Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory

Joel Reardon, Srdjan Capkun, David Basin
Department of Computer Science, ETH Zurich

Abstract

We propose the Data Node Encrypted File System (DNEFS), which uses on-the-fly encryption and decryption of file system data nodes to efficiently and securely delete data on flash memory systems. DNEFS is a generic modification of existing flash file systems or controllers that enables secure data deletion while preserving the underlying systems' desirable properties: application-independence, fine-grained data access, wear-levelling, and efficiency.

We describe DNEFS both abstractly and in the context of the flash file system UBIFS. We propose UBIFSec, which integrates DNEFS into UBIFS. We implement UBIFSec by extending UBIFS's Linux implementation and we integrate UBIFSec in the Android operating system running on a Google Nexus One smartphone. We show that it is efficient and usable; Android OS and applications (including video and audio playback) run normally on top of UBIFSec. To the best of our knowledge, this work presents the first comprehensive and fully-implemented secure deletion solution that works within the specification of flash memory.

1 Introduction

Flash memory is used near universally in portable devices. However, the way modern systems use flash memory has a serious drawback—it does not guarantee deletion of stored data. To the user, data appears to be deleted from the file system, but in reality it remains accessible after deletion [39]. This problem is particularly relevant for modern smartphones, as they store private data, such as communications, browsing, and location history as well as sensitive business data. The storage of such data on portable devices necessitates guaranteed secure deletion.

Secure deletion is the operation of sanitizing data on a storage medium, so that access to the data is no longer possible on that storage medium [9]. This is in contrast to standard deletion, where metadata simply indicates that the data's storage location is no longer needed and can be reused. The time between marking data as deleted and its actual (secure) deletion is called the *deletion latency*. We use the term *guaranteed secure deletion* to denote secure deletion with a fixed, (small) finite upper bound on the deletion latency for all data.

On magnetic storage media, secure data deletion is implemented by overwriting a file's content with non-sensitive information [29], or by modifying the file system to automatically overwrite any discarded sector [2]. However, flash memory cannot perform in-place updates of data (i.e., overwrites) [8]; it instead performs erasures on *erase blocks*, which have a larger granularity than read/write operations. A single erase block may store data for different files, so it can only be erased when all the data in the erase block is marked as deleted or when the live data is replicated elsewhere. Moreover, flash memory degrades with each erasure, so frequent erasures shorten the device's lifetime. Therefore, the simplistic solution of erasing any erase block that contains deleted data is too costly with regards to time and device wear [35].

In this work, we present the Data Node Encrypted File System (DNEFS), which securely and efficiently deletes data on flash memory; it requires only a few additional erasures that are evenly distributed over the erase blocks. DNEFS uses on-the-fly encryption and decryption of individual data nodes (the smallest unit of read/write for the file system) and relies on key generation and management to prevent access to deleted data. We design and implement an instance of our solution for the file system UBIFS [14] and call our modification UBIFSec.

UBIFSec has the following attractive properties. It provides a guaranteed upper bound on deletion latency. It provides fine-grained deletion, also for truncated or overwritten parts of files. It runs efficiently and produces little wear on the flash memory. Finally, it is easy to integrate into UBIFS's existing Linux implementation, and requires no changes to the applications using UBIFS. We deploy UBIFSec on a Google Nexus One smartphone [11] running an Android OS. The system and applications (including video and audio playback) run normally on top of UBIFSec.

Even though DNEFS can be implemented on YAFFS (the file system used on the Android OS), this would have required significant changes to YAFFS. We test DNEFS within UBIFS, which is a supported part of the standard Linux kernel (since version 2.6.27) and which provides interfaces that enable easy integration of DNEFS.

We summarize our contributions as follows. We design DNEFS, a system that enables guaranteed secure data deletion for flash memory—operating within flash memory's specification [26]. We instantiate DNEFS as UBIFSec, analyze its security, and measure its additional battery consumption, throughput, computation time, and flash memory wear to show that it is practical for real-world use. We provide our modification freely to the community [37].

2 Background

Flash Memory. Flash memory is a non-volatile storage medium consisting of an array of electronic components that store information [1]. Flash memory has very small mass and volume, does not incur seek penalties for random access, and requires little energy to operate. As such, portable devices almost exclusively use flash memory.

Flash memory is divided into two levels of granularity. The first level is called *erase blocks*, which are on the order of 128 KiB [11] in size. Each erase block is divided into *pages*, which are on the order of 2 KiB in size. Erase blocks are the unit of erasure, and pages are the unit of read and write operations [8]. One cannot write data to a flash memory page unless that page has been previously *erased*; only the erasure operation performed on an erase block prepares the pages it contains for writing.

Erasing flash memory causes significant physical wear [22]. Each erasure risks turning an erase block into a bad block, which cannot store data. Flash erase blocks tolerate between 10^4 to 10^5 erasures before they become bad blocks. To promote a longer device lifetime, erasures should be evenly levelled over the erase blocks.

MTD Layer. On Linux, flash memory is accessed through the Memory Technology Device (MTD) layer [23]. MTD has the following interface: read and write a page, erase an erase block, check if an erase block is bad, and mark an erase block as bad. Erase blocks are referenced sequentially, and pages are referenced by the erase block number and offset.

Flash File Systems. Several flash memory file systems have been developed at the MTD layer [4, 40]. These file systems are log-structured: a class of file systems that (notably) do not perform in-place updates. A log-structured file system consists of an ordered list of changes from an initial empty state, where each change to the file system is appended to the log's end [34]. Therefore, standard log-structured file systems do not provide secure deletion because new data is only appended.

When a change invalidates an earlier change then the new, valid data is appended and the erase block containing the invalidated data now contains wasted space. Deleting a file, for example, appends a change that indicates the file is deleted. All the deleted file's data nodes remain on the storage medium but they are now invalid and wasting space. A garbage collection mechanism detects and recycles erase blocks with only invalid data; it also copies the remaining valid data to a new location so it may recycle erase blocks mostly filled with invalid data.

Flash Translation Layer. Flash memory is commonly accessed through a Flash Translation Layer (FTL) [1, 15], which is used in USB sticks, SD cards, and solid state drives. FTLs access the raw flash memory directly, but expose a typical hard drive interface that allows any regular file system to be used on the memory. FTLs can either be a hardware controller or implemented in software. An FTL translates logical block addresses to raw physical flash addresses, and internally implements a log-structured file system on the memory [6]. Therefore, like log-structured file systems, FTLs do not provide secure data deletion. In Section 5 we explain how to modify an FTL to use DNEFS to enable efficient secure deletion for any file system mounted on it.

UBI Layer. Unsorted Block Images (UBI) is an abstraction of MTD, where logical erase blocks are transparently mapped to physical erase blocks [10]. UBI's logical mapping implements wear-levelling and bad block detection, allowing UBI file systems to ignore these details. UBI also permits the atomic updating of a logical erase block—the new data is either entirely available or the old data remains.

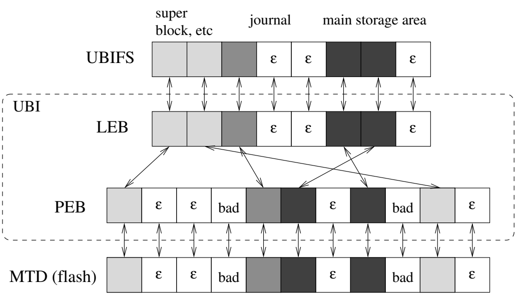


Figure 1: Erase block relationships among MTD, UBI, and UBIFS. Different block shades label different areas of the file system. Empty LEBs are labelled by ε and are not mapped to a corresponding PEB by UBI. Similarly, bad PEBs are labelled and not mapped onto by UBI.

UBI exposes the following interface: read and write to a Logical Erase Block (LEB), erase an LEB, and atomically update the contents of an LEB. UBI LEBs neither become bad due to wear, nor should their erasure counts be levelled.

Underlying this interface is an injective partial mapping from LEBs to physical erase blocks (PEBs), where PEBs correspond to erase blocks at the MTD layer. The lower half of Figure 1 illustrates this relationship. Wear monitoring is handled by tracking the erasures at the PEB level, and a transparent remapping of LEBs occurs when necessary. Remapping also occurs when bad blocks are detected. Despite remapping, an LEB's number remains constant, regardless of its corresponding PEB.

