Sanitizing Data is Not Enough! Towards Sanitizing Structural Artifacts in Flash Media*

Bo Chen[1], Shijie Jia[2,3,4], Luning Xia[2,3], Peng Liu[5]

[1] Department of Computer Science, Michigan Technological University, USA
[2]State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[3]Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China
[4]University of Chinese Academy of Sciences, Beijing, China
[5] College of Information Sciences and Technology, The Pennsylvania State University, USA
bchen@mtu.edu, {jiashijie, halk}@is.ac.cn, pliu@ist.psu.edu

ABSTRACT

Conventional overwriting-based and encryption-based secure deletion schemes can only sanitize data. However, the past existence of the deleted data may leave artifacts in the layout at all layers of a computing system. These structural artifacts may be utilized by the adversary to infer sensitive information about the deleted data or even to fully recover them. The conventional secure deletion solutions unfortunately cannot sanitize them.

In this work, we introduce truly secure deletion, a novel security notion that is much stronger than the conventional secure deletion. Truly secure deletion requires sanitizing both the obsolete data as well as the corresponding structural artifacts, so that the resulting storage layout after a delete operation is indistinguishable from that the deleted data never appeared. We propose TedFlash, a **T**ruly s**e**cure **d**eletion scheme for **Flash**-based block devices. TedFlash can successfully sanitize both the data and the structural artifacts, while satisfying the design constraints imposed for flash memory. Security analysis and experimental evaluation show that TedFlash can achieve the truly secure deletion guarantee with a small additional overhead compared to conventional secure deletion solutions.

1. INTRODUCTION

Securely deleting obsolete data is of paramount importance, as reserving these data may not only endanger data owners' privacy, but also violate retention regulations like HIPAA [19], Gramm-Leach-Bliley Act [13], Sarbanes-Oxley Act [37], and Fifth Directive of The Data Protection Act [12].

To achieve secure data deletion, simply sanitizing the data (i.e., making the data inaccessible by over-writing them or deploying encryption with ephemeral keys) is unfortunately not enough, as the past existence of the data may leave artifacts in the layout of the resulting storage medium at all layers, which may be utilized by the adversary to infer sensitive information about the deleted data or even fully recover them [18, 26]. For example, a CIA document was published by the New York Times in 2000 as a PDF file that contained the original document and an overlay covering up sensitive information in the document. The overlay was then removed by Internet users and the sensitive information about the CIA's role in the 1953 overthrow of Iranian Government was revealed [25].

It thus becomes necessary to sanitize both the obsolete data as well as the structural artifacts introduced by them. Traditional overwriting-based [24] and encryption-based [23, 32, 33, 35, 39, 42] secure data deletion solutions can only sanitize data. However, they cannot sanitize structural artifacts. We thus explore a novel security notion, namely, *truly secure deletion*, that can ensure sanitization of both the data and the structural artifacts. An existing security notion "history independence" [31] is stronger than truly secure deletion. However, we believe that the truly secure deletion is already a very strong security notion. This security notion enables the development of a practical easy-to-deploy data deletion technique which is presented in this paper.

People today are increasingly turning to flash memory for data storage due to its high throughput and decreasing price. Flash-based storage media like SSD drives are used extensively in laptops, even cloud providers allow their users to choose SSDs as the underlying storage media [6]. Other popular flash products like eMMC cards, SD cards, and miniSD cards dominate the storage media of mobile devices.

In this work, we aim to realize truly secure deletion for flash memory, which is in particular challenging due to its special nature: 1) Flash memory is over-write unfriendly. Over-writing a small region (e.g., a 4KB page) requires first erasing a large region (i.e., a 128-KB block); 2) Each flash storage unit can only be programmed/erased for a limited number of times before it is worn out. The special nature of flash memory imposes additional constraints on any scheme specifically designed for flash: (C1) *The number of over-writes should be as less as possible*; (C2) *Writes should be distributed evenly among flash (i.e., wear leveling).*

We propose TedFlash, the first scheme which can sanitize both the data and the structural artifacts from flash and, at the same time, satisfy the aforementioned design constraints. Our key insights are: 1) We write data to flash in such a way that every single write will not introduce additional artifacts to the storage state except the data themselves. This is advantageous, because upon deletion, we can simply sanitize the data, eliminating the need of sanitiz-

*A preliminary version of this technical report was published in ACSAC '16.

	DNEFS [35]	HiFlash [18]	TedFlash
Sanitizing data	Yes	Yes	Yes
Sanitizing structural artifacts	No	Yes	Yes
Satisfying C1	Yes	No	Yes
Satisfying C2	Yes	No	Yes

Table 1: Comparison among representative secure deletion schemes for flash memory

ing the corresponding artifacts, which is probably expensive and may significantly decrease I/O throughput. 2) To ensure every write to flash will not bring in structural artifacts, we always place the data being written to an empty location which is randomly selected (Note that if this is an over-write, the old data will be invalidated and securely sanitized). The random placement technique can eliminate the structural artifacts brought by each write, as the placement of data is independent and does not affect the placements of any other data. Most importantly, the random placement technique is exclusively feasible for flash memory, because: First, random seeks on flash memory are as efficient as sequential seeks, a salient unique characteristic of flash memory compared to mechanical storage media like HDDs. Second, random placements inherently distribute data evenly among flash, leading to good wear leveling.

We design TedFlash in such a way that is compatible with the pervasively deployed "translation layer", which stays between the physical flash layer and the file system layer in literally all computing devices (e.g., smart phones, tablets, and laptops) using flash memory as the underlying storage media. Our design is advantageous as the translation layer is much simpler and contains much less code compared to the file system layer, and retrofitting it will not affect the other layers of the storage system, preserving the independence principle of storage system designs.

Comparisons. We provide in Table 1 a comparison among three representative schemes which can securely delete data from flash, DNEFS [35], HiFlash [18], and TedFlash, respectively. DNEFS was able to sanitize the data from flash, but it is not able to sanitize the structural artifacts. HiFlash was the first design that can provide history independence for flash memory, which can always ensure sanitization of both the data and the structural artifacts. However, it cannot satisfy design constraint C1 or C2, which seems unavoidable due to the strong security requirement of history independence. TedFlash is the first scheme specifically designed to achieve the truly secure deletion guarantee. Most importantly, it can satisfy both design constraint C1 and C2.

Contributions. We summarize our contributions in the following:

– We introduce the first concrete attacks on conventional overwriting-based and encryption-based secure deletion schemes. In our attacks, the adversary can learn sensitive information about the deleted data or even fully recover them by utilizing structural artifacts.

– We define truly secure deletion, a novel security notion that can achieve a much stronger security guarantee than the conventional secure deletion. Truly secure deletion requires sanitizing both the obsolete data as well as the structural artifacts introduced by them.

– We initiate the research of truly secure deletion for flash memory. We propose TedFlash, the first scheme specifically designed to achieve the truly secure deletion guarantee in flash-based block devices. TedFlash can sanitize both the data and the structural artifact, while satisfying the design constraints imposed for flash memory.

– We implement TedFlash in an actual flash device based on an open-source flash firmware.

– We experimentally evaluate TedFlash. Compared to conventional secure deletion for flash memory, our TedFlash can achieve the much stronger truly secure deletion guarantee with a small additional overhead.

2. BACKGROUND

2.1 Flash Memory

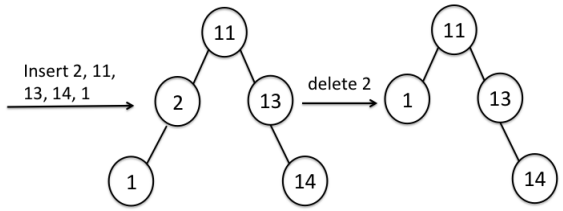
Flash. The flash family contains NAND-type and NOR-type flash. This work concentrates on the NAND-type flash, which is pervasively used in flash-based storage products like SD cards, eMMC cards, USB sticks, and SSD drives. Flash stores information in an array of memory cells. The entire memory cells are grouped into erase blocks. An erase block is the minimal unit of performing erase operations on flash. Each erase block is further divided into a certain number (*e.g.*, 32, 64, or 128) of pages. Typical page size can be 512 bytes, 2KB, and 4KB. A flash page is the I/O unit of NAND flash.

