyond half the pages in a block. When every block is more than half full, the system will switches to *aggressive garbage collection*, where a block with *any* dirty page is considered dirty. Once a block is identified for collection, YAFFS re-writes its valid pages to the leading edge and erases the dirty block, making it available for writing.

IV. SECURITY MODEL

Before describing the design of DEFY, we introduce our adversarial model and security goals.

A. Adversaries

A secure, deniable file system hides the existence of information from an adversary. It conceals all indication as to whether or not there are hidden files or directories, at the file system level. It does not, however, enforce a system-wide information flow policy. Czekis *et al.* demonstrate how it is possible to infer the existence of hidden files using the content of revealed files, *e.g.* indices generated by services for desktop search [17]. We assume users, the OS and applications use the file system appropriately.

We define two types of passive adversaries for deniable file systems: those with one-time access to the device (*single-view adversaries*) and those with periodic access (*snapshot adversaries*). In this context, an adversary's "access" yields a full copy of the disk and a complete description of the file system (*e.g.* through a copy of its source code). Further, the adversary is allowed to force the user to reveal some set of hidden files. As with previous work, the adversary cannot access the device's RAM contents nor capture the running state of the device while outside of the adversary's immediate possession, *e.g.* using a Trojan to implement a, so called, "Evil Maid" attack [18], [43] or extracting cryptographic keys from RAM using a cold boot attack [30]. To help ameliorate these threats, DEFY could be enhanced with an interface to immediately zero RAM data structures (including cryptographic keys) in an emergency situation, *i.e.* a quick lockout feature that may result in limited data loss.

The single-view adversary is one that is able to access the file system and its user only once. This adversarial model describes many natural scenarios: those in which the device is stolen, or it is confiscated and the user detained for questioning. The snapshot adversarial model describes scenarios in which access to the device is granted at distinct points in time, and file system images are collected. For example, upon entering and exiting a guarded facility or at a border-crossing. The snapshot adversary may then use differences in the collected images to identify changed data blocks on the device.

Both models include the ability of adversaries to use various means (*i.e.*, threats and physical violence) to compel the user to reveal some set of files. Both models allow the adversary to perform advanced computer forensics on the disk image, use password cracking programs, employ statistical tests, *etc.* These models subsume all previous deniable file system adversarial models in the literature.

B. Security Definitions

A deniable file system offers *plausible deniability* if the adversary has no means of proving that the user has withheld data, beyond what she has chosen to reveal. Alternatively, the user must be able to convince the adversary that no data has been written to the file system beyond what she has chosen to reveal. In other words, it must be plausible that any unrevealed block on the disk contains no valid data. Additionally, the file system offers *snapshot resistance* if it has plausible deniability, even in the presence of a snapshot adversary. In particular, the user may have written data to the disk between snapshots, and the adversary can determine which blocks have been modified between accesses. It must be possible that any unrevealed block (modified or not) contains no data.

V. Design Requirements

Before we describe the details of our design, we provide an overview of the requirements that we believe should guide the design of any secure, deniable file system.

Deniability Levels: The concept of a deniability level was introduced in previous deniable file systems implementations [9], [28], [33]. A deniability level is a collection of files that form a 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 be revealed. The system should be flexible enough to accommodate the dynamic creation of new deniability levels, rather than pre-specifying and defining the total set of levels at initialization. An implementation may, of course, elect to restrict users to some large, fixed number of levels by default; however, we believe this should not be a restriction imposed by design.

Secure Deletion: Providing secure deletion is complementary to the setting of deniability. Secure deletion assures that a deleted object is permanently inaccessible, even if the device and keys are later revealed to an adversary. We desire secure deletion to be efficient, deniable and granular. Granular deletion means deleting the entire file system or a set of files is as complex as deleting an individual file. Finally, to preserve deniability, deleted data should not appear to be deleted: it should be indistinguishable from both data unused by the system and unrevealed data.

Garfinkel and Shelat [21] survey methods to destroy digital data, and identify two techniques that leave storage devices usable after erasure: secure overwriting and cryptographic key disposal. In secure overwriting, old data are overwritten with new data such that the old data are irrecoverable. Gutmann [23] gives a technique for magnetic storage devices that takes 35 synchronous passes over the data in order to degauss the media. (Fewer passes may be sufficient.) Techniques designed to securely delete data from hard disk drives have been shown to be ineffective for flash-based storage [48].

For systems that employ encryption, Boneh and Lipton propose that data may be securely deleted by "forgetting" the corresponding encryption key [15]; without the key, it is computationally infeasible to ever decrypt the data again. The actual disposal of the encryption key may involve secure overwriting. Results from Reardon *et al.* [36] and Lee *et al.* [26] indicate that key disposal techniques may be the most appropriate technique for flash storage.

Authenticated Encryption: Authenticated encryption provides confidentiality and data integrity [11]. Confidentiality, of course, is essential for hiding data at unrevealed levels. When users may be compelled to relinquish possession of their mobile device, the benefits of data integrity under the loss of physical security are also beneficial. Authenticated encryption, though, requires message expansion—ciphertext are larger than the original plaintext—which is an obstacle to its integration into legacy file systems. Existing work in cryptographic file systems (*e.g.* [6], [14], [52]) use only unauthenticated block

TABLE I: Feature comparison of deniable file systems.

	DEFY	Skillen *et al.* Mobiflage [46]	Pang *et al.* StegFS [33]	McDonald *et al.* StegFS [28]	Anderson *et al.* Scheme 1, [9]	Anderson *et al.* Scheme 2 [9]
Single-view Resistance	✓	✓	✓	✓	✓	✓
Snapshot Resistance	✓		✓			
Arbitrary No. of Levels	✓		✓	✓	✓	✓
Authenticated Encryption	✓					
Efficient Secure Deletion	✓					
Data Loss Resistance	Load-dependent	✓	✓	Probabilistic	Probabilistic	✓
Wear Leveling Aware	✓					

ciphers, which preserves message size to meet the alignment constraints of block-based storage devices. In practice, additional storage must be used and managed for the extra bits associated with ciphertext expansion.