Atomic updates occur by invoking UBI's update function, passing as parameters the LEB number to update along with a buffer containing the desired contents. An unused and empty PEB is selected and the page-aligned data is then written to it. UBI then updates the LEB's mapping to the new PEB, and the previous PEB is queued for erasure. This erasure can be done either automatically in the background or immediately with a blocking system call. If the atomic update fails at any time—e.g., because of a power loss—then the mapping is unchanged and the old PEB is not erased.

UBIFS. The UBI file system, UBIFS [14], is designed specifically for UBI, and Figure 1 illustrates UBIFS's relationship to UBI and MTD. UBIFS divides file data into fixed-sized data nodes. Each data node has a header that stores the data's inode number and its file offset. This inverse index is used by the garbage collector to determine if the nodes on an erase block are valid or can be discarded.

UBIFS first writes all data in a journal. When this journal is full, it is committed to the main storage area by logically moving the journal to an empty location and growing the main storage area to encompass the old journal. An index is used to locate data nodes, and this index is also written to the storage medium. At its core, UBIFS is a log-structured file system; in-place updates are not performed. As such, UBIFS does not provide guaranteed secure data deletion.

Adversarial Model. In this work, we model a novel kind of attacker that we name the *peek-a-boo attacker*. This attacker is more powerful than the strong *coercive* attacker considered in other secure deletion works [27, 30]. A coercive attacker can, at any time, compromise both the storage medium containing the data along with any secret keys or passphrases required to access it. The peek-a-boo attacker extends the coercive attacker to also allow the attacker to obtain ("to peek into") the content of the storage medium at some point(s) in time prior to compromising the storage medium.

Coercive attacks model legal subpoenas that require users to forfeit devices and reveal passwords. Since the time of the attack is arbitrary and therefore unpredictable, no extraordinary sanitization procedure can be performed prior to the compromise time. Since the attacker is given the user's secret keys, it is insufficient to simply encrypt the storage media [17]. The peek-a-boo attacker models an attacker who additionally gets temporary read-access to the medium (e.g., a hidden virus that is forced to send suicide instructions upon being publicly exposed) and then subsequently performs a coercive attack. It is roughly analogous to forward secrecy in the sense that if a secure deletion scheme is resilient to a peek-a-boo attacker, it prevents recovery of deleted data even if an earlier snapshot of the data from the storage medium is available to the attacker.

Figure 2 shows a timeline of data storage and an adversarial attack. We divide time into discrete intervals called *purging epochs*. At the end of each purging epoch any data marked for deletion is securely deleted (purged). We assume that purging is an atomic operation. The lifetime of a piece of data is then defined as all the purging epochs from the one when it was written to the one when it was deleted. We say that data is *securely deleted* if a peek-a-boo attacker cannot recover the data when performing peek and boo attacks in any purging epochs outside the data's lifetime.

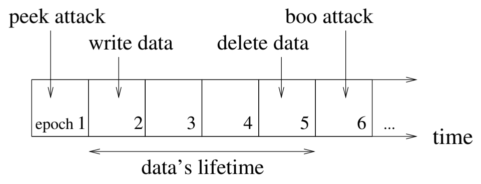


Figure 2: Example timeline for secure deletion. Time is divided into discrete purging epochs. Data is written in epoch 2 and deleted in epoch 5, and the data's lifetime includes all epochs between these. Here, the peek attack (read access to the entire storage medium) occurs in epoch 1 and the boo attack (full compromise of the storage medium and secret keys/passphrases) in epoch 6. More generally, they can occur in any purging epochs outside the data's lifetime.

3 DNEFS

In this section we describe our main contribution: a solution for efficient secure deletion for flash memory. We first list our requirements for secure deletion, and afterwards describe our solution.

3.1 Secure Deletion Requirements

We present four requirements for secure deletion solutions. The solution must be sound, fine-grained, efficient, and simple.

Soundness requires that the solution ensures guaranteed secure data deletion against a strong attacker; we use the peek-a-boo attacker defined in Section 2.

Fine-grained requires the solution to securely delete data, however small. This includes overwritten or truncated files, such as data removed from a long-lived database.

The solution must be *efficient* in terms of resource consumption. For flash memory and portable devices, the relevant resources are battery consumption, computation time, storage space, and device lifetime, i.e., minimizing and levelling wear.

Finally, *simplicity* requires that the solution can be easily implemented as part of existing systems. For our purposes, this means that adding secure deletion to existing file systems must be straightforward. We want to minimize the necessary changes to the existing code and isolate the majority of the implementation in new functions and separate data structures. We want the change to be easily audited and analyzed by security-minded professionals. Moreover, we must not remove or limit any existing feature of the underlying file system.

3.2 Our Solution

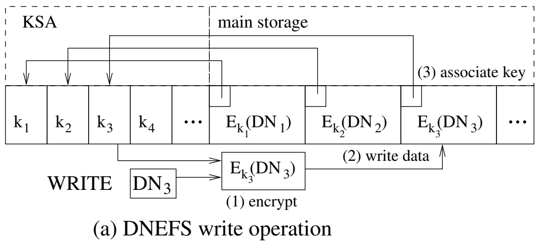
We now present our secure deletion solution and show how it fulfills the listed requirements.

In the spirit of Boneh and Lipton [3], DNEFS uses encryption to provide secure deletion. It encrypts each individual data node (i.e., the unit of read/write for the file system) with a different key, and then manages the storage, use, and purging of these keys in an efficient and transparent way for both users and applications. Data nodes are encrypted before being written to the storage medium and decrypted after being read; this is all done in-memory. The keys are stored in a reserved area of the file system called the key storage area.

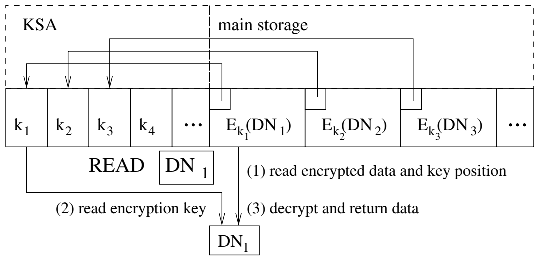
DNEFS works independent of the notion of files; neither file count/size nor access patterns have any influence on the size of the key storage area. The encrypted file data stored on the medium is no different than any reversible encoding applied by the storage medium (e.g., error-correcting codes) because all legitimate access to the data only observes the unencrypted form. This is not an encrypted file system, although in Section 5 we explain that it can be easily extended to one. In our case, encryption is simply a coding technique that we apply immediately before storage to reduce the number of bits required to delete a data node from the data node size to the key size.

Key Storage Area. Our solution uses a small migrating set of erase blocks to store all the data nodes' keys—this set is called the Key Storage Area (KSA). The KSA is managed separately from the rest of the file system. In particular, it does not behave like a log-structured file system: when a KSA erase block is updated, its contents are written to a new erase block, the logical reference to the KSA block is updated, and the previous version of the KSA erase block is then erased. Thus, except while updating, only one copy of the data in the KSA is available on the storage medium. Our solution therefore requires that the file system or flash controller that it modifies can logically reference the KSA's erase blocks and erase old KSA erase blocks promptly after writing a new version.

Each data node's header stores the logical KSA position that contains its decryption key. The erase blocks in the KSA are periodically erased to securely delete any keys that decrypt deleted data. When the file system no longer needs a data node—i.e, it is removed or updated—we mark the data node's corresponding key in the KSA as deleted. This solution is independent of the notion of files; keys are marked as deleted whenever a data node is discarded. A key remains marked as deleted until it is removed from the storage medium and its location is replaced with fresh, unused random data, which is then marked as unused.



(a) DNEFS write operation



(b) DNEFS read operation

Figure 3: (a) writing a new data node DN_3: DN_3 is first encrypted with an unused key k_3 and then written to an empty position in the main storage. A reference to the key's position in the KSA is stored alongside $E_{k_3}(\mathrm{DN}_3)$. (b) reading a data node DN_1: $E_{k_1}(\mathrm{DN}_1)$ is first read from the storage medium along with a reference to its key k_1 in the KSA. The key is then read and used to decrypt and return DN_1.