Compared to mechanical disks, a significant difference of flash memory is, the flash cell cannot be re-programmed before it has been erased. Unfortunately, as erase operations can only be performed on a block basis, over-writing a small page requires first erasing a large block. If the remaining pages of this block are filled with valid data, erasing it requires copying the valid data elsewhere and writing them back after the erase operation has been performed, leading to significant *write amplification*. This explains why flash is over-write unfriendly and any designs relevant to flash memory should avoid frequent over-write operations.

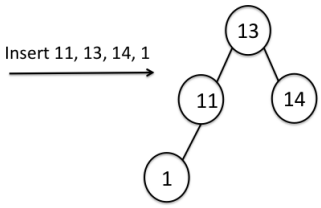
Another significant difference between flash memory and the mechanical disks is, each flash cell has a limited number (e.g., 10K) of program-erase (P/E) cycles before it is worn out and is not stable enough to store information. To prolong the service life of flash memory, *wear leveling* is necessary, by which writes/erasures on flash memory are distributed evenly across it so that no single block has significantly larger P/E cycles than others and fails prematurely.

How to use flash. To be compatible with traditional block-based file systems (e.g., EXT4, FAT32), a flash device is usually emulated as a block device by exposing a block-based access interface, which is the most popular form of flash-based products (e.g., SSDs, eMMCs, SD cards, and USB sticks). This is usually achieved by introducing a special flash firmware, Flash Translation Layer (FTL), between the file system and the raw flash. FTL can translate the logical block addresses to the underlying physical flash addresses, providing a block-based access interface to the upper layer.

Another alternative of using raw flash is to directly build a flash-specific file system over it. Popular flash file systems include YAFFS, UBIFS, JFFS2, and F2FS. However, most of the recent mobile devices are only designed to be compatible with flash-based block devices, and usually do not allow directly accessing the raw flash. For example, the



(a) With deletion



(b) Without deletion

Figure 1: An example showing why structural artifacts matter. T_1, T_2, and T_3 are balanced BSTs

most recent Android smart phones like Nexus 6P use eMMC cards as storage media, and only the old Android phones like Nexus One and Nexus S can allow directly accessing the raw flash.

2.2 Re-thinking "secure deletion"

Conventionally, what "secure deletion" can promise to the data owners is: once the data have been deleted, they will become inaccessible [23, 24, 35, 39]. Most of the prior solutions rely on either over-writing [22, 24, 40] or encryption [23, 32, 33, 35, 39, 42] to make the deleted data *inaccessible*. However, the structural artifacts created by the deleted data, if preserved in the storage media, may enable the adversary to learn sensitive information about the deleted data. In the worse case, the adversary may be able to recover the deleted data by utilizing those structural artifacts and the remaining data (Sec. 3). We believe secure deletion should promise: *once the data have been deleted, the adversary should not be able to learn anything about the deleted data*. Intuitively, if we can ensure after having deleting certain data, the resulting storage state is indistinguishable from that the deleted data never appeared, we should be able to fulfill the aforementioned promise.

2.3 Why Structural Artifacts Matter

To show why structural artifacts matter, we use balanced binary search tree (BST) as an example. To create a balanced BST, we insert 5 nodes in the order of 2, 11, 13, 14, 1, obtaining T_1 (Figure 1(a)). After deleting node 2, we obtain T_2 (Figure 1(a)). However, if we directly create the balanced BST by inserting the 4 nodes in the order of 11, 13, 14, 1, we will obtain T_3 (Figure 1(b)). T_3 and T_2 will be different, due to the past existence of node 2. In other words, although we have deleted node 2, its structural artifacts remain in the data organization. By accessing T_2, the adversary may suspect the past existence of the sensitive data (which now have been deleted), and try to partially or fully recover them. A more concrete attack scenario utilizing these structural artifacts is provided in Sec. 3.

Transaction format: Date, Description, Amount (BTC), Amount ($)

Transaction A:
Tue Jan 14 22:18:25 2014, Received with
1BEzVfsppyhCZczWybjR9f3ZPrAPbNBz7K, 0.006, 3.00

Transaction B:
Wed Jan 15 12:51:34 2014, Received with
1BEzVfsppyhCZczWybjR9f3ZPrAPbNBz7K, 0.10131, 50.00

Transaction C:
Thu Jan 16 01:10:23 2014, Sent to 1Mpq1FwxWUCohqfSczQywLN3btAp46VbnY,
-0.016131, -7.00

Transaction D:
Fri Jan 17 12:16:25 2014, Received with
1BEzVfsppyhCZczWybjR9f3ZPrAPbNBz7K, 0.1, 45.71

Figure 2: Bitcoin transactions stored in the victim's device (obtained from MultiBit, and being anonymized in a few components)

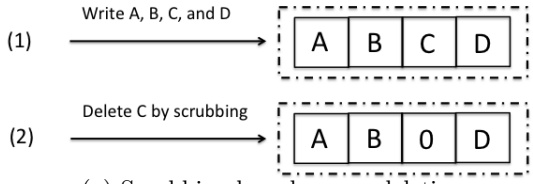
3. ATTACK SCENARIOS

In the following, we provide concrete attack scenarios, in which an adversary is able to recover the data being deleted by performing conventional secure deletion on flash memory. Conventionally, over-writing (e.g., scrubbing [35]) and encryption [35] were used to securely delete data from flash. In the following attack scenarios, we consider a victim who uses his/her computing device equipped with flash memory (NAND flash with 2KB page size) to manage his/her bitcoin transactions. Figure 2 shows a portion of the bitcoin transactions stored in the victim's device (note that these transactions are real bitcoin transactions collected from MultiBit [5], and we anonymized them for privacy concerns). All the four transactions A, B, C, and D in Figure 2 are stored in the device's flash storage, one transaction at each page[1]. As flash media usually prefer log-structured writing technique [7] to reduce over-writes, these transactions are written sequentially to flash pages.

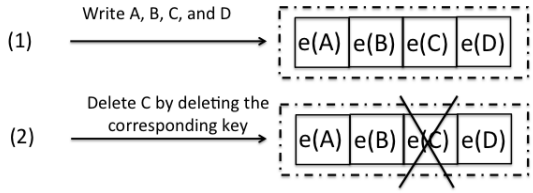
Attack 1: secure deletion based on scrubbing. After having written transaction A, B, C, and D to flash (Figure 3(a)), the victim tried to securely delete C by performing scrubbing[2] over the corresponding flash page. However, scrubbing will convert this page to a page with all "0" bits (i.e., a *zero page*). By having access to the storage state after deletion of C, the adversary will notice there was a deletion on the zero page in the past. Most importantly, he/she will learn sensitive information relating to C: (1) The transaction was generated between "Wed Jan 15 12:51:34 2014" and "Fri Jan 17 12:16:25 2014" as it should be generated after transaction B and before transaction D. (2) The transaction was possibly sending 0.016131BTC to another bitcoin address. This information is obtainable if the adversary is able to learn the overall balance after the four transactions. (3) The transaction was possibly sending bitcoin using address

[1]Note that a bitcoin transaction varies from around 0.2 kilobytes to over 1 kilobyte in size, and is half a kilobyte on average [10]. Therefore, a 2KB flash page should be able to store a bitcoin transaction. As a page is the I/O unit of NAND flash, we consider that each bitcoin transaction is written to one flash page.