Minimizing Data Loss: 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. 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.* [33]. 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.* [9] 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 is 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 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. Many devices, then, 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 versions of encrypted data, providing an adversary with a timeline 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 systems to be modified without rebuilding from source.

VI. DESIGN OVERVIEW

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 Table I. The following provides a high-level description of DEFY's main design features.

Deniability Levels: DEFY supports one or more deniability levels, each associated with a *level directory*, and permits the dynamic creation of levels. Each level directory exists under the root directory of the file system. All files for a deniability level are located below its level directory. Each deniability level is associated with a unique name and cryptographic key, derived from a user-provided password. The system maintains no record of what 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, helps to minimize the chance of overwriting (since only unrevealed levels risk overwriting), and follows the conventions of previous work.

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 categorize the sensitivity of their data.

Authenticated Encryption: The two key challenges associated in implementing authenticated encryption in DEFY are: (1) designing a file system that can accommodate the data

Input: Data Page $\langle d_1, \ldots, d_m \rangle$ with page ID id, OOB data d_{oob}, counter x, and per-level keys K_ℓ, M_ℓ
1: $ctr_1 \leftarrow$ PAD-128$(id||x||1)$
2: $c_1, \ldots, c_m, c_{oob} \leftarrow$ AES-CTR$_{K_\ell}^{ctr_1}(d_1, \ldots, d_m, d_{oob})$
3: $\sigma \leftarrow$ HMAC-SHA256$_{M_\ell}(c_1, \ldots, c_m, c_{oob})$
4: $ctr_2 \leftarrow$ PAD-128$(id||x||0)$
5: $x_1, \ldots, x_m, x_{oob} \leftarrow$ AES-CTR$_\sigma^{ctr_2}(c_1, \ldots, c_m, c_{oob})$
6: $t \leftarrow \sigma \oplus x_1 \ldots \oplus x_m \oplus x_{oob}$
Output: Tag t, Page $\langle x_1, \ldots, x_m \rangle$ and OOB x_{oob}

(a) AON Encryption.

Input: Encrypted Page $\langle x_1, \ldots, x_m \rangle$ with page ID id, OOB data x_{oob}, counter x, tag t, per-level keys K_ℓ, M_ℓ
1: $ctr_2 \leftarrow$ PAD-128$(id||x||0)$
2: $\sigma \leftarrow t \oplus x_1 \oplus \ldots \oplus x_m \oplus x_{oob}$
3: $c_1, \ldots, c_m, c_{oob} \leftarrow$ AES-CTR$_\sigma^{ctr_2}(x_1, \ldots, x_m, x_{oob})$
4: $\sigma' \leftarrow$ HMAC-SHA256$_{M_\ell}(c_1, \ldots, c_m, c_{oob})$
5: if $\sigma' \neq \sigma$ return $\bot$
6: $ctr_1 \leftarrow$ PAD-128$(id||x||1)$
7: $d_1, \ldots, d_m, d_{oob} \leftarrow$ AES-CTR$_{K_\ell}^{ctr_1}(c_1, \ldots, c_m, c_{oob})$
Output: Page $\langle d_1, \ldots, d_m \rangle$, OOB d_{oob}

(b) AON Decryption.

Fig. 1: Authenticated encryption/decryption for a page using the all-or-nothing transform in DEFY.

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.

DEFY's encryption scheme is presented in Figure 1. The algorithm takes as input a data page, broken into m, 128-bit messages $(d_1, \ldots, d_m)$, the OOB data (d_{oob}), a unique page identifier (id), a unique global counter (x), a per-level encryption key (K_ℓ) and a per-level MAC key (M_ℓ). The algorithm implements an encrypt-then-MAC scheme: first encrypting the page and OOB data using AES in counter mode (AES-CTR), then MACing the resulting ciphertext using a SHA-based message authentication code (HMAC-SHA256). An additional encryption using AES-CTR using the authenticator as the key is performed to complete an all-or-nothing transform (described later). A tag (t) is created by XOR-ing the ciphertext blocks $(x_1, \ldots, x_m, x_{oob})$ with the authenticator (σ). This small tag is not secret; rather, it is an expansion of the encrypted data and is subject to the all-or-nothing property. The encrypted page $(x_1, \ldots, x_m)$ is written to disk as data, the encrypted OOB data (x_{oob}) is written to the OOB area, and the tag (t) is stored as metadata in the parent object.

The same counter and key pair should never be used for encryption more than once. For the block cipher in counter mode, we extract a unique counter value, padding this to 128-bits in length using some appropriate padding scheme (PAD-128). This value is derived from the page's physical disk address (id) and a global sequence counter (x); both are associated with 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 1–3 of Figure 1a may be combined into a single call of OCB mode [39], which requires only one pass over the data to be made 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 a means for efficient secure deletion. The original AON transform, due to Rivest [38], is a cryptographic function that, given only a par-

tial 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 [11], to make efficient smart-card transactions [12], [13], [24], message authentication [19], and threshold-type cryptosystems using symmetric primitives [8].

Our design implements an encryption-based secure deletion scheme based on Peterson *et al.*'s AON technique for secure deletion of versioned data [34]. The all-or-nothing transform allows any subset of a ciphertext block to be deleted (*e.g.* through overwriting) in order to delete the entire ciphertext; without all ciphertext blocks, the page 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 tag and can be efficiently overwritten to securely delete the corresponding page. 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.

Metadata for DEFY: Metadata in YAFFS have been re-purposed to support authenticated encryption and secure deletion. Every DEFY metadata object supports the storage of tags for its child objects: data pages 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 tag, overwriting the previous tag, securely deleting the old object. As a result of storing a new tag, the parent object is modified. Thus, creating, deleting or modifying an object in DEFY will trigger a *tag cascade* for all directory objects in that object's path, up to the file system root. See Figure 2 for a simplified overview of DEFY's hierarchical metadata design.