When a new data node is written to the storage medium, an unused key is selected from the KSA and its position is stored in the data node's header. DNEFS does this seamlessly, so applications are unaware that their data is being encrypted. Figure 3 illustrates DNEFS's write and read algorithms.

Purging. Purging is a periodic procedure that securely deletes keys from the KSA. Purging proceeds iteratively over each of the KSA's erase blocks: a new version of the erase block is prepared where the used keys remain in the same position and all other keys (i.e., unused and deleted keys) are replaced with fresh, unused, cryptographically-appropriate random data from a source of hardware randomness. Such random data is inexpensive and easy to generate, even for resource-constrained devices [38]. Fresh random data is then assigned to new keys as needed. We keep used keys logically-fixed because their corresponding data node has already stored—immutably until an erasure operation—its logical position. The new version of the block is then written to an arbitrary empty erase block on the storage medium. After completion, all erase blocks containing old versions of the logical KSA erase block are erased, thus securely deleting the unused and deleted keys along with the data nodes they encrypt.

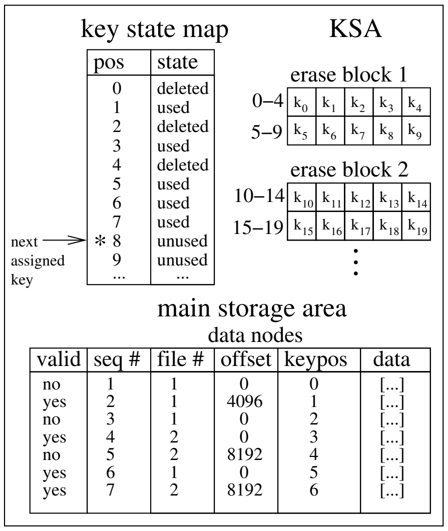
The security of our system necessitates that the storage medium can be properly instructed to erase an erase block. Therefore, for flash memory, DNEFS must be implemented either into the logic of a file system that provides access to the raw flash memory (e.g., UBIFS) or into the logic of the flash controller (e.g., solid state drive). As Swanson et al. [36] observe, any implementation of secure deletion on top of an opaque flash controller cannot guarantee deletion, as its interface for erase block erasure is not security focused and may neglect to delete internally created copies of data due to wear levelling. Our use of UBI bypasses obfuscating controllers and allows direct access to the flash memory.

By only requiring the secure deletion of small densely-packed keys, DNEFS securely deletes all the storage medium's deleted data while only erasing a small number of KSA erase blocks. Thus, encryption is used to reduce the number of erasures required to achieve secure deletion. This comes at the cost of assuming a computationally-bounded adversary—an information-theoretic adversary could decrypt the encrypted file data. We replace unused keys with new random data to thwart the peek-a-boo attacker: keys are discarded if they are not used to store data in the same deletion epoch as they are generated.

While DNEFS is designed to batch deleted data nodes, thus erasing fewer erase blocks per deleted data node, there is no technical reason that prohibits immediate secure deletion. In particular, files can be marked as sensitive [2] so that purging is triggered whenever a data node for such a file is deleted, resulting in one erase block erasure. Purging can also be triggered by an application, for example after it clears its cache.

If a KSA erase block becomes a bad block while erasing it, it is possible that its contents will remain readable on the storage medium without the ability to remove them [21]. In this case, it is necessary to re-encrypt any data node whose encryption key remains available and to force the garbage collection of those erase blocks on which the data nodes reside.

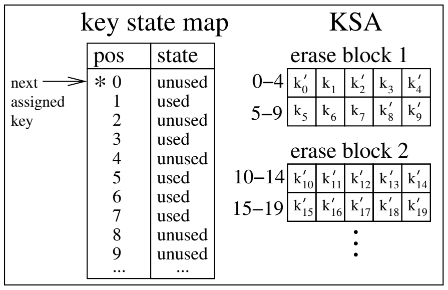
Key State Map. The *key state map* is an in-memory map that maps key positions to key states $\{$*unused*, *used*, *deleted*$\}$. Unused keys can be assigned and then marked as used. Used keys are keys that encrypt some valid data node, so they must be


key state map
KSA
next assigned key
erase block 1
0–4
k_0 k_1 k_2 k_3 k_4
5–9
k_5 k_6 k_7 k_8 k_9
erase block 2
10–14
k_{10} k_{11} k_{12} k_{13} k_{14}
15–19
k_{15} k_{16} k_{17} k_{18} k_{19}
main storage area
data nodes

pos	state
0	deleted
1	used
2	deleted
3	used
4	deleted
5	used
6	used
7	used
* 8	unused
9	unused
...	...

valid	seq #	file #	offset	keypos	data
no	1	1	0	0	[...]
yes	2	1	4096	1	[...]
no	3	1	0	2	[...]
yes	4	2	0	3	[...]
no	5	2	8192	4	[...]
yes	6	1	0	5	[...]
yes	7	2	8192	6	[...]

(a) state before purging keys


key state map
KSA
next assigned key
erase block 1
0–4
k'_0 k_1 k'_2 k_3 k'_4
5–9
k_5 k_6 k_7 k'_8 k'_9
erase block 2
10–14
k'_{10} k'_{11} k'_{12} k'_{13} k'_{14}
15–19
k'_{15} k'_{16} k'_{17} k'_{18} k'_{19}

pos	state
* 0	unused
1	used
2	unused
3	used
4	unused
5	used
6	used
7	used
8	unused
9	unused
...	...

(b) state after purging keys

Figure 4: Example of a key state map, key storage area, and main storage area during a purging operation. (a) shows the state before and (b) shows the state after purging. Some keys are replaced with new values after purging, corresponding to data nodes that were unused or deleted. The table of data nodes illustrate a log-structured file system, where newer versions of data nodes for the same file/offset invalidate older versions.

preserved to ensure availability of the file system's data. Deleted keys are keys used to encrypt deleted data—i.e., data nodes that are no longer referenced by the index—and should be purged from the system to achieve secure deletion. Figure 4 shows an example key state map and a KSA before and after a purging operation: unused and deleted keys are replaced with new values and used keys remain on the storage medium.

While mounting, the key state map must be correctly constructed; the procedure for this depends on the file system in which it is integrated. However, log-structured file systems are capable of generating a file system *index* data structure that maps data nodes to their (most recently written) location in flash memory. We require only that the file system also determines the key location for the data node in the index, and so the state of each key position can be generated by marking these key locations as used and assuming all other locations are deleted.

We define a *correct* key state map as one that has (with high probability) the following three properties: (1) every unused key must not decrypt any data node—either valid or invalid, (2) every used key must have exactly one data node it can decrypt and this data node must be referred to by the index, and (3) every deleted key must not decrypt any data node that is referred to by the index. Observe that an unused key that is marked as deleted will still result in a correct key state map, as it affects neither the security of deleted data nor the availability of used data.

The operation of purging performed on a correct key state map guarantees DNEFS's soundness: purging securely deletes any key in the KSA marked as deleted; afterwards, every key decrypts at most one valid data node, and every data node referred to by the index can be decrypted. While the encrypted version of the deleted data node still resides in flash memory, our adversary is thereafter unable to obtain the key required to decrypt and thus read the data. A correct key state map also guarantees the integrity of our data during purging, because no key that is used to decrypt valid data will be removed.

We define DNEFS's purging epoch's duration (Section 2) as the time between two consecutive purging operations. When a data node is written, it is assigned a key that is currently marked as unused in the current purging epoch. The purging operation's execution at the purging epochs' boundaries ensures that all keys currently marked as unused were not available in any previous purging epoch. Therefore, a peek or boo attack that occurs in any prior purging epoch reveals neither the encrypted data node nor its encryption key. When data is deleted, its encryption key is marked as deleted in the current purging epoch. Purging's execution before the next purging epoch guarantees that key marked as deleted in one epoch is unavailable in the KSA in the next epoch. Therefore, a peek or boo attack that occurs in any later purging epoch may reveal the encrypted data node but not the key. A computationally-bounded peek-a-boo attacker is unable to decrypt the data node, ensuring that the data is not recoverable and therefore securely deleted.