[2]Overwriting in flash is usually not feasible except using scrubbing, in which a flash page is converted to a page of all "0" bits by programming all the remaining "1" bits to "0" bits, as flash can allow programming a single "1" bit to"0".



(a) Scrubbing-based secure deletion



(b) Encryption-based secure deletion. e is an encryption function

Figure 3: Attacking conventional secure deletion schemes for flash memory.

$1BEzVfsppyhCZczWybjR9f3ZPrAPbNBz7K$, as this address was used to receive bitcoin in previous transactions, and bitcoin protocol only allows a transaction to send bitcoin from an address which has received bitcoin previously. By further searching blockchain [1] using address $1BEzVfsppyhCZczWybjR9f3ZPrAPbNBz7K$, the adversary can identify a bitcoin transaction which can satisfy both (1) and (2), obtaining transaction C's receiving address $1Mpq1FwxWUCohqfSczQywLN3btAp46VbnY$.

Attack 2: secure deletion based on encryption. After having encrypted and written the encrypted transaction A, B, C, and D to flash (Figure 3(b)), the victim tried to securely delete C by deleting its corresponding decryption key [35]. By having access to the storage state after deletion, the adversary can identify that the page holding transaction C has been deleted, as all the other pages storing transaction A, B, and D can be successfully decrypted to correct plain-texts[3]. Based on the plain-texts of transaction A, B, and D, the adversary may learn sensitive information relating to transaction C, using the attack described in "Attack 1".

4. MODEL AND DEFINITIONS

4.1 System Model

We consider a flash-based block device (Figure 4). The device exposes a *block-based access interface*, by which the file system from upper layer can read/write the device using regular block-device addresses. We define "data node" as the unit of data reading/writing from/to the block device by the upper layer. Let N be the capacity of the block device in terms of data nodes. Let i denote the block-device address, then $0 \leq i \leq N-1$. The access interface at least needs to provide two entry-points:

– Block_Device_Read(i, &data): read a data node from block-device address i

– Block_Device_Write(i, data): write a data node to block-device address i

The delete operation (e.g., TRIM [11] from the upper layer) can simply be implemented by Block_Device_Write(i, NULL). The *NAND flash* within the device consists of n erase blocks, each composed of s pages. The *flash translation layer* translates a block-device address to a flash address. For example, the block-device address i will be translated to flash address (a, b), where a identifies the erase block and b identifies the flash page in this erase block.

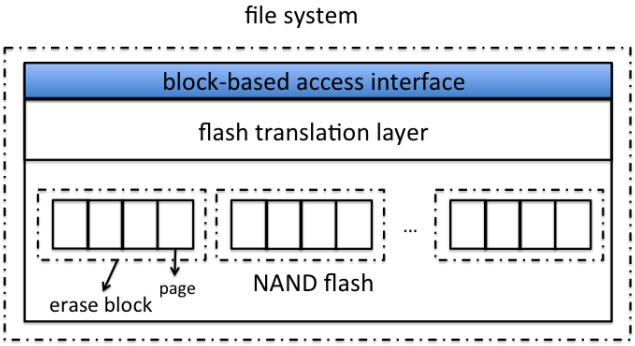


Figure 4: System model

4.2 Adversarial Model

We consider an adversary who tries to learn the sensitive information about the deleted data by having access to the storage state of a flash-based medium. We assume the adversary is computationally bounded, and is able to access the storage state no more than once. This applies to a lot of real-world scenarios, e.g., the attacker steals a smart phone equipped with an eMMC card or a laptop equipped with an SSD drive [15, 41], or breaks into a data center obtaining a snapshot of a target flash-based storage medium. Note that the adversary is not allowed to have access to random bits flipped during the running of the algorithm [25].

4.3 Security Definition

A *truly secure deletion* scheme should achieve two security properties: 1) sanitizing data from the storage medium, such that the adversary is not able to have access to them; 2) sanitizing structural artifacts introduced by the data being deleted, such that the adversary is not able to infer any sensitive information from the storage state about the deleted data. A formal definition of truly secure deletion is as follows.

Let $\mathscr{D}$ be a secure deletion scheme. Let S_1 be the sequence of operations (e.g., insert, delete) performed on the storage medium. $S_1 = \{O_{t_0}, O_{t_1}, ..., O_{t_i}, ..., O_{t_l}\}$, in which O_{t_i} represents the operation at time t_i, where $0 \leq i \leq l$. Let O' be the set of those operations from S_1 that delete data items, and O'' be the set of those operations from S_1 that insert the corresponding data items. Let S_2 be the sequence of operations excluding the operations in O' and O''. The adversary obtains a complete image of the storage state at time t_p, where t_p is after t_l but before a new operation is performed. Let P be the probability that the adversary can differentiate which operation sequence, S_1 or S_2, led to the storage state at time t_p. We say $\mathscr{D}$ is a truly secure deletion scheme if and only if $P \to 0$.

[3]For encryption-based secure deletion [35], decryption keys are usually stored in disks, and to achieve secure deletion, those keys for the deleted data will be removed

Truly secure deletion VS conventional secure deletion. Truly secure deletion requires sanitizing both the data and the structural artifacts introduced by the data being deleted. In other words, once the data have been deleted, the resulting storage state is indistinguishable from that the data being deleted never appeared. This can guarantee that the adversary, by looking at the storage state after a deletion has been performed, will not be able to learn anything about the deleted data (e.g., the exact content, the location, and any other sensitive information relevant to the deleted data). This offers a much stronger security guarantee than conventional secure deletion [23, 24, 35, 39], which can only ensure that the deleted data become inaccessible.

5. TEDFLASH

5.1 Overview

To achieve *truly secure deletion* guarantee for flash-based block devices, we design TedFlash. Our TedFlash is able to sanitize both the obsolete data and the structural artifacts introduced by such data. To sanitize the structural artifacts, we use a random placement table to organize the data stored on flash. In this way, a new data node is always placed to a randomly selected location on flash, and will not affect the placement of any other data, hence no structural artifacts are introduced. Upon deletion, we can simply sanitize the obsolete data, eliminating the need of sanitizing the structural artifacts introduced by them. The random placement scheme well fits the nature of flash memory, as random seeks can be performed efficiently in flash memory. In addition, random placements can distribute writes/erasures evenly among flash, leading to good wear leveling.

As data nodes are placed randomly on flash, we need to keep track of these placements, i.e., the mappings between the block-device addresses and the flash locations. These mappings should be committed to flash in case of system failures like power lost. This unfortunately will create another issue: Updating a single mapping requires performing an over-write on flash, which is expensive as it requires erasing the entire encompassing erase block and writing back the remaining data stored in this block, leading to significant write amplification. We mitigate this by 1) organizing mappings in such a way that can preserve locality from the upper layer; and 2) accumulating multiple subsequent updates on mappings, and performing them together. As locality is preserved in the mappings, the batched updates may target only a few different erase blocks, significantly reducing the number of block erasures required for per update. In addition, to avoid introducing structural artifacts in the mappings, we organize mappings in such a way that writing a new mapping will not affect the locations of any other mappings.

TedFlash securely sanitizes a data node from flash by erasing its encompassing erase block. When the size of a data node is close to that of an erase block, securely deleting a data node will not cause significant write amplification. However, when the data node size is small, erasing the encompassing block will lead to enormous write amplification and significantly decrease throughput. Batching multiple subsequent deletions on data nodes and performing them together cannot help as the random placements in TedFlash have destroyed locality from the upper layer. We further optimize TedFlash by encrypting each data node with a unique key, so that upon a deletion, we can delete the corresponding key, delaying the deletion of the encrypted data node. As keys are much smaller than data nodes, it becomes much easier to organize them in a locality-preserving manner, which can be utilized to reduce the amortized overhead of deleting keys. Note that to avoid introducing structural artifacts in keys, we also organize keys in such a way that writing a new key will not affect the locations of any other keys.