Tags for the level directory are collocated in a *tag storage area* (TSA), which is managed separately from the rest of the file system. When the level tags are updated, they are written to a new block, and the previous version is erased and re-written with pseudo-random data. The approach of using a specially-managed area of flash storage to achieve secure deletion is akin to the strategies proposed by Reardon *et al.*
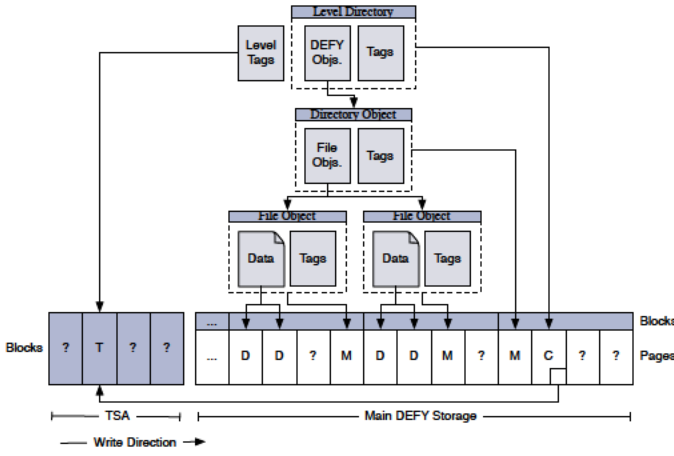
Fig. 2: An overview of the hierarchical structure of DEFY's metadata.

[36] and Lee *et al.* [26], and represents the state of the art for achieving deletion in flash-based storage. DEFY's hierarchy of tags ensures that every cascade will effectively delete all old versions of objects (data and metadata), irrespective of where they are stored on the device (see Figure 3). This property is essential to achieve secure deletion and plausibly deniability in a wear leveled device.

This hierarchical architecture has a number of advantages to achieving fine-grained and efficient secure deletion. Individual objects, be they pages, files, or directories, may be securely deleted by overwriting their corresponding tags and performing a tag cascade. This granularity extends to the level directories, allowing a user to securely delete an entire level or the entire file system by overwriting the tag storage area. And because YAFFS, and thus DEFY, stores all metadata objects in memory, tag cascades only affect in-memory structures, and require no additional device I/O, limiting performance overheads to the computation of new tags.

Checkpoints: Each time DEFY is unmounted, and periodically during file system operation, DEFY writes a *checkpoint*, which represents the current head of the log, and a marker for the last-known consistent state, for a particular level. Checkpoints are common in log-structured file systems [41], [44], including YAFFS [27], and designed to provide an efficient way to return a file system to consistent state after a crash. This is typically achieved by writing all dirty data and metadata to disk followed by special marker containing the current time and pointer to the most current inode map. After a crash, the file system scans the log "backwards" looking for the most recent checkpoint object, as any data that appears before the checkpoint is known to be in a consistent state.

Checkpoints work similarly in DEFY: for each revealed level, a special marker indicating the current head of the log is occasionally written. The contents of a checkpoint include an encrypted level directory object as well as a pointer to the level tags in the tag storage area. A checkpoint is encrypted with its respective level key, allowing it to be decrypted and identified when revealing a level (see Section VII).

Minimized Data Loss: DEFY does not persist to disk any data structure that track used or free pages; any page that is not able to be decrypted during mount is considered free by the page allocator. This gives rise to the possibility of overwriting data stored within an unrevealed level. The primary challenges to preventing overwriting in DEFY stem from its goal to maintain a log-structured file system. Wear leveling devices preclude DEFY from leveraging solutions used in prior work, which require modifying data in place or writing to completely random locations. Instead, DEFY employs three strategies to mitigate data loss that might result from overwriting.

First, DEFY enforces a policy that when a level is revealed, all lower levels are also revealed. Thus, accidental overwriting effects only blocks at higher levels. If a user always reveals the highest level during routine operation (in private), no accidental overwriting occurs. This may be acceptable in many scenarios where the adversary's accesses can be anticipated (*e.g.* border crossings). A user may keep the highest level revealed at all times, only occasionally closing levels when the situation warrants.

Second, DEFY enforces a one-level-per-block policy. Pages marked as free within a revealed block are therefore guaranteed to be free. Further, this simplifies allocation strategies and prevents data loss at a sub-block level.

Third, DEFY writes checkpoints in such a way as to prevent the immediate overwriting of higher levels when the file system is mounted at a lower level. When multiple levels are mounted, checkpoints are written to independent blocks, ordered from highest to lowest privilege (see Figure 4). Thus, data written while mounted at a lower level will avoid checkpoints for higher levels until the log wraps completely.

Page Allocation: DEFY allocates pages using an in-memory, free-page bitmap that is created when the file system is mounted. Any page not able to be decrypted by any revealed level keys is marked as free. DEFY manages only one free-page bitmap, but each block is tagged with a level, ensuring that all pages in allocated to a block are at the same level. If a block has not been fully allocated when the file system is unmounted, it is filled with pseudo-random data generated by our AON transform using an ephemeral key.

VII. FILE SYSTEM OPERATIONS

We summarize the core functions used to maintain the DEFY file system.

Mounting & Revealing Levels: Each deniability level is associated with a unique encryption and MAC key, derived from a password and level name using a password-based key derivation function (PBKDF). In our implementation, we use PBKDF2 [25] using a configurable number of iterations, although scrypt and bcrypt [35] are suitable alternatives.