Conclusion. DNEFS provides guaranteed secure deletion against a computationally-bounded peek-a-boo attacker. When an encryption key is securely deleted, the data it encrypted is then inaccessible, even to the user. All invalid data nodes have their corresponding encryption keys securely deleted during the next purging operation. Purging occurs periodically, so during normal operation the deletion latency *for all* data is bounded by this period. Neither the key nor the data node is available in any purging epoch prior to the one in which it is written, preventing any early peek attacks from obtaining this information.

4 UBIFSec

We now describe UBIFSec: our instantiation of DNEFS for the UBIFS file system. We first give an overview of the aspects of UBIFS relevant for integrating our solution. We then describe UBIFSec and conclude with an experimental validation.

4.1 UBIFS

UBIFS is a log-structured flash file system, where all file system updates occur out of place. UBIFS uses an index to determine which version of data is the most recent. This index is called the Tree Node Cache (TNC), and it is stored both in volatile memory and on the storage medium. The TNC is a B+ search tree [7] that has a small entry for every data node in the file system. When data is appended to the journal, UBIFS updates the TNC to reference its location. UBIFS implements truncations and deletions by appending special non-data nodes to the journal. When the TNC processes these nodes, it finds the range of TNC entries that correspond to the truncated or deleted data nodes and removes them from the tree.

UBIFS uses a commit and replay mechanism to ensure that the file system can be mounted after an unsafe unmounting without scanning the entire device. Commit periodically writes the current TNC to the storage medium, and starts a new empty journal. Replay loads the most recently-stored TNC into memory and chronologically processes the journal entries to update the stale TNC, thus returning the TNC to the state immediately before unmounting.

UBIFS accesses flash memory through UBI's logical interface, which provides two features useful for our purposes. First, UBI allows updates to KSA erase blocks (called KSA LEB's in the context of UBIFSec) using its atomic update feature; after purging, all used keys remain in the same *logical* position, so references to KSA positions remain valid after purging. Second, UBI handles wear-levelling for all the PEBs, including the KSA. This is useful because erase blocks assigned to the KSA see more frequent erasure; a fixed physical assignment would therefore present wear-levelling concerns.

4.2 UBIFSec Design

UBIFSec is a version of UBIFS that is extended to use DNEFS to provide secure data deletion. UBIFS's data nodes have a size of 4096 bytes, and our solution assigns each of them a distinct 128-bit AES key. AES keys are used in counter mode, which turns AES into a semantically-secure stream cipher [20]. Since each key is only ever used to encrypt a single block of data, we can safely omit the generation and storage of initialization vectors (IVs) and simply start the counter at zero. Therefore, our solution requires about 0.4% of the storage medium's capacity for the KSA, although there exists a tradeoff between KSA size and data node granularity, which we discuss in Section 4.3.

Key Storage Area. The KSA is comprised of a set of LEBs that store random data used as encryption keys. When the file system is created, cryptographically-suitable random data is written from a hardware source of randomness to each of the KSA's LEBs and all the keys are marked as unused. Purging writes new versions of the KSA LEBs using UBI's atomic update feature; immediately after, `ubi_flush` is called to ensure all PEBs containing old versions of the LEB are synchronously erased and the purged keys are inaccessible. This flush feature ensures that any copies of LEBs made as a result of internal wear-levelling are also securely deleted. Figure 5 shows the LEBs and PEBs during a purging operation; KSA block 3 temporarily has two versions stored on the storage medium.

Key State Map. The key state map (Section 3.2) stores the key positions that are unused, used, and deleted. The correctness of the key state map is critical in ensuring the soundness of secure deletion and data integrity. We now describe how the key state map is created and stored in UBIFSec. As an invariant, we require that UBIFSec's key state map is always correct before and after executing a purge. This restriction—instead of requiring correctness at all times after mounting—is to allow writing new data during a purging operation, and to account for the time between marking a key as used and writing the data it encrypts onto the storage medium.

The key state map is stored, used, and updated in volatile memory. Initially, the key state map of a freshly-formatted UBIFSec file system is correct as it

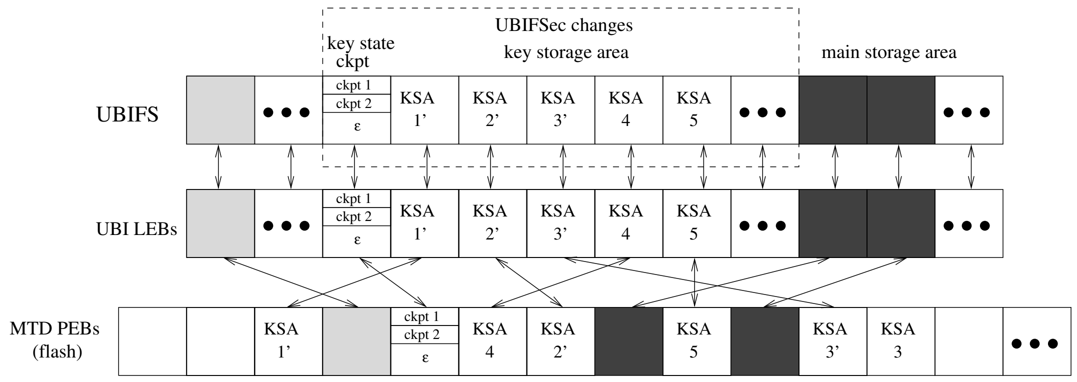


Figure 5: Erase block relationships among MTD, UBI, and UBIFSec, showing the new regions added by UBIFSec (cf. Figure 1). In this example, a purging operation is ongoing—the first three KSA LEBs have been updated and the remaining LEBs still have their old value. In the MTD layer, an old version of KSA 3 is temporarily available.

consists of no data nodes, and every key is fresh random data that is marked as unused. While mounted, UBIFSec performs appropriate key management to ensure that the key state map is always correct when new data is written, deleted, etc. We now show that we can always create a correct key state map when mounting an arbitrary UBIFSec file system.

The key state map is built from a periodic checkpoint combined with a replay of the most recent changes while mounting. We checkpoint the current key state map to the storage medium whenever the KSA is purged. After a purge, every key is either unused or used, and so a checkpoint of this map can be stored using one bit per key—less than 1% of the KSA's size—which is then compressed. A special LEB is used to store checkpoints, where each new checkpoint is appended; when the erase block is full then the next checkpoint is written at the beginning using an atomic update.

The checkpoint is correct when it is written to the storage medium, and therefore it is correct when it is loaded during mounting if no other changes occurred to the file system. If the file system changed after committing and before unmounting, then UBIFS's replay mechanism is used to generate the correct key state map: first the checkpoint is loaded, then the replay entries are simulated. Therefore, we always perform purging during regular UBIFS commits; the nodes that are replayed for UBIFS are exactly the ones that must be replayed for UBIFSec. If the stored checkpoint gets corrupted, then a full scan of the valid data nodes rebuilds the correct key state map. A consistency check for the file system also confirms the correctness of the key state map with a full scan.

As it is possible for the storage medium to fail during the commit operation (e.g., due to a loss of power), we now show that our invariant holds regardless of the condition of unmounting. Purging consists of atomically updating each LEB containing deleted keys and afterwards writing a new checkpoint. UBI's atomic update feature ensures that any failure before completing the update is equivalent to failing immediately before beginning. Therefore, the following is the complete list of possible failure points: before the first purge, between some purges, after all the purges but before the checkpoint, during the checkpoint, or after the checkpoint but before finishing other UBIFS commit actions.

First, failure can occur before purging the first LEB, which means the KSA is unchanged. When remounting the device, the loaded checkpoint is updated with the replay data, thereby constructing the exact key state map before purging—taken as correct by assumption.

Second, failure can occur after purging one, several, or indeed all of the KSA's LEBs. When remounting the device, the loaded checkpoint merged with the replay data reflects the state before the first purge, so some purged LEBs contain new unused data while the key state map claims it is a deleted key. As these are cryptographically-suitable random values, with high probability they cannot successfully decrypt any existing valid data node.

