Duplicates also Matter! Towards Secure Deletion on Flash-based Storage Media by Removing Duplicates

Niusen Chen, Bo Chen
{niusenc,bchen}@mtu.edu
Department of Computer Science, Michigan Technological University
Houghton, MI, USA

ABSTRACT

Flash memory has been used extensively as external storage of smartphones, tablets, IoT devices, laptops, etc. Therefore, more and more sensitive or even mission critical data are stored in flash and, once the data turn obsolete, securely deleting them is necessary for both regulation compliance and privacy protection. Traditional secure deletion on flash memory mainly focuses on sanitizing data. However, unique nature of flash memory may cause various data "remnants" and, even though the data are removed, the remnants may be utilized by the adversary to recover the deleted data, compromising the secure deletion guarantee.

Based on both theoretic analysis and experiments using real-world workloads, we have identified one common type of remnants in the flash memory, namely duplicates, which are caused by unique internal functions of flash storage media including garbage collection, wear leveling, bad block management. We propose RedFlash, a novel secure deletion scheme which can efficiently *Re*move both the data and the corresponding *d*uplicates towards secure deletion on *Flash* memory. Security analysis and experimental evaluation show that RedFlash can ensure the secure deletion guarantee, at the cost of a small performance degradation, compared to a regular (non-secure) flash controller.

CCS CONCEPTS

• **Information systems** → **Flash memory**; • **Security and privacy** → **Data anonymization and sanitization**.

KEYWORDS

secure deletion; flash memory; duplicates; FTL; snapshot adversary

ACM Reference Format:
Niusen Chen, Bo Chen. 2022. Duplicates also Matter! Towards Secure Deletion on Flash-based Storage Media by Removing Duplicates. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30-June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3488932.3523255

1 INTRODUCTION

Retention regulations like the EU General Data Protection Regulation (GDPR) [7], the DoD 5220.22-M [6], the Health Insurance Portability and Accountability Act (HIPAA) [29], mandate consistent procedures for data protection. All of them also implement policies and procedures to address a significant data protection issue, namely, how to securely delete electronic data once they turn invalid. This is of paramount importance for both individuals and organizations. For individuals, the recovery of the deleted data may endanger their privacy or may bring life threats, e.g., John thought he had removed his nude photos from his smartphone, but later, a hacker successfully restores the nude pictures and threatens to post them in a public website. For organizations, the recovery of the deleted data may ruin their reputation. For example, a sensitive CIA (Central Intelligence Agency) document was published by New York Times as a PDF file that contained the original document with sensitive information covered up by an overlay; Internet users then removed the overlay and revealed sensitive information about the role of CIA in the 1953 overthrow of the Iranian Government [42].

Secure deletion usually requires "rendering target data recovery infeasible using state of the art laboratory techniques" [51] (**secure deletion guarantee**). Achieving such a guarantee in modern computing devices, however, is a challenging task. A major reason is, a modern computing device is usually equipped with a complicate storage system which consists of a few different layers, e.g., application layer, file system layer, storage medium layer; and therefore, performing secure deletion at the upper layers is usually not able to eliminate the data, since data leakage may be observed at the lower layers. For example, given a Microsoft Word document, deleting data from the document itself cannot guarantee that the data really become inaccessible. Upon receiving a delete request issued by Microsoft Word (at the application layer), the system may simply modify the metadata at the file system layer (e.g., changing the block allocation table and invalidating the data being deleted) to make the data appear to have been removed. However, the actual content is still preserved at the storage medium layer and may be extracted by the adversary through disk forensics. Therefore, secure deletion should be performed at the lower layers (e.g., storage medium layer) in order to achieve the secure deletion guarantee.

Hard disk drives (HDDs) and flash memory are two mainstream storage media for computing devices nowadays, and people are increasingly turning to flash memory for external storage due to its high speed, high reliability and low power consumption. Compared to HDDs, flash memory removes any moving mechanical parts and exhibits a completely different physical nature, e.g., erase-before-write design, out-of-place update, only allowing a finite number of program-erase (P/E) cycles, suffering from read disturb errors, etc. Therefore, a flash storage medium usually introduces a flash translation layer (FTL) to transparently handle its unique nature, exposing a block access interface externally. This type of flash-based

ASIA CCS '22, May 30-June 3, 2022, Nagasaki, Japan

ACM ISBN 978-1-4503-9140-5/22/05.
https://doi.org/10.1145/3488932.3523255

block devices can be found broadly in real world, e.g., solid state drives (SSD), universal flash storage (UFS) cards, MultiMediaCard (MMC), secure digital (SD) cards, etc. Secure deletion techniques have been designed for computing devices using HDDs [13, 36, 48, 67, 69] which however, *cannot work correctly for computing devices using flash storage as they suffer from new attacks unique for flash memory*. An immediate attack is that they usually rely on overwriting target data (or cryptographic keys) at the block layer to ensure secure deletion; however, due to the use of out-of-place update in flash storage media [85], data overwritten at the block layer will be temporarily preserved in the flash memory, and extracted by the adversary later to compromise the secure deletion guarantee. Later secure deletion techniques [24, 25, 32, 33, 43, 46, 53, 54, 72, 84, 85] incorporate the secure deletion strategy into the flash storage medium so that they can directly "destroy" the flash memory data, but still, they cannot achieve the secure deletion guarantee because: *they only target data to be deleted, but are unaware of the fact that, unique functions implemented inside a flash storage medium may produce various data remnants across the entire flash and, without carefully handling such remnants, the adversary may utilize them to partially or completely recover the deleted data.*