DEFY maintains no record of what levels exist in the system; therefore, when revealing a level, DEFY must attempt to decrypt every block in the system looking for a valid and current checkpoint for that level. Once identified, the level directory is decrypted and mounted in the file system name space. Remaining objects can be decrypted lazily, on demand.
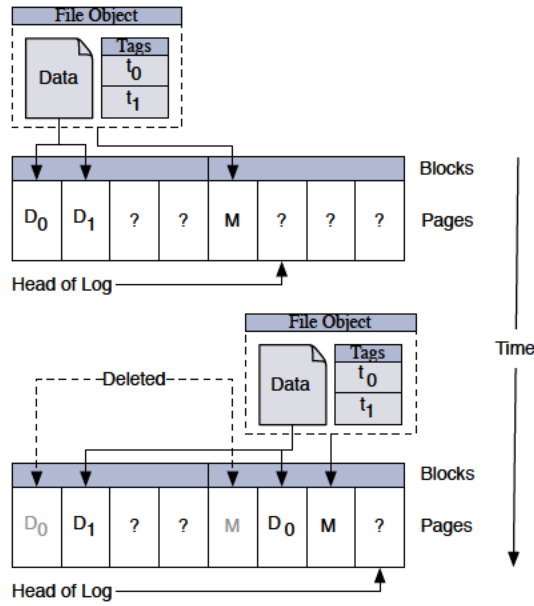
Fig. 3: A page-level view of a file being updated. In this example, the first logical page of the file is updated. This results in the replacement of the prior tag (t_0) with a new tag (t_0'), effectively deleting the prior version of the data page (D_0). A new file object is re-written (M') and a new tag for that object is stored in its parent object, effectively deleting the previous object (M).
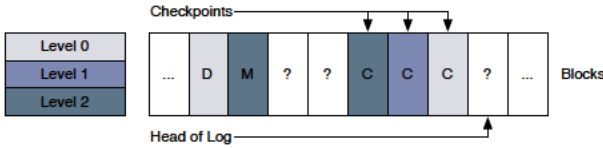

Fig. 4: A view of multi-level blocks in DEFY.

During this process, DEFY ensures the entire disk looks pseudo-random, by encrypting any empty (all-zero) blocks with an ephemeral key. Empty blocks only exist on a fresh disk, or in the rare occasion when power is lost after erasing but before writing a block. Thus, this procedure occurs rarely, and only incurs overhead when first mounting a DEFY partition.

Due to DEFY's deletion policies there are no out-of-date pages, so any decrypted page is a valid and live page in the system. Further, authenticated encryption ensures that, with very high probability, only authentic pages will decrypt correctly.

When initially mounting a DEFY partition, a user provides between zero and ℓ level names and passwords. Other levels may be revealed dynamically after the file system has been mounted.

Creating Levels: When a user creates a deniability level, she provides the system with a unique level name and password. These are used to generate that level's keys, using a password-based key derivation scheme. DEFY checks that the level key does not open an existing level by looking for a valid level

checkpoint for that key pair (described above). If the level is free to be created, the system creates a level directory under the root file system, as well as all the necessary RAM data structures, and writes a checkpoint at that level.

The user must choose where the new level is placed relative to the total order of currently revealed levels. To enforce the total ordering, the user is not allowed to place the new level above the highest level currently revealed (as there may be some higher unrevealed level). If there are no currently open levels, a new level is created under the assumption that there are no other levels in the system.

Indeed, a total order on levels is not strictly necessary in DEFY. The consequence of having two levels that are incomparable is no greater than the consequences of having unrevealed levels: potential data loss. One new consequence of failing to maintain a total order on levels, however, is that there will be no highest level, offering a global view of the valid data stored. Practically, having such a view allows the possibility of performing certain maintenance operations manually, such as garbage collection.

Links: DEFY supports both hard and soft links. They are represented by file objects in DEFY, which are created and encrypted just like a normal file object. Links across levels are disallowed by policy, preventing users from inadvertently disclosing unrevealed levels.

Deleting & Truncating Data: When an object is unlinked, the DEFY object is removed from memory, and a tag cascade is triggered, just as with a write operation. Similarly, a truncation results in the corresponding tags to be overwritten, and a new metadata object written to disk.

We remark that moving data across deniability levels requires no special consideration: it can be decomposed into a write to the destination and the deletion of the source. Because of DEFY's use of a semantically secure block cipher, an adversary is unable to identify identical files across deniability levels. A file moved to a higher level will simply appear to be deleted when DEFY is mounted at a lower level.

Closing Levels & Un-mounting: When the user wishes to close a level to hide its contents, DEFY also closes all revealed, higher levels. Starting with the highest revealed level, DEFY writes a checkpoint to the next available block. It then zeros the RAM data structures for that level (including cryptographic keys), removing any evidence that it was ever open. This process repeats for the next lowest level, until all levels at and above ℓ have been closed.

Un-mounting the file system is a special case of closing all levels. Thus, the lowest level's checkpoint is written last. As explained above, writing the final checkpoint at the lowest level serves two purposes: (1) it provides plausible deniability that the disk was merely mounted and unmounted at the lowest level, between snapshots; and, (2) it ensures that all higher-level data is written behind this marker, minimizing data loss should the disk be mounted at an intermediate level.

Garbage Collection: Unlike YAFFS, DEFY performs no active garbage collection. There is no strategy to reclaim pages

by combining and re-writing pages to new locations, without increasing the possibility of overwriting hidden data. Garbage collection can be employed safely when the highest deniability level is revealed; however, there is no systematic way for DEFY to recognize this event. Old data within a deniability level will eventually be reallocated when the log wraps around the device, since old data are unable to be decrypted, and thus marked as free. If immediate capacity is an issue, users may opt to mount at the highest level and manually start garbage collection.

Interfacing with DEFY: All the above DEFY operations require system interfaces, either extending their POSIX counterparts or requiring entirely new APIs. One approach to implementing these interfaces is to develop entirely new system calls, requiring one to modify the VFS and rebuild the kernel. This is particularly problematic in the mobile domain, where users may not have the option of installing custom kernels. Instead, DEFY interfaces are invoked by passing DEFY-specific flags via the IOCTL system call. A set of user-level tools, which invoke these IOCTLs, are provided to simplify DEFY's interface.