Third, failure can occur while writing to the checkpoint LEB. When the checkpoint is written using atomic updates, then failing during the operation is equivalent to failing before it begins (cf. previous case). Incomplete checkpoints are detected and so the previous valid checkpoint is loaded instead. After replaying all the nodes, the key state map is equal to its state immediately before purging the KSA. This means that all entries marked as deleted are actually unused entries, so the invariant holds.

Old ckpt value	Replay's effect	Ckpt value	Value after recovery	Cause	Key's state
unused	nothing	unused	unused	no event	correct
unused	mark used	used	used	key assigned	correct
unused	mark deleted	unused	deleted	key assigned, deleted	correct
used	nothing	used	used	no event	correct
used	mark used	used	used	cannot occur	correct
used	mark deleted	unused	deleted	key deleted	correct

Table 1: Consequences of replaying false information during committing.

Finally, failure can occur after successfully purging the KSA and checkpointing the key state map, but before completing the regular UBIFS commit. In this case, the current key state map correctly reflects the contents of the KSA. When mounting, the replay mechanism incorrectly updates it with the journal entries of the previous iteration. Table 1 shows the full space of possibilities when replaying old changes on the post-purged checkpoint. It shows that it is only possible for an unused key to be erroneously marked as deleted, which still results in a correct key state map.

In summary, the correctness of the key state map before and after a purge is invariant, regardless of when or how the file system was unmounted. This ensures secure deletion's soundness as well as the integrity of the valid data on the storage medium.

Summary. UBIFSec instantiates DNEFS for UBIFS, and so it provides efficient fine-grained guaranteed secure deletion. UBIFSec is efficient in storage space: the overhead for keys is fixed and it needs less than one percent of the total storage medium's capacity. The periodic checkpointing of UBIFSec's key state map ensures that UBIFS's mounting time is not significantly affected by our approach.

Our implementation of UBIFSec is available as a Linux kernel patch for version 3.2.1 [37]. As of the time of writing, we are in the process of integrating UBIFSec into the standard UBIFS distribution.

4.3 Experimental Validation

We have patched an Android Nexus One smart phone's Linux kernel to include UBIFSec and modified the phone to use it as the primary data partition. In this section, we describe experiments with our implementation on both the Android mobile phone and on a simulator.

Our experiments measure our solution's cost: additional battery consumption, wear on the flash memory, and time required to perform file operations. The increase in flash memory wear is measured using a simulator, and the increase in time is measured on a Google Nexus One smartphone by instrumenting the source code of UBIFS and UBIFSec to measure the time it takes to perform basic file system operations. We further collected timing measurements from the same smartphone running YAFFS: the flash file system currently used on Android phones.

Android Implementation. To test the feasibility of our solution on mobile devices, we ported UBIFSec to the Android OS. The Android OS is based on the Linux kernel and it was straightforwards to add support for UBIFS. The source code was already available and we simply applied our patch and configured the kernel compiler to include the UBI device and the UBIFS file system.

Wear Analysis. We measured UBIFSec's wear on the flash memory in two ways: the number of erase cycles that occur on the storage medium, and the distribution of erasures over the erase blocks. To reduce the wear, it is desirable to minimize the number of erasures that are performed, and to evenly spread the erasures over the storage medium's erase blocks.

We instrumented both UBIFS and UBIFSec to measure PEB erasure frequency during use. We varied UBIFSec's purging frequency and computed the resulting erase block allocation rate. This was done by using a low-level control (`ioctl`) to force UBIFS to perform a commit. We also measured the expected number of deleted keys and updated KSA LEBs during purging operation.

We simulated the UBI storage medium based on Nexus One specifications [11]. We varied the period between UBIFSec's purging operation, i.e., the duration of a purging epoch: one of 1, 5, 15, 30, and 60 minutes. We used a discrete event simulator to write files based on the writing behaviour collected from an Android mobile phone [32]. Writing was performed until the file system began garbage collection; thenceforth we took measurements for a week

Purge period	PEB erasures per hour	Updates per KSA purge	KSA updates per hour	Deleted keys per purged LEB	Wear ineq (%)	Lifetime (years)
Stardard UBIFS	21.3 ± 3.0	-	-	-	16.6 ± 0.5	841
60 minutes	26.4 ± 1.5	6.8 ± 0.5	6.8 ± 0.5	64.2 ± 9.6	17.9 ± 0.2	679
30 minutes	34.9 ± 3.8	5.1 ± 0.6	9.7 ± 2.0	50.3 ± 9.5	17.8 ± 0.3	512
15 minutes	40.1 ± 3.6	3.7 ± 0.4	14.9 ± 1.6	36.3 ± 8.2	19.0 ± 0.3	447
5 minutes	68.5 ± 4.4	2.6 ± 0.1	30.8 ± 0.7	22.1 ± 4.3	19.2 ± 0.5	262
1 minute	158.6 ± 11.5	1.0 ± 0.1	61.4 ± 4.6	14.1 ± 4.4	20.0 ± 0.2	113

Table 2: Wear analysis for our modified UBIFS file system. The expected lifetime is based on the Google Nexus One phone's data partition, which has 1571 erase blocks with a (conservative) lifetime estimate of 10^4 erasures.

of simulated time. We averaged the results from four attempts and computed 95% confidence intervals.

To determine if our solution negatively impacts UBI's wear levelling, we performed the following experiment. Each time UBI unmaps an LEB from a PEB (thus resulting in an erasure) or atomically updates an LEB (also resulting in an erasure), we logged the erased PEB's number. From this data, we then compute the PEBs' erasure distribution.

To quantify the success of wear-levelling, we use the Hoover economic wealth inequality indicator—a metric that is independent of the storage medium size and erasure frequency. This metric comes from economics, where it quantifies the unfairness of wealth distributions. It is the simplest measure, corresponding to an appropriately normalized sum of the difference of each measurement to the mean. For our purposes, it is the fraction of erasures that must be reassigned to other erase blocks to obtain completely even wear. Assuming the observations are $c_1, \ldots, c_n$, and $C = \sum_{i=1}^{n} c_i$, then the inequality measure is $\frac{1}{2} \sum_{i=1}^{n} \| \frac{c_i}{C} - \frac{1}{n} \|$.

Table 2 presents the results of our experiment. We see that the rate of block allocations increases as the purging period decreases, with 15 minutes providing a palatable tradeoff between additional wear and timeliness of deletion. The KSA's update rate is computed as the product of the purging frequency and the average number of KSA LEBs that are updated during a purge. As such, it does not include the additional costs of executing UBIFS commit, which is captured by the disparity in the block allocations per hour. We see that when committing each minute, the additional overhead of committing compared to the updates of KSA blocks becomes significant. While we integrated purging with commit to simplify the implementation, it is possible to separate these operations. Instead, UBIFSec can add purging start and finish nodes as regular (non-data) journal entries. The replay mechanism is then extended to correctly update the key state map while processing these purging nodes.

The expected number of keys deleted per purged KSA LEB decreases sublinearly with the purging period and linearly with the number of purged LEBs. This is because a smaller interval results in fewer expected deletions per interval and fewer deleted keys.

Finally, UBIFSec affects wear-levelling slightly, but not significantly. The unfairness increases with the purging frequency, likely because the set of unallocated PEBs is smaller than the set of allocated PEBs; very frequent updates will cause unallocated PEBs to suffer more erasures. However, the effect is slight. It is certainly the case that the additional block erasures are, for the most part, evenly spread over the device.

Throughput and Battery Analysis A natural concern is that UBIFSec might introduce significant costs that discourage its use. We therefore experimentally evaluated the read/write throughput, battery consumption, and computation time of UBIFSec's Android implementation (Linux version 2.6.35.7) on a Google Nexus One mobile phone. We compare measurements taken for both Android's default file system (YAFFS) and for the standard version of UBIFS.

To measure battery consumption over time, we disabled the operating system's suspension ability, thus allowing computations to occur continuously and indefinitely. This has the unfortunate consequence of maintaining power to the screen of the mobile phone. We first determined the power consumption of the device while remaining idle over the course of two hours starting with an 80% charged battery with a total capacity of 1366 mAh. The result was nearly constant at 121 mA. We subtract this value from all other power consumption measures.

To measure read throughput and battery use, we repeatedly read a large (85 MiB) file; we mounted the drive as read-only and remounted it after each read to ensure that all read caches were cleared. We read the file using `dd`, directing the output to `/dev/null` and recorded the observed throughput.