5.2 A Random Placement Table

A random placement table [30] works as follows. Consider an array t of l slots. Each slot is initially empty (e.g., each slot is filled with 0). Elements are inserted into and deleted from t in the following ways:

Insert. To insert an element x, we pick a number i from $\{0, 1, 2, \cdots, l-1\}$ uniformly at random. If $t[i]$ is empty, we store x at $t[i]$. Otherwise, we continue probing slots at random until we find an empty slot which can be used to store x.

Delete. To delete an element stored at $t[i]$ $(0 \leq i \leq l-1)$, we simply re-set $t[i]$ to 0.

As it is proved in Sec. 6.1, deleting an element from the aforementioned random placement table can achieve the *truly secure deletion* guarantee. A significant advantage of this random placement table comes from its efficiency on insert and delete operations, which can be performed in constant time.

5.3 TedFlash Design

5.3.1 How to Sanitize the Structural Artifacts

The *truly secure deletion* guarantee requires sanitizing all the corresponding structural artifacts when certain data have been deleted. To achieve this guarantee in a flash-based block device, we can use a "post-processing" solution: upon deleting a data node, we sanitize all the corresponding structural artifacts, such that the storage state after this deletion is indistinguishable from that the deleted data node never existed. Post-processing may be prohibitively expensive for flash, as it may require relocating a large number of data and incur a lot of over-writes just for a single deletion, significantly decreasing throughout.

We thus turn to a more promising "pre-processing" solution. We use a special data structure to organize the data stored on flash, by which the placement of a new data node will not affect the placements of any other data nodes. A history independent data structure [31] can immediately satisfy this requirement. However, simply applying an existing history independent data structure is impractical for flash as it may either require a large number of data re-locations for a simple write [14, 31] or incur significant over-writes [18].

We use the random placement table introduced in Sec. 5.2 to organize the data in flash, which is shown to be able to achieve the *truly secure deletion* guarantee (Sec. 6.1). The random placement table amazingly fits the nature of flash memory: First, it requires performing random seek, while in flash, the random seek is as efficient as the sequential seek, a salient characteristic of flash memory. Second, placing the writes to random locations can inherently achieve good wear leveling as writes/erasures can be distributed evenly among the entire flash.

By utilizing the random placement table, the data node written to block-device address i is placed to an empty location which is randomly selected from flash. If the block-device address i has been written before, the old data node will become invalid and should be securely sanitized by erasing the corresponding flash block, then the new data node will be placed to a new randomly selected empty location.

5.3.2 Handling Metadata

TedFlash stores each data node from the upper layer to a randomly selected flash location, e.g., the data node written to block-device address i is stored in a random location (a, b). Due to the randomness, we need to keep track of the mappings between the block-device addresses and the flash locations.

As data nodes may be frequently written/deleted by the upper layer, the corresponding mappings may need to be updated frequently. However, updating mappings is expensive since they are usually committed to flash, which is update unfriendly. To efficiently update them, we organize them in a locality-preserving manner. In this way, it is possible to reduce the amortized overhead by batching multiple subsequent update operations and performing them together.

Organization and management of mappings. We design a MAP table to organize the mappings. Our MAP tale always maintains a fixed size of N rows. In the MAP table, the mapping for block-device address i is always located in the i-th row, where $0 \leq i \leq N-1$. Initially, we fill "NULL" as the default flash location for each block-device address. If a data node is written to block-device address i, and stored by TedFlash to a randomly selected flash location (a, b), we will update the mapping in row i by changing the flash location to (a, b). If a data node is deleted from block-device address i, we will update the mapping in row i by changing the flash location to "NULL". If a read is performed on block-device address i, we will read the mapping in row i, obtain the corresponding flash location, and read the corresponding data node. The MAP table is stored on a contiguous area of flash, consisting of multiple contiguous erase blocks. We call this area *metadata area*.

The aforementioned design is advantageous, because: First, it preserves in the metadata area the locality from the upper layer, so that we can reduce the amortized overhead by batching multiple subsequent update operations on mappings. Second, writing a new mapping will not introduce any structural artifacts, as it will not affect the placement of any other data (including mappings and regular data).

Efficiently reading mappings. As the I/O granularity of NAND flash is a page, reading a sole mapping from flash usually requires reading the entire encompassing page, which is expensive. We can improve the performance by caching all or a portion of the MAP table. Note that commodity flash devices are usually equipped with a certain amount of SRAM and DRAM which can be used for the caching purpose. For example, Jasmine OpenSSD Platform [3] has 96KB SRAM and 64MB DRAM; LPC-H3131 [4] has 192KB SRAM and 32MB DRAM.

Efficiently updating mappings. Updating a mapping from metadata area requires performing an over-write on flash, which is expensive and should be avoided. By accumulating multiple subsequent updating operations and performing them together, we can significantly reduce the overhead for per-mapping update. Due to locality, the accumulated updates may only target the mappings from a few different erase blocks, and by waiting, we reduce the number of block erasures necessary per update operation. Note that accumulating multiple update operations in volatile storage (e.g., SRAM or DRAM) is problematic, as they will be lost upon system failures (e.g., power lost). We address this issue by utilizing a few journal blocks. The mapping update operations which have not been committed are stored sequentially to the journal blocks. After the journal blocks are filled, we commit all the updates to the MAP table, and erase the journal blocks. Next time we will pick a new set of journal blocks for wear leveling consideration.

Wear leveling for metadata area. After introducing the MAP table, we face some wear leveling issues: 1) The programs/erasures may not distribute evenly among the erase blocks holding the metadata area. This is because: Writes to block device may concentrate on a few hot places [18]; As the metadata area preserves locality from the upper layer, the writes to the key area will also concentrate on a few hot spots. 2) The erase blocks holding the metadata area have significantly more programs/erasures per block compared to those for data area, because the total number of writes to both areas are approximately the same, but the metadata area is much smaller. A good wear leveling solution can be periodically moving the metadata area around the entire flash, which is feasible as the metadata area is very small. For example, if each mapping is 4-byte in size, and each data node is 4KB in size, the metadata area only occupies 0.1% of the entire flash. For a 10GB eMMC card, that is only 10MB.

5.3.3 Operations of TedFlash

We describe the detailed procedure for the main operations of TedFlash in Figure 5.

Block_Device_Write provides an entry point for the upper layer to write data to block-device addresses. It can also be used as an entry point to delete data from a flash-based block device by using NULL data as input. Given a block-device address i, we first search the MAP table, obtaining the corresponding flash location. If the flash location is not NULL, this is an over-write. Otherwise, it is a new write. For the over-write, we first sanitize the old data from flash. For both the new write and the over-write, we place the new data to a new empty location which is randomly selected. To obtain such a flash location, we may need to perform multiple trials. One optimization can be, we keep track of the empty flash locations in RAM, and randomly pick one when needed. Finally, we update the mapping for block-device address i.

Block_Device_Read provides an entry point for the upper layer to read data from block-device addresses. Given a block-device address i, we first search the MAP table, obtaining the corresponding flash location. We then read the data from flash. A read FAILURE will occur if the flash location does not possess any valid data.