VIII. SECURITY ANALYSIS

Our goal is to show DEFY achieves plausible deniability in the snapshot adversary model—the strongest model considered in this setting. We begin our argument in the more restrictive single-view adversary model, and expand this to include the snapshot adversary model. It is interesting to note that these two adversarial models are "close" when considering log-structured file systems, *i.e.* unlike other deniable file systems, for DEFY, the weaker security notion is not significantly easier to achieve than the stronger one.

Typical arguments for security against single-view adversaries [9], [28], [33] require demonstrating that three classes of blocks—unallocated blocks, formerly allocated (deleted) blocks, and unrevealed blocks—are indistinguishable to an adversary. In prior systems, this property stems from the indistinguishability of ciphertexts, but those cryptographic arguments are, alone, not sufficient proof. Instead, an additional argument is required, leveraging some procedural or system property, *e.g.* random block placement. This is due to the fact that the view of the system state can leak information that may aid the task of distinguishing blocks, and the adversary is not restricted to viewing blocks in isolation.

In DEFY, these three categories of blocks, in isolation, are indistinguishable to a computationally-bound adversary. Unallocated blocks are initialized as random ciphertexts, indistinguishable from both unrevealed blocks and formerly allocated blocks. This is due to the semantic security of the underlying cryptographic transforms employed, and the fact that previously allocated blocks are securely deleted. These properties follow from the original proofs accompanying those cryptographic constructions (see, *e.g.*, [10], [11], [38]). We have not introduced or even modified any of these cryptographic tools but, rather, our contributions are in using them in a new and important context: to demonstrate how ciphertext blocks are parsed and stored in DEFY, to accommodate ciphertext expansion by co-opting the out-of-bound portion of flash memory, to leverage dedicated tag storage and to employ

(a) Reading a block.
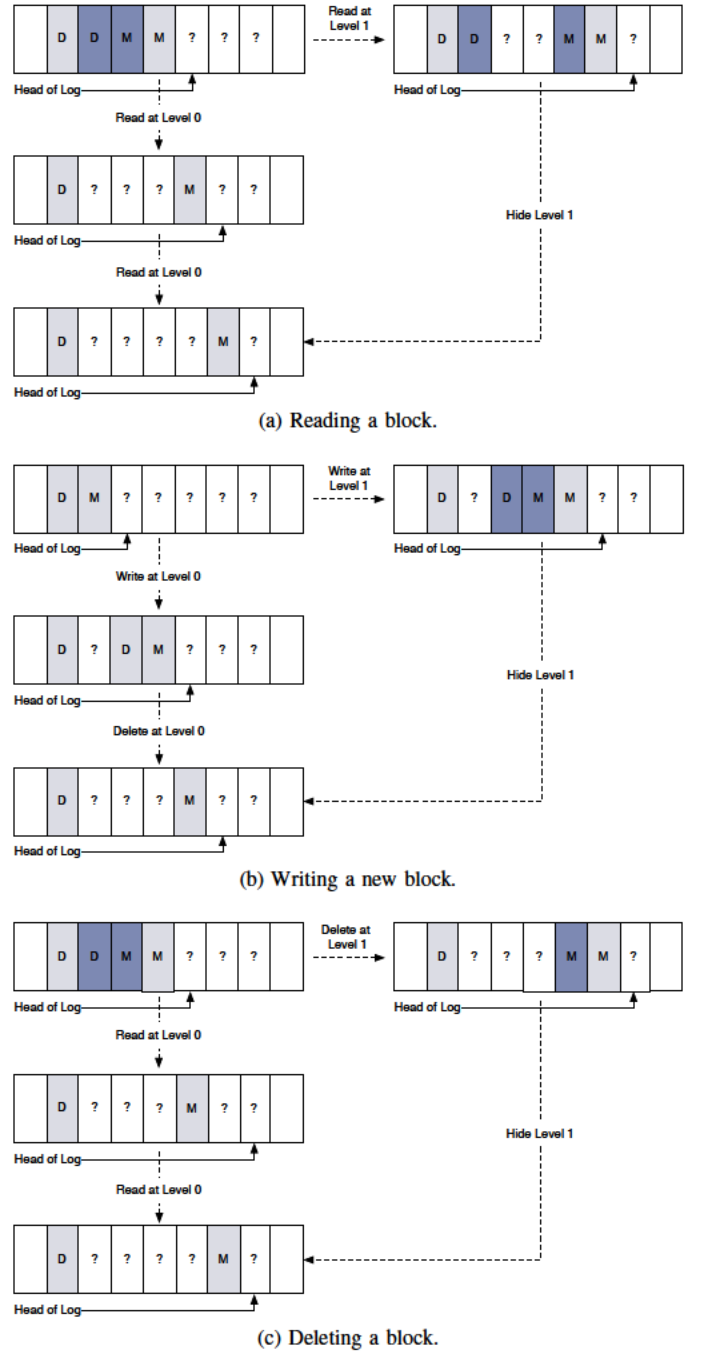

(b) Writing a new block.

(c) Deleting a block.

Fig. 5: Simplified cases to consider for plausible deniability. Here, light grey blocks are encrypted using the level 0 key, and purple blocks are encrypted using the level 1 key. Blocks that contain question marks are indistinguishable to an adversary.

tag rotation to achieve practical and efficient secure deletion from a systems perspective (even when cryptographic keys are exposed). We must, however, show these categories of blocks remain indistinguishable, even *in context*, rather than simply in isolation.