We began each experiment with an 80% charged battery and ran it for 10 minutes observing constant behaviour. Table 3 presents the results for this experiment. For all filesystems, the additional battery consumption was constant: 39 mA, about one-third of the idle cost. The throughput achieved with that power varied, and so along with our results we compute the amount of data that can be read using 13.7 mAh—1% of the Nexus One's battery. The write throughput and battery consumption was measured by using `dd` to copy data from `/dev/zero` to a file on the flash file system. Compression was disabled for UBIFS for comparison with YAFFS. When the device was full, the throughput was recorded. We immediately started `dd` to write to the same file, which begins by overwriting it and thus measuring the battery consumption and reduction in throughput imposed by erase block erasure concomitant with writes.

We observe that the use of UBIFSec reduces the throughput for both read and write operations when compared to UBIFS. Some decrease is expected, as the encryption keys must be read from flash while reading and writing. To check if the encryption operations also induce delay, we performed these experiments with a modified UBIFSec that immediately returned zeroed memory when asked to read a key, but otherwise performed all cryptographic operations correctly. The resulting throughput for read and write was identical to UBIFS, suggesting that (for multiple reads) cryptographic operations are easily pipelined into the relatively slower flash memory read/write operations.

Some key caching optimizations can be added to UBIFSec to improve the throughput. Whenever a page of flash memory is read, the entire page can be cached at no additional read cost, allowing efficient sequential access to keys, e.g., for a large file. Long-term use of the file system may reduce its efficiency as gaps between used and unused keys result in new files not being assigned sequential keys. Improved KSA organization can help retain this efficiency.

Write throughput, alternatively, is easily improved with caching. The sequence of keys for data written in the next purging epoch is known at purging time when all these keys are randomly generated and written to the KSA. By using a heuristic on the expected number of keys assigned during a purging epoch, the keys for new data can be kept in memory as well as written to the KSA. Whenever a key is needed, it is taken and removed from this cache while there are still keys available.

Caching keys in memory opens UBIFSec to attacks. We ensure that all memory buffers containing keys are overwritten when the key is no longer needed during normal decryption and encryption operations. Caches contain keys for a longer time but are cleared during a purging operation to ensure deleted keys never outlive their deletion purging epoch. Applications storing sensitive data in volatile memory may remain after the data's deletion and so secure memory deallocation should be provided by the operating system to ensure its unavailability [5].

	YAFFS	UBIFS	UBIFSec
Read rate (MiB/s)	4.4	3.9	3.0
Power usage (mA)	39	39	39
GiB read per %	5.4	4.8	3.7
Write rate (MiB/s)	2.4	2.1	1.7
Power usage (mA)	30	46	41
GiB written per %	3.8	2.2	2.0

Table 3: I/O throughput and battery consumption for YAFFS, UBIFS, and UBIFSec.

Timing Analysis. We timed the following file system functions: mounting/unmounting the file system and writing/reading a page. Additionally, we timed the following functions specific to UBIFSec: allocation of the cryptographic context, reading the encryption key, performing an encryption/decryption, and purging a KSA LEB. We collected dozens of measurements for purging, mounting and unmounting, and hundreds of measurements for the other operations (i.e., reading and writing). We controlled the delay caused by our instrumentation by repeating the experiments instead of executing nested measurements, i.e., we timed encryption and writing to a block in separate experiments.

We mounted a partition of the Android's flash memory first as a standard UBIFS file system and then as UBIFSec file system. We executed a sequence of file I/O operations on the file system. We collected the resulting times and present the 80th percentile measurements in Table 4. Because of UBIFS's implementation details, the timing results for reading data nodes contain also the time required to read relevant TNC pages (if they are not currently cached) from the storage medium, which is reflected in the increased delay. Because the data node size for YAFFS is half that of UBIFS, we also doubled the read/write measurements for YAFFS for comparison. Finally, the mounting time for YAFFS is for mounting after a safe unmount—for an unsafe unmount, YAFFS requires a full device scan, which takes several orders of magnitude longer.

The results show an increase in the time required for each of the operations. Mounting and unmount-

File system	80th percentile execution time (ms)		
operation	YAFFS	UBIFS	UBIFSec
mount	43	179	236
unmount	44	0.55	0.67
read data node	0.92	2.8	4.0
write data node	1.1	1.3	2.5
prepare cipher	-	-	0.05
read key	-	-	0.38
encrypt	-	-	0.91
decrypt	-	-	0.94
purge one block	-	-	21.2

Table 4: Timing results for various file system functionality on an android mobile phone.

ing the storage medium continues to take a fraction of a second. Reading and writing to a data node increases by a little more than a millisecond, an expected result that reflects the time it takes to read the encryption key from the storage medium and encrypt the data. We also tested for noticeable delay by watching a movie in real time from a UBIFSec-formatted Android phone running the Android OS: the video was 512x288 Windows Media Video 9 DMO; the audio was 96.0 kbit DivX audio v2. The video and audio played as expected on the phone; no observable latency, jitter, or stutter was observed during playback while background processes ran normally.

Each atomic update of an erase block takes about 22 milliseconds. This means that if every KSA LEB is updated, the entire data partition of the Nexus One phone can be purged in less than a fifth of a second. The cost to purge a device grows with its storage medium's size. The erasure cost for purging can be reduced in a variety of ways: increasing the data node size to use fewer keys, increasing the duration of a purging epoch, or improving the KSA's organization and key assignment strategy to minimize the number of KSA LEBs that contain deleted keys. The last technique works alongside lazy on-demand purging of KSA LEBs that contain no deleted keys, i.e., only used and unused keys.

Granularity Tradeoff Our solution encrypts each data node with a separate key allowing efficient secure deletion of data from long-lived files, e.g., databases. Other related work instead encrypts each file with a unique key, allowing secure deletion only at the granularity of an entire file [19]. This is well suited for media files, such as digital audio and photographs, which are usually created, read, and deleted in their entirety. However, if the encrypted file should permit random access and modification,

Data node size	KSA size	Copy cost
(flash pages)	(EBs per GiB)	(EBs)
1	64	0
8	8	0.11
64	1	0.98
512	0.125	63.98
4096	0.016	511.98

Table 5: Data node granularity tradeoffs assuming 64 2-KiB pages per erase block.

then one of the following is true: (i) the cipher is used in an ECB-like mode, resulting in a system that is not semantically secure, (ii) the cipher is used in a CBC-like mode where all file modifications require re-encryption of the remainder of the file, (iii) the cipher is used in a CBC-like mode with periodic IVs to facilitate efficient modification, (iv) the cipher is used in counter mode, resulting in all file modifications requiring rewriting the entire file using a new IV to avoid the two-time pad problem [20], or (v) the cipher is used in counter mode with periodic IVs to facilitate efficient modifications.

We observe the that first option is inadequate as a lack of semantic security means that some information about the securely deleted data is still available. The second and fourth options are special cases of the third and fifth options respectively, where the IV granularity is one per file and file modifications are woefully inefficient. Thus, a tradeoff exists between the storage costs of IVs and additional computation for modifications. As the IV granularity decreases to the data node size, the extra storage cost required for IVs is equal to the KSA storage cost for DNEFS's one key per data node, and the modification cost is simply that of the single data node.

We emphasize that a scheme where IVs were not stored but instead deterministically computed, e.g., using the file offset, would inhibit secure deletion: so long as the file's encryption key and previous version of the data node were available, the adversary could compute the IV and decrypt the data. Therefore, all IVs for such schemes must be randomly generated, stored, and securely deleted.

Table 5 compares the encryption granularity trade off for a flash drive with 64 2-KiB pages per erase block. To compare DNEFS with schemes that encrypt each file separately, simply consider the data node size as equal to the IV granularity or the expected size file size. The KSA size, measured in erase blocks per GiB of storage space, is the amount of storage required for IVs and keys, and is the worst case number of erase blocks that must be erased during each purging operation. The copy cost, also measured in erase blocks, is the amount of data that must be re-written to the flash storage medium due to a data node modification that affects only one page of flash memory. For example, with a data node size of 1024 KiB and a page size of 2 KiB, the copy cost for a small change to the data node is 1022 KiB. This is measured in erase blocks because the additional writes, once filling an entire erase block, result in an additional erase block erasure, otherwise unnecessary with a smaller data node size.