5.4 Optimizing TedFlash

TedFlash is efficient when the size of data node is close to that of the erase block. However, if the data node size is small (e.g., 4KB), deleting a data node will be expensive as it requires erasing the entire flash block which stores this data node, leading to significant write amplification. Batching

Block_Device_Write($i, data$):

1. Read the mapping located in the i-th row of the MAP table, obtaining the flash location (a, b)
2. If $data$ is NULL:
   (1) Delete the data node stored in flash location (a, b)
   (2) Update the mapping located in the i-th row of the MAP table by changing the flash location to NULL;
   (3) Return SUCCESS
3. If (a, b) is not NULL:
   (1) Delete the data node stored in flash location (a, b)
4. Choose a number r uniformly at random from a set $R = \{0, 1, ..., N-1\}$
5. Calculate a new flash location (x, y):
   (1) $x = r/\frac{s \cdot |page|}{|data\ node|}$
   (2) $y = r\%\frac{s \cdot |page|}{|data\ node|}$
6. Check whether the flash location (x, y) is empty or not. If it is not empty, go to step 4
7. Write $data$ to flash location (x, y)
8. Update the mapping located in the i-th row of the MAP table by changing the flash location to (x, y)
9. Return SUCCESS

Block_Device_Read($i, \&data$):

1. Read the mapping located in the i-th row of the MAP table, obtaining the flash location (a, b)
2. If (a, b) is NULL, return FAILURE
3. Otherwise, read the data node from flash location (a, b) to $\&data$
4. Return SUCCESS

Figure 5: TedFlash operations. SUCCESS: 0; FAILURE: -1

multiple subsequent delete operations and performing them together unfortunately cannot help, as TedFlash randomizes the placement of data nodes on flash, which completely disturbs the locality from the upper layer.

One optimization could be to use partial scrubbing [26] to remove the desired data nodes. However, this approach is only suitable for SLC flash, as it may cause errors to MLC flash [26]. Additionally, it only suits applications of data which are meaningful in bytes [26].

Motivated by DNEFS [35], we optimize the performance of delete operations in the following way: We encrypt each data node with a unique key. Upon deleting a data node, we only delete the corresponding key, and delay the deletion of the encrypted data node. This can bring several advantages: 1) As the key size (e.g., 16 bytes) is significantly smaller than the data node size, keys can be stored in a much more compact manner, which makes it possible to optimize the performance of deleting keys. 2) By delaying the deletion of encrypted data nodes, we avoid instantly erasing the corresponding flash block, which is expensive. A flash block will not be erased until it has accumulated enough number of obsolete data nodes, so that the amortized overhead of deletion can be significantly reduced. Compared to the original TedFlash, the optimized TedFlash seems to have reduced the security, as the deleted data, though become inaccessible, are still preserved in the storage state before the space occupied by them are actually reclaimed. By observing the storage state after a delete operation, the adversary is able differentiate the storage state from that the deleted data never existed. However, as it is shown in Sec. 6.1, compared to the original TedFlash, the adversary cannot obtain any more knowledge except some additional randomness.

The question now becomes, how can we efficiently delete keys? As deleting a key usually requires performing an over-write on this key, we can efficiently delete them by adapting the idea from metadata handling: We organize keys in a locality-preserving manner, so that we can batch multiple subsequent key deletions, and perform them together to reduce the amortized overhead.

Organization and management of keys. We organize keys using a key table, which has a fixed size of N rows. The design of the key table is similar to the MAP table, in which the key used for encrypting the data node associated with block-device address i ($0 \leq i \leq N-1$) is always stored in the i-th row of the table. Initially, each row of the key table is filled with a fresh unused key which is cryptographically-appropriate random data. When a new data node is written on block-device address i, we will encrypt it using the key located in the i-th row of the key table. When an old data node is deleted, its corresponding key will be replaced by a new unused key. When a data node is read, the corresponding key will be read in order to decrypt this data node. We store the key table in a contiguous region of flash, which consists of multiple contiguous erase blocks. We call this flash area *key area*. Similar to the metadata area, the key area can also preserve the locality from the upper layer. In addition, the writing of new keys will not introduce structural artifacts as it will not affect the placement of any other data.

Efficiently deleting/reading keys. Deleting a key from key area requires performing an over-write on flash, which is expensive and should be avoided. By batching multiple subsequent key deletions and performing them together, we can significantly reduce the overhead for per-key deletion as the key area preserves locality from the upper layer. We can utilize a few journal blocks for this batching purpose (Sec. 5.3.2). To improve the efficiency of reading keys, we can utilize SRAM or DRAM equipped with the flash device to cache all or a portion of keys (Sec. 5.3.2).

Wear leveling for key area. The key area faces similar wear leveling issues as writes to the key area are much more condensed and the erase blocks for holding the key area will have much more programs/erasures per block. As the key area is small (e.g., if each key is 128-bit in size, and each data node is 4KB in size, the key area only occupies 0.4% of the entire flash), we can simply performing wear leveling by periodically moving the key area around the entire flash.

Reclaiming space occupied by obsolete data nodes. As the optimized TedFlash only deletes keys, the obsolete (encrypted) data nodes remain in flash. To reclaim space, we can periodically perform "garbage collection", erasing those flash blocks that accumulate a certain number (e.g., 50%) of obsolete data nodes.

6. ANALYSIS AND DISCUSSION

6.1 Security Analysis

THEOREM 6.1. *Deleting elements from the random placement table can achieve the truly secure deletion guarantee.*

PROOF. Let $S_1 = \{O_{t_1}, ..., O_{t_i}, ..., O_{t_l}, O_{t_{l+1}}\}$ be the sequence of l insert operations and 1 delete operation performed on the random placement table, in which O_{t_i} represents the operation being performed at time t_i. The operation $O_{t_{l+1}}$ is the delete operation. The adversary can have access to the table at time t_p, where t_p is after t_{l+1}. Let O_{t_j} be the operation of inserting the corresponding element being deleted by operation $O_{t_{l+1}}$, where $1 \leq j \leq l$. Let S_2 be the sequence of operations excluding $O_{t_{l+1}}$ and O_{t_j}. We need to discuss 3 cases: 1) $j = l$; 2) $j = 1$; 3) $1 \leq j < l$.

Case 1: If $j = l$, $S_2 = \{O_{t_1}, ..., O_{t_i}, ..., O_{t_{l-1}}\}$. Note that we initialize all the slots of the random placement table to 0, and a delete operation will reset the corresponding slot to 0. In S_1, the operation O_{t_l} sets a slot to non-zero, while the subsequent operation $O_{t_{l+1}}$ resets this slot to 0. Thus, for the adversary who can only access the table at time t_p, S_1 is equivalent to an operation sequence in which O_{t_l} and $O_{t_{l+1}}$ never appear, which is S_2. In other words, the adversary cannot differentiate which operation sequence, S_1 or S_2, led to the state at time t_p.

Case 2: If $j = 1$, $S_2 = \{O_{t_2}, ..., O_{t_i}, ..., O_{t_l}\}$. Let A be the same insert operation as O_{t_1}. We construct a new operation sequence $S_3 = \{O_{t_2}, ..., O_{t_i}, ..., O_{t_l}, A, O_{t_{l+1}}\}$. Based on our discussion for Case 1, we know that by accessing the state at time t_p, the adversary cannot differentiate which operation sequence, S_3 or S_2, led to the state. The only difference between sequence S_1 and S_3 is when to perform the operation A which inserts an element x into the table: S_1 inserts x at time t_1, while S_3 inserts x at time t', where $t_l < t' < t_{l+1}$. As the random placement table always places the element to a randomly selected location, from the adversary point of view (who can access the state at time t_p), no difference can be observed between inserting x at time t_1 and t'. Thus, S_1 and S_3 is equivalent from the adversary's viewpoint. In other words, the adversary cannot differentiate which operation sequence, S_1 or S_2, led to the state at time t_p.