As stated previously, we cannot employ random block placement in DEFY, and wear-leveling undermines deniability. Instead, we must adapt a procedural strategy similar to that

employed by Pang *et al.*, by creating indecipherable blocks during normal file system use. Instead of writing explicit "dummy" data, we securely delete old pages when new pages are written. Each time data or metadata is modified, a new page is written to the head of the log to reflect that change; if the change invalidates a previous version of the page, the old page is immediately deleted as part of the tag cascade. This results in indecipherable "gaps" in the log during ordinary operation, such as writing and modifying data (see, *e.g.*, Figure 3). Our task is to show that for each consequential operation in DEFY, our procedural protections generate a filesystem state that corresponds to some plausible alternative world lacking unrevealed blocks (*i.e.* those written at a higher deniability level) entirely.

Thus, to consider a snapshotting adversary, it suffices to consider two arbitrarily close snapshots of a simplified DEFY file system state, showing for each operation at a higher deniability level, the view of the system when revealed at a lower level is plausibly deniable. That is, that there exists at least one set of operations performed only at a lower level that results in the same file system state. The initial state we consider holds a small number of contiguous blocks, with either one or two active deniability levels; our arguments are equally applicable to more complex initial states. The operations we consider are: reading, writing, and deleting (truncating) a block at a deniability level greater than level zero. Each case is considered separately, next.

When reading a unrevealed block, the metadata block for that level is updated (*i.e.*, to reflect the changes to the "last access time" metadata values) and, procedurally, the metadata blocks for all lower levels are written first (see Figure 5a). To an adversary unable to interpret unrevealed blocks, the view of the system is identical to one in which no higher level blocks exist, but a sequence of reads occurred causing new metadata blocks to be written. When deleting a higher level block (see Figure 5c), the view and alternative history is the same.

When writing a higher level block (see Figure 5b), the deniable view corresponds to several possible actions: a sequence of reads, or a file-append followed by truncation, or a write followed by erasure. Indeed, these plausible equivalencies are demonstrative and not exhaustive. Operations on larger amounts of data serve only to increase the set of possible alternative histories. As any single operation with an arbitrarily close snapshot is deniable, we conclude that a sequence of these operations admits as many plausible alternative histories.

IX. PERFORMANCE EVALUATION

While high performance is not the primary focus of DEFY, it is certainly a valuable criterion for judging a system's potential adoption and use. We are most interested in comparing the performance of DEFY relative to other secure file systems for mobile devices. We compare the performance of our prototype with the latest versions of WhisperYAFFS, YAFFS, ext3 and ext4. We choose YAFFS and the ext family of file systems as they represent the state of the art in file systems used on solid-state drives in many Linux distributions and mobile devices, particularly those based on Android. WhisperYAFFS acts as our baseline for encrypted file system performance on a mobile device. To achieve this, we put in significant engineering effort
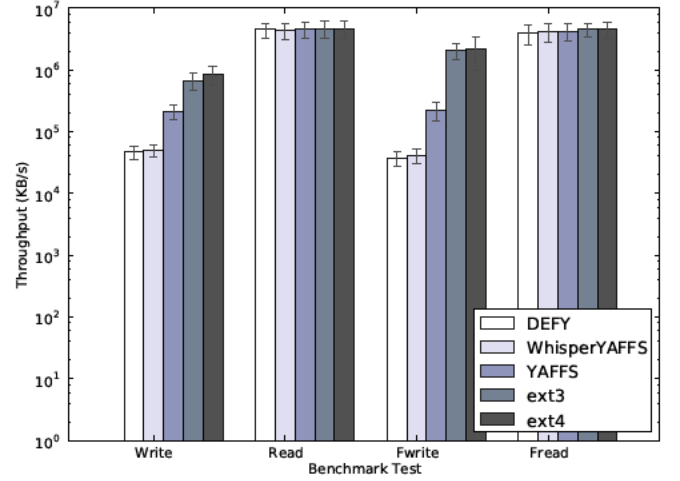

Fig. 6: Average throughput and standard deviation in KB/s over four runs for each file system using the IOZone benchmark tool (presented in log scale).
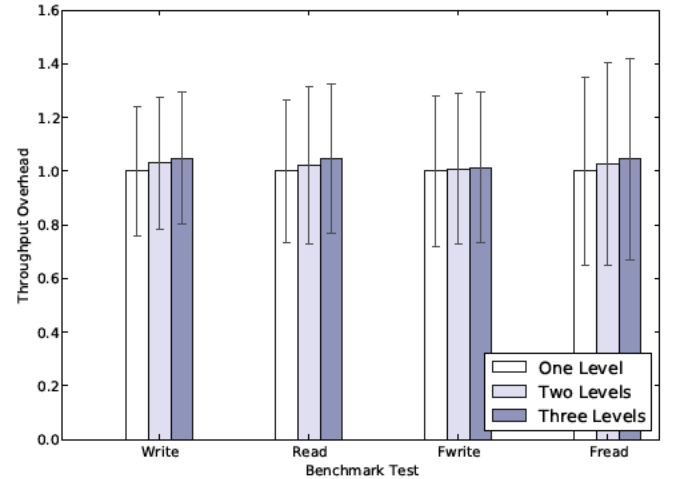

Fig. 7: Average throughput over four runs for DEFY with three revealed deniability levels using the IOZone benchmark tool. Results are normalized by the average cost for one level, to reveal per-level overheads.

to make WhisperYAFFS compatible with version 3.x of the Linux kernel; we are in the process of forking and releasing our modifications. Unfortunately, we are unable to compare DEFY's performance with prior implementations of deniable file system systems, as they are all no longer maintained, or were never released, or are incompatible with the modern Linux kernel. Mobiflage's architect notes the same [45].

Evaluation was performed on an Ubuntu 13.04 machine with 4GB of memory and a single processor. The system was augmented with the `nandsim` MTD device simulator [1] configured to emulate a 64MB flash device with 2KB pages.