As we observed earlier, reducing the number of keys required to be read from flash per byte of data improves read and write throughput. From these definitions, along with basic geometry of the flash drive, it is easy to compute the values presented in Table 5. When deploying DNEFS, the administrator can choose a data node size by optimizing for the costs given how frequently small erasures and complete purges are executed.

5 Extensions and Optimizations

Compatibility with FTLs. The most widely-deployed interface for flash memory is the Flash Translation Layer (FTL) [1], which maps logical block device sectors (e.g., a hard drive) to physical flash addresses. While FTLs vary in implementation, many of which are not publicly available, in principle DNEFS can be integrated with FTLs in the following way. All file-system data is encrypted before being written to flash, and decrypted whenever it is read. A key storage area is reserved on the flash memory to store keys, and key positions are assigned to data. The FTL's in-memory logical remapping of sectors to flash addresses must store alongside a reference to a key location. The FTL mechanism that rebuilds its logical sector to address mapping must also rebuild the corresponding key location. Key locations consist of a *logical* KSA erase block number and the actual offset inside the erase block. Logically-referenced KSA erase blocks are managed by storing metadata in the final page of each KSA erase block. This page is written immediately after successfully writing the KSA block and stores the following information: the logical KSA number so that key references need not be updated after purging, and an epoch number so that the most recent version of the KSA block is known. With this information, the FTL is able to replicate the features of UBI that DNEFS requires.

Generating a correct key state map when mounting is tied to the internal logic of the FTL. Assuming that the map of logical to physical addresses along with the key positions is correctly created, then it is trivial to iterate over the entries to mark the corresponding keys as used. The unmarked positions are then purged to contain new data. The FTL must also generate cryptographically-secure random data (e.g., with an accelerometer [38]) or be able to receive it from the host. Finally, the file system mounted on the FTL must issue TRIM commands [16] when a sector is deleted, as only the file system has the semantic context to know when a sector is deleted.

Purging Policies. Purging is currently performed after a user-controlled period of time and before unmounting the device. More elaborate policies are definable, where purging occurs once a threshold of deleted keys is passed, ensuring that the amount of exposable data is limited, so the deletion of many files would thus act as a trigger for purging. A low-level control allows user-level applications to trigger a purge, such as an email application that purges the file system after clearing the cache. We can alternatively use a new extended attribute to act a trigger: whenever any data node belonging to a sensitive file is deleted, then DNEFS triggers an immediate purge. This allows users to have confidence that most files are periodically deleted, while sensitive files are promptly deleted.

Securely Deleting Swap. A concern for secure deletion is to securely delete any copies of data made by the operating system. Data that is quite large may be written to a swap file—which may be on the same file system or on a special cache partition. We leave as future work to integrate our solution to a secure deleting cache. (There exist encrypted swap partitions [31], but not one that securely deletes the memory when it is deallocated.) We expect it to be simple to design, as cache data does not need to persist if power is lost; an encryption-based approach can keep all the keys in volatile memory and delete them immediately when they are no longer needed.

Encrypted File System. Our design can be trivially extended to offer a passphrase-protected encrypted file system: we simply encrypt the KSA whenever we write random data, and derive the decryption key from a provided passphrase when mounting. Since, with high probability, each randomly-generated key in the KSA is unique, we can use a block cipher in ECB mode to allow rapid decryption of randomly accessed offsets without storing additional initialization vectors [20].

Encrypted storage media are already quite popular, as they provide confidentiality of stored data against computationally-bounded non-coercive attackers, e.g., thieves, provided the master secret is unavailable in volatile memory when the attack occurs [13]. It is therefore important to offer our encrypted file system design to avoid users doubly-encrypting their data: first as an encrypted file system and then for secure deletion.

6 Related Work

Secure deletion for magnetic media is a well-researched area. Various solutions exist at different levels of system integration. User-level solutions such as *shred* [29] open a file and overwrite its entire content with insensitive data. This requires that the file system performs in-place updates, otherwise old data still remains. As such, these solutions are inappropriate for flash memory.

Kernel-level secure deletion has been designed and implemented for various popular block-structured file systems [2, 17]. These solutions typically change the file system so that whenever a block is discarded, its content is first sanitized before it is added to the list of free blocks. This ensures that even if a file is truncated or the user forgets to use a secure deletion tool, the data is still sanitized. It is also beneficial for journalled file systems where in-place updates do not immediately occur, so overwriting a file may not actually overwrite the original content. The sanitization of discarded blocks still requires in-place updates, and is therefore inapplicable to flash memory.

The use of encryption to delete data was originally proposed by Boneh and Lipton [3], where they used the convenience of deleting small encryption keys to computationally-delete data from magnetic backup tapes. Peterson et al. [28] used this approach to improve the efficiency of secure deletion in a versioned backup system on a magnetic storage medium. They encrypt each data block with an all-or-nothing cryptographic expansion transformation [33] and colocate the resulting key-sized tags for every version of the same file in storage. They use in-place updates to remove tags, and keys are colocated to reduce the cost of magnetic disk seek times when deleting all versions of a single file. DNEFS also colocates keys separately to improve the efficiency of secure deletion. However, DNEFS is prohibited from performing in-place updates, and our design focuses on minimizing and levelling erasure wear.

Another approach is Perlman's Ephemerizer [27], a system that allows communication between parties where messages are securely deleted in the presence of a coercive attacker. Data is encrypted with ephemeral keys that are manged by the eponymous trusted third party. Each message is given an expiration time at creation, and an appropriate key is generated accordingly. The Ephemerizer stores the keys and provides them when necessary to the communicating parties using one-time session keys. When the expiration time passes, it deletes any material required to create the key, thus ensuring secure deletion of the past messages. Perlman's work is a protocol using secure deletion as an assumed primitive offered by the storage medium. Indeed, if this approach is implemented on a flash-based smart card, DNEFS can be used to implement it.

Reardon et al. [32] have shown how to securely delete data from log-structured file systems from user-space—that is, without modifying the hardware or the file system—provided that the user can access the flash directly and is not subjected to disk quota limitations. Their proposal is to fill the storage medium to capacity to ensure that no wasted space remains on the storage medium, thus ensuring the secure deletion of data. This is a costly approach in terms of flash memory wear and execution time, but from user-space it is the only solution possible. They also showed that occupying a large segment of the storage medium with unneeded data improves the expected time deleted data remains on the storage medium, and reduces the amount of space that needs to be filled to guarantee deletion.

Swanson et al. [36] considered verifiable sanitization for solid state drives—flash memory accessed through an opaque FTL. They observed that the manufacturers of the controller are unreliable even when implementing their own advertised sanitization procedure, and that the use of cryptography is insufficient when the ultimate storage location of the cryptographic key cannot be determined from the logical address provided by the FTL. They propose a technique for static sanitization of the entire flash memory—that is, all the storage medium's contained information is securely removed. It works by originally encrypting all the data on the drive before being written. When a sanitize command is issued, first the memory containing the keys is erased, and then every page on the device is written and erased. Our solution focuses on the more typical case of a user wanting to securely delete some sensitive data from their storage media but not wanting to completely remove all data available on the device. Our solution requires access to the raw flash, or a security-aware abstraction such as UBI that offers the `ubi_flush()` function to synchronously remove all previous versions (including copies) of a particular LEB number.

Wei et al. [39] have considered secure deletion on flash memory accessed through an FTL (cf. Section 2). They propose a technique, called *scrubbing*, which writes zeros over the pages of flash memory without first erasing the block. This sanitizes the data because, in general, flash memory requires an erasure to turn a binary zero to a binary one, but writes turn ones into zeros. Sun et al. [35] propose a hybrid secure deletion method built on Wei et al.'s scheme, where they also optimize the case when there is less data to copy off a block then data to be zero overwritten.

Scrubbing securely deletes data immediately, and no block erasures must be performed. However, it requires programming a page multiple times between erasures, which is not appropriate for flash memory [22]. In general, the pages of an erase block must be programmed sequentially [26] and only once. An option exists to allow multiple programs per page, provided they occur at different positions in the page; multiple overwrites to the same location officially result in undefined behaviour [26]. Flash manufacturers prohibit this due to *program disturb* [21]: bit errors that can be caused in spatially proximate pages while programming flash memory.