Case 3: if $1 < j < l$, $S_2 = \{O_{t_1}, ..., O_{t_{j-1}}, O_{t_{j+1}}, ..., O_{t_l}\}$. We separate the operation sequence S_1 into 2 sub-sequences, S'_1 and S''_1, in which S'_1=$\{O_{t_1}, ..., O_{t_{j-1}}\}$ and S''_1=$\{O_{t_j}, ..., O_{t_l}, O_{t_{l+1}}\}$. We also separate the operation sequence S_2 into 2 sub-sequences, S'_2 and S''_2, in which S'_2=$\{O_{t_1}, ..., O_{t_{j-1}}\}$ and S''_2=$\{O_{t_{j+1}}, ..., O_{t_l}\}$. Since $S'_1 = S'_2$, case 3 can be further reduced to a new case: we want to judge whether the adversary can differentiate which operation sequence, S''_1 or S''_2, led to the state at time t_p after removing all the elements created by S'_1 from this state. This new case is exactly what we have discussed in Case 2.

The aforementioned proof can be easily generalized to the case where multiple pairs of insert/delete operations are present.

□

Theorem 6.1 proves that the random placement table can achieve the truly secure deletion guarantee. TedFlash relies on the random placement table to write/delete data nodes to/from flash, and thus is able to achieve the truly secure deletion guarantee. However, to improve the efficiency of updating mappings, TedFlash chooses to batch multiple updates in journal blocks and perform them together later. By accessing the journal blocks, the adversary will learn which data nodes have been deleted recently, leading to leakage of a few most recent delete operations (e.g., if the number of journal blocks being used is 1, and each block has 64 pages, the maximal leakage will be 64 most recent delete operations). This leakage will be eliminated when the journal blocks are filled and erased.

Similar to the original TedFlash, the optimized TedFlash has the leakage of a few most recent delete operations stored in the journal blocks (for the purpose of efficiently updating keys and mappings). In addition, as the optimized TedFlash only deletes keys, the corresponding encrypted data nodes will remain in flash until the corresponding flash block is erased. Thus, the adversary will observe the existence of these encrypted data nodes. However, the adversary cannot obtain more knowledge except some randomness, because: First, the encrypted data nodes cannot be decrypted by the adversary as their corresponding keys have been deleted. Thus, for the adversary, those encrypted data nodes are no more than randomness; Second, the adversary cannot correlate the flash pages storing those encrypted data nodes with the block-device addresses, as the corresponding mappings have been deleted; Third, the adversary cannot learn anything from the locations of those encrypted data nodes, as the entire flash layout has been randomized.

6.2 Discussion

Truly secure deletion and history independence. History independence [31] ensures that by having access to a storage state, the adversary is not able to identify the operation sequence which leads to this state. Thus, given two operation sequences leading to the same storage state: one sequence has a delete operation and its corresponding insert operation, and the other sequence does not have the aforementioned delete and insert operation, the adversary will not be able to differentiate which operation sequence led to this storage state. In other words, history independence guarantees after having removed a data record, all the corresponding structural artifacts will have been removed, achieving truly secure deletion (under an implied assumption that the storage state itself will not leak any information about the deleted data, e.g., there is no correlation between the content remaining in the current state and the deleted data). However, a scheme that achieves the truly secure deletion guarantee is not necessarily able to achieve the history independence guarantee. This is because: 1) History independence ensures that no structural artifacts will be introduced. However, truly secure deletion only ensures that the structural artifacts introduced by the data being deleted will be removed; 2) History independence ensures that an adversary cannot identify the operation order in the past, e.g., the order of insert operations in a voting machine, preserving order privacy. However, truly secure deletion does not provide any guarantees on the past operation order.

Note that when using "pre-processing" approach to achieve the truly secure deletion guarantee, we need to plan ahead and ensure that every new data will not bring in structural artifacts, as we cannot predict which data will be deleted in the future. In this sense, *using "pre-processing" approach seems overkill and can achieve a guarantee stronger than truly secure deletion*, but it is still unclear the relationship between this guarantee and history independence[4].

[4]Although Molnar at al. [30] claimed that the random placement table can achieve history independence, but there is no proof available to justify their claim.

Truly secure deletion and undetectable secure deletion. Jia et al. [26] proposed undetectable secure deletion that can hide the deletion history from the adversary. Undetectable secure deletion is a security notion weaker than the truly secure deletion, because: 1) A scheme that can achieve undetectable secure deletion guarantee is not necessarily able to achieve the truly secure deletion guarantee, as it cannot sanitize the structural artifacts introduced by the deleted data; 2) A scheme that can achieve the truly secure deletion guarantee can always achieve the undetectable secure deletion guarantee, as it can always hide the deletion history.

Handling cache-related issues. To efficiently read mappings/keys, we choose to cache all or a portion of mappings/keys in the SRAM/DRAM. Two issues need to be addressed: 1) The mappings/keys being cached may become inconsistent if the corresponding mappings/keys stored in flash have been updated; 2) If only a portion of mappings/keys is cached, a target mapping/key may not be satisfied from the cache (i.e., a cache miss). To handle the first issue, when a mapping/key is updated, we update it in both the cache and the flash. To handle the second issue, upon a cache miss, we read the target mapping/key from flash, and update the cache using a certain replacement strategy like FIFO and LRU.

The nature of the data residing on flash. TedFlash removes both the obsolete data and the structural artifacts brought by these data, such that by having access to the current storage state, the adversary cannot differentiate it from the storage state that the deleted data never appeared. In this sense, the adversary should not be able to learn anything about the deleted data, because by accessing the storage state after deletion, he/she does not gain any additional advantages compared to having access to the storage state in which the deleted data never appear. Thus, we believe TedFlash should work for both encrypted and non-encrypted file system. For example, considering plain-text "Tim is an HIV patient", after removing sensitive information "HIV", we obtain "Tim is a patient". TedFlash ensures that, by having access to the storage state of "Tim is a patient" after having deleted "HIV", the adversary gains no additional advantages over having access to the storage state of "Tim is a patient" in which "HIV" never appears. Therefore, the adversary should not be able to learn anything about "HIV". And also, we need to emphasize that TedFlash aims to protect the secrecy of the data having been deleted from flash, rather than the data still residing on flash.

System robustness. Corruption of either MAP table or key table may lead to failures of the entire flash. Therefore, protecting the MAP table and the key table is of significant importance to increase the robustness of flash devices equipped with TedFlash. The solution could be to introduce redundancy. In storage domain, multiple coding techniques like replication [20], erasure coding [16] and network coding [17] can be utilized to add redundancy. Considering both the MAP table and the key table are small in size as well as their dynamic nature, we can simply use replication and create duplicate copies for both of them. This will increase the overhead for committing data to flash, as we need to update the duplicate copies of both tables. One remediation could be to synchronize the updates to the duplicate copies after a certain number of writes. This can reduce the additional overhead for robustness, but the most recent updates may be lost upon failures of these tables.

7. IMPLEMENTATION AND EVALUATION

7.1 Implementation

We implemented TedFlash based on OpenNFM [7], an open source NAND flash controller framework. For comparison, we also implemented DNEFS [35] and HiFlash [18] using OpenNFM. DNEFS encrypts each data node with a different key and collocates keys in a key storage area on the flash. It can sanitize the data by efficiently deleting keys. However, it cannot achieve truly secure deletion as the structural artifacts remain in the flash layout. HiFlash relies on a one-one mapping technique to achieve history independence, hence truly secure deletion guarantee (Sec. 6.2). However, the one-one mapping technique cannot satisfy design constraint C1 or C2, and is thus impractical for flash.

OpenNFM uses an architecture consisting of three layers: FTL, UBI, and MTD. FTL mainly handles mappings between block device and flash, so that the flash device can provide a uniform block device interface to the external computing components like file systems. UBI mainly takes care of wear leveling and bad block management. MTD provides a raw flash abstraction, handling the physical characteristics of different flash chips. To implemented DNEFS, HiFlash, and TedFlash, we modified OpenNFM as follows:

- DNEFS: We added encryption/decryption as well as key management to FTL.
- HiFlash: We modified FTL to support one-one mapping, and modified UBI to support the special wear leveling required by HiFlash [18].
- TedFlash: We modified FTL to support random placement. We also incorporated encryption/decryption as well as key management described in the optimized TedFlash. We introduced journal blocks to efficient update the MAP table and the key table.