Measurements were taken using IOZone [32], an industry-standard file system benchmarking tool designed to measure a wide variety of file system performance characteristics. In this work, we focus on measuring four core file system operations:

read and write operations, both unbuffered and buffered. We believe these characterize the most common and I/O-bound operations for a file system.

With each test, IOZone performs an I/O operation on a number of uniformly-sized files, up to some maximum size. For example, when measuring the write performance of a 64KB sized file, IOZone attempts to write sixteen 4KB files, eight 8KB files, and so on, up to one 64KB file. For each benchmark, we average the throughputs across four runs. Results, with standard deviations, are presented in Figure 6.

We find that DEFY performs comparably with WhisperYAFFS, while both DEFY and WhisperYAFFS underperform when compared with YAFFS, ext3, and ext4. This is not unexpected, due in large part to the additional computation requirements necessary to support their cryptographic operations. It is notable that these results suggest our AON transform comes at an expense similar to AES in XTS mode [20], used by WhisperYAFFS. Further, we believe the additional computational and I/O requirements for tag cascading have little to no impact on normal file system operations.

We also find that the number of deniability levels has little effect on file system performance. We configured a DEFY partition with three deniability levels and performed the same IOZone benchmarks mounted under all three levels (*i.e.* one level, two level, and all levels revealed). Figure 7 presents the results. These findings demonstrate that performance in DEFY is an artifact of IO irrespective of the number of revealed (or unrevealed) levels. DEFY's log-structured nature writes new data to the head of the log, regardless from which level is performing the write. Writing data from many levels concurrently may cause a single level's blocks to be fragmented across the device, which could lead to poor sequential read performance for rotating media. Indeed, poor sequential read performance is a contributing factor to why log-structured files systems has not been more widely adopted. However, the uniform random access performance of solid state drives render data non-contiguity largely irrelevant, allowing system engineer's to once again enjoy the manifold benefits of a log structure (*e.g.* implicit versioning, inherent consistency, and simplified data structures).

X. Deniable Filesystems in Practice

We remark that the use of deniable file systems from social, legal, and usability perspectives has not been well-explored in the literature. In particular, all deniable file systems to date employ user passwords for securing files at a certain deniability level. This allows an adversary to undermine plausible deniability at the cost of a password-guessing attack. In our adversarial setting, some password protection mechanisms can be successfully utilized (*e.g.*, using password-based key derivation functions to increase the cost of brute force attacks) while others cannot. For example, although not needed, password salts for each level's password, would need to be stored somewhere persistent and, thus, could undermine plausible deniability. Similarly, key management techniques used in traditional disk encryption software, such as key wrapping [7], [31], [40], could likewise undermine deniability. Currently, all deniable file systems demand the user select good passwords, and DEFY is no exception to this.

Relatedly, attacks against deniable file systems may be possible when adversaries have access to data from external sources ("hints") or are otherwise unconstrained by the single-view and snapshot models. For example, Skillen and Mannan propose a "colluding carrier" attack [46], where the adversary colludes with a wireless provider or ISP (*e.g.*, through governmental writ), collecting network trace data to aid later forensic analysis of the device. Discrepancies between the device logs and the carrier's logs may enable an adversary to conclude the presence of hidden data, and compel hidden levels to be revealed. Skillen and Mannan make suggestions to restrict these threats to deniable file systems for mobile devices, including disabling wireless connectivity (or limiting network connectivity to WiFi only) during privileged use, and using multiple SIM cards and carriers to make log collection difficult. We, too, acknowledge DEFY's limitations to resist this type of attack, and suggest that users follow those same practices in countries where carriers may assist forensic investigation.

More generally, we find the ultimate guarantees of deniable file systems have not been critically examined. In all existing definitions, the onus is on the adversary to prove that the system contains hidden data. This reflects the *presumption of innocence* common in many legal systems, *i.e.* the adversary must prove the user's guilt and until then she is presumed innocent. Given that adversaries have been known to torture individuals for their passwords [47], it is unclear if a system founded upon this tenant is viable for use in truly hostile environments. In practice, the very *existence* of a deniable file system may draw unwanted attention. To a casual observer, DEFY looks and behaves like a full-disk encryption scheme, so that revealing a single level may be convincing.

XI. Conclusion

We have presented DEFY, a deniable file system for solid-state memory, usable as a file system for mobile devices and laptops. Current design patterns for deniable file systems cannot be easily adapted for many mobile devices; this is largely due to system design assumptions about the storage media that, while valid for many settings, are inappropriate for solid-state drives. The physical properties of solid-state memory require wear leveling and disallow in-place updates, motivating our use of a log-structured file system. Thus, DEFY is the first log-structured deniable file system. At first glance, log-structured systems appear to deeply conflict with the goal of empowering a user to deny actions from the recent past. We apply techniques from a secure, versioning file system in a completely new way, to support a log-structure with a deniable history. As the first deniable file system designed for log-structured storage, we believe DEFY fills a gap in the space of privacy enhancing technologies for devices using solid-state drives, such as mobile devices. DEFY also supports other features useful in a mobile setting, including authenticated encryption and fine-grained secure deletion of data. Our DEFY prototype implementation is based on YAFFS and WhisperYAFFS, and is released as an open-source project on BitBucket[1]. Preliminary evaluation demonstrates performance similar to that experienced with full-disk encryption on these devices, *i.e.*, WhisperYAFFS.

---

[1] https://bitbucket.org/solstice/defy/

REFERENCES

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

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

[3] "GuardianProject/ChatSecure:Android," 2014, https://guardianproject.info/apps/chatsecure.

[4] "GuardianProject/Orbot," 2014, https://guardianproject.info/apps/orbot/.

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

[6] "Truecrypt," 2014, http://www.truecrypt.org/.

[7] Accredited Standards Committee, X9, Inc., "ANS X9.102- Wrapping of Keys and Associated Data," http://eprint.iacr.org/2004/340.pdf, November 2004.