Wei et al. performed experiments to quantify the rate at which such errors occur: they showed that they do exist but their frequency varies widely among flash types, a result also confirmed by Grupp et al. [12]. Wei et al. use the term scrub budget to refer to the number of times that the particular model of flash memory has experimentally allowed multiple overwrites without exhibiting a significant risk of data errors. When the scrub budget for an erase block is exceeded, then secure deletion is instead performed by invoking garbage collection: copying all the remaining valid data blocks elsewhere and erasing the block. Wei et al. state that modern densely packed flash memories are unsuitable for their technique as they allow as few as two scrubs per erase block [39]. This raises serious concerns for the future utility of Wei et al.'s approach and highlights the importance of following hardware specifications.

Lee et al. [19] propose secure deletion for YAFFS. They encrypt each file with a different key, store the keys in the file's header, and propose to modify YAFFS to colocate file headers in a fixed area of the storage medium. They achieve secure deletion by erasing the erase block containing the file's header, thus deleting the entire file. More recently, Lee et al. [18] built on this approach by extending it to perform standard data sanitization methods prescribed by government agencies (e.g., NSA [25], DoD [24]) on the erase blocks containing the keys.

We can only compare our approach with theirs in design, as their approaches were not implemented. First, the approach causes an erase block deletion for every deleted file. This results in rapid wear for devices that create and delete many small cache files. Reardon et al. [32] observed that the Android phone's normal usage created 10,000 such files a day; if each file triggers an erase block erasure, then this solution causes unacceptable wear on the dedicated segment of the flash memory used for file headers. Their proposal encrypts the entire file before writing it to the storage medium with a single key and without mention of the creation or storage of IVs. (See Section 4.3 for more analysis on per-file versus per-data-node encryption.) Our solution differs from theirs by batching deletions and purging based on an interval, by considering the effect on wear levelling, by allowing fine-grained deletions of overwritten and truncated data, and by being fully implemented.

7 Conclusions

DNEFS and its instance UBIFSec are the first feasible solution for efficient secure deletion for flash memory operating within flash memory's specification. It provides *guaranteed secure deletion* against a computationally-bounded peek-a-boo attacker by encrypting each data node with a different key and storing the keys together on the flash storage medium. The erase blocks containing the keys are logically updated to remove old keys, replacing them with fresh random data that can be used as keys for new data. It is *fine-grained* in that parts of files that are overwritten are also securely deleted.

We have implemented UBIFSec and analyzed it experimentally to ensure that it is *efficient*, requiring a small evenly-levelled increase in flash memory wear and little additional computation time. UBIFSec was *easily added* to UBIFS, where cryptographic operations are added seamlessly to UBIFS's read/write data path, and changes to key state are handled by UBIFS's existing index of data nodes.

Acknowledgments

This work was partially supported by the Zurich Information Security Center. It represents the views of the authors. We would like to thank our anonymous reviews for their many helpful comments and Artem Bityutskiy for his help integrating UBIFSec into UBIFS.

References

[1] Ban, A. Flash file system. US Patent, no. 5404485, 1995.

[2] Bauer, S., and Priyantha, N. B. Secure Data Deletion for Linux File Systems. *Usenix Security Symposium* (2001), 153–164

[3] Boneh, D., and Lipton, R. J. A revocable backup system. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6* (Berkeley, CA, USA, 1996), USENIX Association, pp. 91–96.

[4] Charles Manning. How YAFFS Works. 2010.

[5] Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M. Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), SSYM'05, USENIX Association.

[6] Chung, T.-S., Park, D.-J., Park, S., Lee, D.-H., Lee, S.-W., and Song, H.-J. A survey of Flash Translation Layer. *Journal of Systems Architecture 55*, 5-6 (2009), 332–343.

[7] Cormen, T., Leiserson, C., and Rivest, R. *Introduction to Algorithms.* McGraw Hill, 1998.

[8] Gal, E., and Toledo, S. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys 37* (2005), 138–163.

[9] Garfinkel, S., and Shelat, A. Remembrance of Data Passed: A Study of Disk Sanitization Practices. *IEEE Security & Privacy* (January 2003), 17–27.

[10] Gleixner, T., Haverkamp, F., and Bityutskiy, A. UBI - Unsorted Block Images. 2006.

[11] Google, Inc. Google Nexus Phone.

[12] Grupp, L. M., Caulfield, A. M., Coburn, J., Swanson, S., Yaakobi, E., Siegel, P. H., and Wolf, J. K. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 24–33.

[13] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM 52* (May 2009), 91–98.

[14] Hunter, A. A Brief Introduction to the Design of UBIFS. 2008.

[15] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. 1998.

[16] Intel Corporation. Intel Solid-State Drive Optimizer. 2009.

[17] Joukov, N., Papaxenopoulos, H., and Zadok, E. Secure Deletion Myths, Issues, and Solutions. *ACM Workshop on Storage Security and Survivability* (2006), 61–66.

[18] Lee, B., Son, K., Won, D., and Kim, S. Secure Data Deletion for USB Flash Memory. *Journal of Information Science and Engineering 27* (2011), 933–952.

[19] Lee, J., Yi, S., Heo, J., and Park, H. An Efficient Secure Deletion Scheme for Flash File Systems. *Journal of Information Science and Engineering* (2010), 27–38.

[20] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. *Handbook of Applied Cryptography.* CRC Press, 2001.

[21] Micron, Inc. Design and Use Considerations for NAND Flash Memory Introduction. 2006.

[22] Micron Technology, Inc. Technical Note: Design and Use Considerations for NAND Flash Memory. 2006.

[23] Memory Technology Devices (MTD): Subsystem for Linux. 2008.

[24] National Industrial Security Program Operating Manual. July 1997.

[25] NSA/CSS Storage Device Declassification Manual. November 2000.

[26] Open NAND Flash Interface. Open NAND Flash Interface Specification, version 3.0. 2011.

[27] Perlman, R. The Ephemerizer: Making Data Disappear. Tech. rep., Mountain View, CA, USA, 2005.

[28] Peterson, Z., Burns, R., and Herring, J. Secure Deletion for a Versioning File System. *USENIX Conference on File and Storage Technologies* (2005).

[29] Plumb, C. shred(1) - Linux man page.

[30] Pöpper, C., Basin, D., Capkun, S., and Cremers, C. Keeping Data Secret under Full Compromise using Porter Devices. In *Computer Security Applications Conference* (2010), pp. 241–250.

[31] Provos, N. Encrypting virtual memory. In *Proceedings of the 9th USENIX Security Symposium* (2000), pp. 35–44.

[32] Reardon, J., Marforio, C., Capkun, S., and Basin, D. Secure Deletion on Log-structured File Systems. *7th ACM Symposium on Information, Computer and Communications Security* (2012).

[33] Rivest, R. L. All-Or-Nothing Encryption and The Package Transform. In *Fast Software Encryption Conference* (1997), Springer Lecture Notes in Computer Science, pp. 210–218.

[34] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems 10* (1992), 1–15.

[35] Sun, K., Choi, J., Lee, D., and Noh, S. Models and Design of an Adaptive Hybrid Scheme for Secure Deletion of Data in Consumer Electronics. *IEEE Transactions on Consumer Electronics 54* (2008), 100–104.

[36] Swanson, S., and Wei, M. SAFE: Fast, Verifiable Sanitization for SSDs. October 2010.

[37] UBIFSec Patch.

[38] Voris, J., Saxena, N., and Halevi, T. Accelerometers and randomness: perfect together. In *Proceedings of the fourth ACM conference on Wireless network security* (New York, NY, USA, 2011), WiSec '11, ACM, pp. 115–126.

[39] Wei, M., Grupp, L. M., Spada, F. M., and Swanson, S. Reliably Erasing Data from Flash-Based Solid State Drives. In *Proceedings of the 9th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2011), pp. 105–117.

[40] Woodhouse, D. JFFS: The Journalling Flash File System. In *Ottawa Linux Symposium* (2001).