In this work, for the first time, we have successfully identified a common type of data remnants, namely duplicates, produced internally by unique functions of flash storage. We have experimentally confirmed the existence of duplicates by running real-world workloads. In addition, leveraging the duplicates, we have successfully attacked the traditional secure deletion schemes. Finally, we have designed RedFlash, the first secure deletion scheme for flash memory which can efficiently remove both the data and the corresponding duplicates towards secure deletion.

There are a few key insights in the design of RedFlash. First, we efficiently locate the duplicates associated with the deleted data without relying on RAM (i.e. *no RAM*) and brute-force searching over the external storage (i.e. *no searching*). This is based on our observations that RAM is vulnerable to power-loss and searching the external storage is expensive. Our key idea is to chain each data node and the pages storing its duplicates together in the flash memory and, once a data node is deleted, we can efficiently locate all those pages storing its duplicates by traversing the chain and delete them, without relying on RAM and searching over external storage. Each page in the duplicate chain will maintain a "dup field", which stores its association to other pages in the chain. A few additional ideas are: 1) Having observed that each out-of-band (OOB) area (i.e., a special area associated with each flash page) is usually under utilized, we choose to store the "dup field" in each OOB area, eliminating the need of using additional space from a flash page. 2) The duplicate chain could be broken if some of the pages in the chain are reclaimed by the flash controller earlier. We thus make the chain more robust by associating a newly added page with all the existing pages in the chain. In addition, we embed a unique chain ID to each page in the chain, preventing deleting a page which has been reclaimed earlier and used to store other data. The uniqueness of the chain ID is ensured by taking advantage of the block device address storing the corresponding data node.

Second, we efficiently remove data as well as their corresponding duplicates by adapting scrubbing [85]. Using conventional block erasure may lead to significant write amplification when the data being removed are small in size. Using scrubbing can efficiently remove data of small size with $O(1)$ complexity, but it is only good for SLC (single-level cell) flash, rather than MLC (multi-level cell) flash due to bit corruptions. Having observed that performing scrubbing on a flash page typically corrupts bits stored in its paired page of the same flash block, we enable efficient deletion of data from the MLC flash, by first relocating the data stored in the paired page, and then performing scrubbing on both the page and the paired page.

Contributions. Our major contributions in RedFlash are:
– We have theoretically analyzed why duplicates exist due to the unique working principle of the internal functions of flash storage media. We have also experimentally confirmed the existence of duplicates by running real-world workloads.
– We have designed RedFlash to securely and efficiently delete both the data and their corresponding duplicates from NAND flash.
– We have analyzed the security of RedFlash. We have implemented RedFlash into an open-source NAND flash controller framework, and performed experimental evaluation (for SLC flash) and simulation (for MLC flash) to assess performance of RedFlash.

2 BACKGROUND KNOWLEDGE ON FLASH MEMORY AND SECURE DELETION

2.1 Flash Memory

Flash memory especially NAND flash has dominated the mass storage of mobile computing devices today. NAND flash consists of memory cells, which are grouped into pages (typically 512B, 2KB, or 4KB in size), and pages are further grouped into blocks (typically 16KB, 128KB, 256KB, or 512KB in size). Each page has a small out-of-band (*OOB*) area [39], typically 64 bytes for a 2KB page, and 224 bytes for a 4KB page [4]. The OOB can be used to store additional information like error correction code, a flag indicating that an encompassing block is bad, etc. Depending on the number of bits each memory cell can store, NAND flash includes single-level cell (**SLC**, in which each cell stores 1 bit), and multi-level cell (**MLC**, in which each cell stores more than 1 bit, e.g., 2 bits).