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

[9] R. Anderson, R. Needham, and A. Shamir, "The steganographic file system," in *Information Hiding, Second International Workshop (IH '98)*, Portland, Oregon, USA, April 1998, pp. 73–82.

[10] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A Concrete Security Treatment of Symmetric Encryption," in *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1997, pp. 394–403.

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

[12] M. Blaze, "High-bandwidth encryption with low-bandwidth smartcards," in *Fast Software Encryption*, vol. 1039, 1996, pp. 33–40, lecture Notes in Computer Science.

[13] M. Blaze, J. Feigenbaum, and M. Naor, "A formal treatment of remotely keyed encryption," in *Advances in Cryptology – EUROCRYPT '98*, vol. 1403, 1998, pp. 251–265, Lecture Notes in Computer Science.

[14] M. Blaze, "A cryptographic file system for UNIX," in *Proceedings of the ACM Conference on Computer and Communications Security*, 1993, pp. 9–16.

[15] D. Boneh and R. Lipton, "A revocable backup system," in *Proceedings of the USENIX Security Symposium*, July 1996, pp. 91–96.

[16] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky, "Deniable encryption," *Advances in Cryptology — CRYPTO '97*, pp. 90–104, 1997.

[17] A. Czekis, D. J. St Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier, "Defeating encrypted and deniable file systems: Truecrypt v5.1a and the case of the tattling os and applications," *3rd USENIX Workshop on Hot Topics in Security (HotSec '08)*, 2008.

[18] D. Defreez, "Android privacy through encryption," Master's thesis, Southern Oregon University, May 2012, available at http://goo.gl/94HBb.

[19] Y. Dodis and J. An, "Concealment and its applications to authenticated encryption," in *Advances in Cryptology – EUROCRYPT '03*, vol. 2656, 2003, Lecture Notes in Computer Science.

[20] M. Dworkin, "Recommendation for block cipher modes of operation: The XTS-AES mode for confidentiality on storage devices," National Institute of Standards and Technology, NIST Special Publication SP-800-38E, January 2010.

[21] S. L. Garfinkel and A. Shelat, "Remembrance of data passed: A study of disk sanitization practices," *IEEE Security and Privacy*, vol. 1, no. 1, pp. 17–27, 2003.

[22] P. Gasti, G. Ateniese, and M. Blanton, "Deniable cloud storage: sharing files via public-key deniability," in *WPES '10: Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, Oct. 2010.

[23] P. Gutmann, "Secure deletion of data from magnetic and solid-state memory," in *Proceedings of the USENIX Security Symposium*, July 1996, pp. 77–90.

[24] M. Jakobsson, J. Stern, and M. Yung, "Scramble all. Encrypt small." in *Fast Software Encryption*, vol. 1636, 1999, lecture Notes in Computer Science.

[25] B. Kaliski, "PKCS #5: Password-based cryptography specification," IETF Network Working Group, Request for Comments RFC 2898, Sept. 2000.

[26] 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*, vol. 26, pp. 27–38, 2010.

[27] C. Manning, "How YAFFS works," 23 May 2012, available at http://goo.gl/0Mdja.

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

[29] J. Mull, "How a Syrian refugee risked his life to bear witness to atrocities," March 2012, toronto Star Online; posted 14-March-2012. [Online]. Available: http://www.thestar.com/news/world/2012/03/14/how_a_syrian_refugee_risked_his_life_to_bear_witness_to_atrocities.html

[30] T. Müller and M. Spreitzenbarth, "Frost: Forensic recovery of scrambled telephones," in *Applied Cryptography and Network Security (ACNS'13)*, 2013, pp. 373–388.

[31] National Institute of Standards and Technology, "AES key wrap specification," November 2001.

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

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

[34] Z. N. J. Peterson, R. 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*, 2005.

[35] N. Provos and D. Mazieres, "A future-adaptable password scheme," in *Proceedings of the USENIX Annual Technical Conference*, 1999.

[36] 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*, 2012, pp. 333–348. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/reardon

[37] Reporters Without Borders, "Internet enemies," 12 March 2012, available at http://goo.gl/x6zZ1.

[38] R. L. Rivest, "All-or-nothing encryption and the package transform," in *Fast Software Encryption Conference*, vol. 1267, 1997, pp. 210–218, lecture Notes in Computer Science.

[39] 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*, November 2001, pp. 196–205.

[40] P. Rogaway and T. Shrimpton, "Deterministic authenticated-encryption a provable-security treatment of the key-wrap problem," in *Advances in Cryptology – EUROCRYPT 06*, vol. 4004, 2007, Lecture Notes in Computer Science.

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

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

[43] B. Schneier, ""Evil maid" attacks on encrypted hard drives," 23 Oct. 2009, http://goo.gl/Z1Kny. [Online]. Available: https://www.schneier.com/blog/archives/2009/10/evilmaidattac.html

[44] 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*, January 1993, pp. 307–326.

[45] A. Skillen, "Deniable storage encryption for mobile devices," Master's thesis, Concordia University, 2013.

[46] 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. [Online]. Available: http://spectrum.library.concordia.ca/975074/

[47] M. Weaver, "Developer tortured by raiders with crowbars," *Daily Telegraph*, 31 October 1997.

[48] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2011.

[49] "GitHub: WhisperSystems/RedPhone," http://goo.gl/Mmz9s, Whisper-Systems, 2012.

[50] "GitHub: WhisperSystems/TextSecure," http://goo.gl/3qoV8, Whisper-Systems, 2012.

[51] "GitHub: WhisperSystems/WhisperYAFFS: Wiki," http://goo.gl/Qsku4, WhisperSystems, 2012. [Online]. Available: https://github.com/WhisperSystems/WhisperYAFFS/wiki

[52] C. P. Wright, M. C. Martino, and E. Zadok, "Ncryptfs: A secure and convenient cryptographic file system," in *Proceedings of the USENIX Technical Conference*, 2003, pp. 197–210.