We ported the modified OpenNFM to LPC-H3131 [4], a development board equipped with 180 MHz ARM microcontroller, 512MB NAND flash, and 32 MB SDRAM. The NAND flash has 128KB block size and 2KB page size, thus the entire NAND flash has approximately $4,000$ erase blocks, and each block is composed of 64 pages. For TedFlash, the MAP table is less than 1MB in size, as it contains $4,000 * 64$ mappings, each of which can be represented by 18 bits.

We benchmarked the original OpenNFM, DNEFS, HiFlash, and TedFlash using fio [2]. When running fio, we used the non-buffered I/O option. The fio is run in a host computer with 8 Intel i7 CPUs at 3.40GHz, 4GB RAM, and Windows 7 Pro 32-bit.

7.2 Evaluation

7.2.1 Throughput

The impact of encryption on throughput. The development board (LPC-H3131) we used is equipped with a low-end microcontroller (180 MHz) without hardware encryption module, and software-based encryption runs extremely slow on it. Therefore, the experimental results including the overhead of encryption/decryption will be biased for this board, as encryption/decryption will dominate

	READ (MB/s)	WRITE (MB/s)
without encryption	520	470
with encryption	460	460
throughput decrease	11.5%	2.1%

Table 2: The impact of encryption on read/write throughout for a board equipped with encryption hardware module. The data were obtained for AES-128 from SAGE S881 [8], a board with 1 MB SRAM and a built-in encryption & decryption module

# of journal blocks	SW	RW
(1, 1)	225	211
(2, 2)	596	534
(3, 3)	668	650
(0, 0)	680	658

Table 3: The write throughput (KB/s) of TedFlash when varying the number of journal blocks. (x, y) means x journal blocks for mappings and y journal blocks for keys. $(0, 0)$ means no updates are performed on the metadata/key area. SW - sequential write, RW - random write

the entire overhead. We observed that a large number of flash controller chips in the new-generation smartphones and SSDs have been equipped with hardware encryption module. For example, most of the SSDs manufactured by SAMSUNG [9] support AES hardware encryption. With encryption hardware enabled, the impact of encryption on throughput should be very small. Our experimental evaluation in Table 2 confirmed the aforementioned statement. Therefore in the following, we excluded the overhead resulted from encryption/decryption for all the schemes.

The impact of journal blocks on throughput. To efficiently update mappings and keys, we use a few journal blocks to accumulate multiple updates and perform them together on the metadata/key area. Table 3 shows the throughput of TedFlash when the number of journal blocks varies. Note that each block has 64 pages, and each page can be used to store 1 update operation[5], therefore, a journal block can at most accumulate 64 update operations.

We observed that when the number of journal blocks increases, the write throughput increases. This is because, using more journal blocks is able to accumulate more updates before performing them together, leading to a higher probability that more updates will belong to the same erase blocks, hence reducing the overall number of erasures. However, as the operations in the journal blocks will be leaked, using more journal blocks will have more leakages (Sec. 6.1).

Throughput comparisons. We compared the read/write throughput among DNEFS, HiFlash, and TedFlash. Benchmarking results are shown in Table 4. To efficiently read mappings, we cached the MAP table in the board's DRAM for all the schemes. To efficiently read keys, we cached the keys in the DRAM for both DNEFS and TedFlash. In addition, we used 3 journal blocks for key updating and 3 journal blocks for mapping updating, respectively. We have several observations:

[5]The I/O granularity of NAND flash is a page, thus we can only use 1 page to store 1 update

Scheme	SR	RR	SW	RW
OpenNFM	1,831	1,524	1,155	968
DNEFS	1,259	1,218	766	672
TedFlash	1,125	1,099	668	650
HiFlash	1,099	1,065	277	87

Table 4: Comparisons of read/write throughput (KB/s) among TedFlash and other secure deletion schemes. SR - sequential read, RR - random read

(a) The read/write throughput of both DNEFS and TedFlash is decreased compared to the original OpenNFM. This is mainly due to the additional overhead in key management, including both key reading and updating.

(b) The read/write throughput of TedFlash is slightly reduced compared to DNEFS (approximately 10%). This is because, random seek is not significantly but slightly slow than sequential seek. This justifies that TedFlash can achieve a much stronger secure deletion guarantee with a small additional overhead.

(c) The write (especially random write) throughput of HiFlash is significantly reduced compared to TedFlash. This is due to the significant write amplification caused by the large number of over-writes resulted from the one-one mapping technique. This justifies that by relaxing the security offered by HiFlash, TedFlash can achieve a significantly better performance.

7.2.2 *Wear Leveling*

To evaluate the wear leveling effectiveness of TedFlash, we used the Hoover economic wealth inequality indicator [35]. This metric is originally used to quantify the unfairness of wealth distributions. It corresponds to an appropriately normalized sum of the difference of each measurement to the mean. In terms of flash memory, it indicates the fraction of erasures that must be re-assigned to other erase blocks in order to obtain completely even wear. Assuming the erasure counts of all the erase blocks are e_1, e_2,..., e_n, and $E = \sum_{i=1}^{n} e_n$, then the wear levering inequality can be computed as: $\frac{1}{2}\sum_{i=1}^{n} \| \frac{e_i}{E} - \frac{1}{n} \|$.

We repeatedly wrote data to the board, filled the 512MB flash storage, and then erased the data. After having written 500GB data, we calculated the number of erasures performed on each flash block, and computed the wear levering inequality, obtaining 6×10^{-4}. This small value indicates a good wear leveling effectiveness, and justifies that random placements indeed can achieve good wear leveling, as it distributes data nodes among the entire flash in a uniformly random manner so that each flash location has an equal opportunity to be programmed/erased.

8. RELATED WORK

Encryption-based secure deletion. Lee et al. [28, 29] proposed a secure deletion scheme for YAFFS. They forced the current and the previous keys of a file to be stored in the same flash block, so that a file can be deleted by a single block erase. Lee et al. [27] further extended this solution with standard data sanitization operations on the key containing blocks. Reardon et al. [35] introduced DNEFS, in which they encrypted each data node with a unique key and collocated keys in a key storage area on the flash.

Overwriting-based secure deletion. Sun et al. [38] used zero overwriting and block cleaning to securely delete data from flash. Reardon et al. [36] introduced purging and ballooning at the user-level, and zero overwriting at the kernel level for secure deletion in YAFFS. Wei et al. [35] proposed to use scrubbing to efficiently sanitize data from flash pages without performing block erasures. In flash, programming "0" bit to "1" bit is not possible except performing a block erasure. However, programming "1" bit to "0" bit is feasible. Based on this observation, they sanitized data from a page by programming all the remaining "1" bits to "0" bits. TrueErase [21] was a framework which relied on block erasure to delete data and metadata upon user request. DEFY [34] performed all-or-nothing transform on the data, creating a small message expansion for the data. In this way, the data can be securely and efficiently removed by only removing this small expansion.

The aforementioned secure deletion schemes can sanitize data from flash, but none of them can sanitize structural artifacts. NFPS [26] aimed to conceal the past existence of the deleted data in flash memory. However, it still cannot sanitize the structural artifacts introduced by the deleted data. HiFlash [18] can achieve history independence, and is thus able to sanitize the structural artifacts, achieving truly secure deletion (Sec. 6.2). To achieve history independence, HiFlash always places the data written to the same block-device address to the same flash location, so that the placements of any data are independent and the resulting storage layout is canonical and not impacted by the "history". Such a "one-one mapping" mechanism unfortunately will cause significant over-writes to a few hot locations, leading to significant overhead as well as write unevenness. This may be the unavoidable cost for achieving such a strong security guarantee. In some sense, the design rational of truly secure deletion is to achieve a good trade-off between security and performance by slightly reducing the desired security guarantee.

9. CONCLUSION

In this paper, we propose TedFlash, a truly secure deletion scheme for flash-based block devices. Truly secure deletion is a security notion that is much stronger than conventional secure deletion, as it can sanitize both the obsolete data as well as the corresponding structural artifacts, while the conventional secure deletion can only sanitize the obsolete data. TedFlash places data nodes to flash using a random placement table which can ensure that placing a new data node will not introduce structural artifacts. This random placement technique well fits the nature of NAND flash in terms of random seek and wear leveling. Security analysis and experimental evaluation show that TedFlash can achieve the truly secure deletion guarantee with a small additional overhead compared to the conventional secure deletion.

Acknowledgment

This work was supported by ARO W911NF-15-1-0576. Bo Chen would also like to thank the support from Center for Information Assurance at the University of Memphis. Shijie Jia and Luning Xia were supported by National 973 Program of China under award No. 2014CB340603. Peng Liu was supported by NSF CNS-1422594, NSF CNS-1505664, and ARO W911NF-13-1-0421 (MURI). The authors would like to thank anonymous ACSAC reviewers for their insightful suggestions and advice. We would also like to thank Radu Sion for his contribution in the early stages of the work.

10. REFERENCES

[1] Blockchain. `https://blockchain.info/`.
[2] fio, `http://freecode.com/projects/fio`.
[3] Jasmine openssd platform, `http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform`.
[4] Lpc-h3131, `https://www.olimex.com/Products/ARM/NXP/LPC-H3131/`.
[5] Multibit. `https://multibit.org/`.
[6] New ssd-backed elastic block storage. `https://aws.amazon.com/blogs/aws/new-ssd-backed-elastic-block-storage/`.
[7] Opennfm, `https://code.google.com/p/opennfm/`.
[8] Sage s881. `http://www.sage-micro.com/chip-7.html`.
[9] Samsung ssd. `http://www.samsung.com/cn/consumer/memory/ssd`.
[10] Scalability - bitcoin wiki. `https://en.bitcoin.it/wiki/Scalability`.
[11] Trim, `http://en.wikipedia.org/wiki/Trim_%28computing%29`.
[12] Uk data protection act 1998 (dpa). http://www.legislation.gov.uk/ukpga/1998/29.
[13] 106th United States Congress. Gramm-Leach-Bailey Act. `http://www.gpo.gov/fdsys/pkg/PLAW-106publ102/pdf/PLAW-106publ102.pdf`, 1999.
[14] Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, FOCS '07, pages 272–282. IEEE Computer Society, 2007.
[15] Bing Chang, Zhan Wang, Bo Chen, and Fengwei Zhang. Mobipluto: File system friendly deniable storage for mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 381–390. ACM, 2015.
[16] Bo Chen, Anil Kumar Ammula, and Reza Curtmola. Towards server-side repair for erasure coding-based distributed storage systems. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 281–288. ACM, 2015.
[17] Bo Chen, Reza Curtmola, Giuseppe Ateniese, and Randal Burns. Remote data checking for network coding-based distributed storage systems. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 31–42. ACM, 2010.
[18] Bo Chen and Radu Sion. Hiflash: A history independent flash device. *arXiv preprint arXiv:1511.05180*, 2015.
[19] United States Congress. Health Insurance Portability and Accountability Act. `http://www.hhs.gov/ocr/privacy/index.html`, 1996.
[20] Reza Curtmola, Osama Khan, Randal Burns, and Giuseppe Ateniese. Mr-pdp: Multiple-replica provable data possession. In *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*, pages 411–420. IEEE, 2008.

[21] Sarah Diesburg, Christopher Meyers, Mark Stanovich, Michael Mitchell, Justin Marshall, Julia Gould, An-I Andy Wang, and Geoff Kuenning. Trueerase: Per-file secure deletion for the storage data path. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 439–448. ACM, 2012.

[22] Simson L Garfinkel and Abhi Shelat. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security & Privacy*, (1):17–27, 2003.

[23] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. Vanish: Increasing data privacy with self-destructing data. In *USENIX Security Symposium*, pages 299–316, 2009.

[24] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, volume 14, 1996.

[25] Jason D Hartline, Edwin S Hong, Alexander E Mohr, William R Pentney, and Emily C Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[26] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Nfps: Adding undetectable secure deletion to flash translation layer. In *Proceedings of The 11th ACM Asia Conference on Computer and Communications Security (ASIACCS '16)*. ACM, 2016.

[27] Byunghee Lee, Kyungho Son, Dongho Won, and Seungjoo Kim. Secure data deletion for usb flash memory. *J. Inf. Sci. Eng.*, 27(3):933–952, 2011.

[28] Jaeheung Lee, Junyoung Heo, Yookun Cho, Jiman Hong, and Sung Y. Shin. Secure deletion for nand flash file system. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1710–1714, New York, NY, USA, 2008. ACM.

[29] Jaeheung Lee, Sangho Yi, Junyoung Heo, Hyungbae Park, Sung Y Shin, and Yookun Cho. An efficient secure deletion scheme for flash file systems. *J. Inf. Sci. Eng.*, 26(1):27–38, 2010.

[30] David Molnar, Tadayoshi Kohno, Naveen Sastry, and David Wagner. Tamper-evident, history-independent, subliminal-free data structures on prom storage-or-how to store ballots on a voting machine. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[31] Moni Naor and Vanessa Teague. Anti-persistence: History independent data structures. In *In Proceedings of ACM symposium on Theory of computing*, pages 492–501. ACM Press, 2001.

[32] Radia Perlman. The ephemerizer: Making data disappear. In *Journal of Information System Security*. Citeseer, 2005.

[33] Radia Perlman. File system design with assured delete. In *Security in Storage Workshop, 2005. SISW'05. Third IEEE International*, pages 6–pp. IEEE, 2005.

[34] Timothy M Peters, Mark A Gondree, and Zachary NJ Peterson. Defy: A deniable, encrypted file system for log-structured storage. 2015.

[35] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 17–17. USENIX Association, 2012.

[36] Joel Reardon, Claudio Marforio, Srdjan Capkun, and David Basin. Secure deletion on log-structured file systems. *arXiv preprint arXiv:1106.0917*, 2011.

[37] U.S. Senator Paul Sarbanes and U.S. Representative Michael G. Oxley. Sarbanes-Oxley Act. `http://www.sec.gov/about/laws.shtml#sox2002`, 2002.

[38] Kyoungmoon Sun, Jongmoo Choi, Donghee Lee, and Sam H Noh. Models and design of an adaptive hybrid scheme for secure deletion of data in consumer electronics. *Consumer Electronics, IEEE Transactions on*, 54(1):100–104, 2008.

[39] Yang Tang, Patrick PC Lee, John Lui, and Radia Perlman. Secure overlay cloud storage with access control and assured deletion. *Dependable and Secure Computing, IEEE Transactions on*, 9(6):903–916, 2012.

[40] Michael Yung Chung Wei, Laura M Grupp, Frederick E Spada, and Steven Swanson. Reliably erasing data from flash-based solid state drives. In *FAST*, volume 11, pages 8–8, 2011.

[41] Xingjie Yu, Bo Chen, Zhan Wang, Bing Chang, Wen Tao Zhu, and Jiwu Jing. Mobihydra: Pragmatic and multi-level plausibly deniable encryption storage for mobile devices. In *Information Security*, pages 555–567. Springer, 2014.

[42] Apostolis Zarras, Katharina Kohls, Markus Dürmuth, and Christina Pöpper. Neuralyzer: Flexible expiration times for the revocation of online data. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy*, pages 14–25. ACM, 2016.