Compared to traditional mechanical drives, NAND flash exhibits some special characteristics: 1) The unit of a read/program operation is a page, but the unit of an erase operation is a block. Usually, the read operation (around 20 μs) is much faster than the program operation (around 200 μs), which is much faster than the erase operation (around 1.5 ms). Also, reading a cell may cause its nearby cells in the same block to change over time, causing read disturb errors. 2) A flash page cannot be re-programmed before it is erased. Therefore, to overwrite a flash page requires first erasing the entire encompassing block, which requires copying out valid data in this block and writing them back after the block is erased, causing significant write amplification. To mitigate this issue, flash memory usually uses an out-of-place [38, 85] instead of in-place update strategy. 3) Each flash block has a limited number of program-erase (P/E) cycles (e.g., 10K - 100K), and if a flash block is programmed/erased more than a certain threshold, the flash block will turn "bad".
Flash file system vs. flash translation layer. Traditional file systems built for HDDs cannot be directly applied to flash memory due to its unique nature. Instead, a flash-specific file system (e.g.,

YAFFS [87], JFFS [2], and F2FS [8]) needs to be used. The flash file system however, requires direct access to the raw NAND flash, which is rarely supported in computing devices now, and hence flash file systems are used rarely today. The most popular alternative of using flash memory is emulating it as a block device via a piece of special firmware called flash translation layer (*FTL*), such that traditional block-based file systems (e.g., EXT, FAT, NTFS) can be directly used. The FTL stays between the file system and the raw NAND flash, implementing a few key functions including garbage collection, wear leveling, bad block management, and read disturb management.

Garbage collection. Due to the out-of-place update feature of flash memory, an overwrite operation from the upper layer will invalidate flash pages storing the obsolete data. Those flash pages will be reclaimed by garbage collection. A typical procedure of garbage collection [78] is: 1) selecting one block which has the largest number of invalid pages as a victim block; and 2) copying valid data in the victim block to a free block; and 3) erasing the victim block.

Wear leveling. A flash block can be programmed/erased for a limited number of times, and once the limit is reached, the block will be worn out. To prolong overall service life of flash memory, wear leveling (static or dynamic) [88] will be used to distribute programmings/erasures (P/Es) evenly across the flash.

Bad block management. A flash block may turn "bad" over time and cannot reliably store data. Bad block management typically introduces a bad block table to keep track of bad blocks [76]. Once a block turns bad, it will be added to the bad block table and will no longer be used; additionally, valid data stored in it will be copied to good blocks.

The FTL may also implement *read disturb management* [40]. Typical techniques for read disturb management are: 1) The FTL counts the total number of reads on each flash block since last erase and, once the count of a flash block exceeds a threshold, the data stored on it will be copied to a new block which is rarely read. 2) The FTL actively detects frequently read pages, and proactively moves them to rarely read blocks [40].

2.2 Secure Deletion on Flash Memory

The overwriting-based approaches. To securely delete data from flash memory, we can use block erasure [24, 25, 32] or scrubbing [43, 46, 84, 85]. Flash memory can be erased in unit of blocks; therefore, to delete a portion of data from a flash block, we can store valid data in this block elsewhere, and perform a *block erasure*. The block erasure is expensive when deleting small data. Instead, *scrubbing* programs all the remaining "1" bits on a page to "0", such that the entire page is turned to "0"s and the stored data are removed, which is much more efficient when deleting small data.

The encryption-based approaches. The data stored in flash pages are encrypted using unique keys and, secure deletion of data can be achieved by purging the corresponding keys stored in a condensed key storage area of flash [24, 53, 54, 72]. Simply destroying the cryptographic keys while preserving the encrypted data may not ensure secure deletion, because: First, the cryptographic keys may have been present in various sources including processor caches or memory when data were used, and may be extracted after secure deletion [18, 31, 41, 60, 64]. Second, most existing encryption algorithms provide conditional security and may be broken over time [16, 17, 30], e.g., by quantum computers; in addition, keys may be leaked over time [55, 57], e.g., being coerced [47] or stolen; therefore, the adversary can preserve the encrypted data claimed to have been securely deleted via encryption keys purging, and may be able to decrypt them successfully in the future [31, 56].

3 COMPROMISING EXISTING SECURE DELETION SCHEMES USING DUPLICATES

In this section, we first justify the existence of duplicates across the entire flash, both theoretically and experimentally. Leveraging the duplicates, we can compromise the existing secure deletion schemes for flash memory.

3.1 Theoretically Analyzing The Existence of Duplicates in Flash Storage Media

A flash storage medium is managed by the flash translation layer (FTL) and, the FTL implementations are very diverse in different flash storage media. Our methodology is to analyze typical functions implemented in main-stream FTLs as well as typical algorithms used to implement each function. In this way, our analysis can capture a majority of the existing FTL implementations in the wild. In the following, we analyze each typical function and show why duplicates are produced.

Duplicates created by garbage collection. During garbage collection, the flash controller will choose a victim block with the largest number of invalid pages. Then, it will select a block from a pool of free blocks, and copy the data in the valid pages of the victim block to the free block. However, the flash controller typically[1] will not immediately erase the victim block, since an erase operation is time-consuming (Sec. 2.1). Instead, the victim block will be added to the free block pool [26, 49], and will be erased only when the system is idle [1, 34, 45]. Thus, duplicates are preserved in victim blocks.

Duplicates created by wear leveling. We mainly focus on static wear leveling (Sec. 2.1) which is used more broadly in real-world flash storage media than dynamic wear leveling. Upon wear leveling, the flash controller chooses a block (identified as M) which has the smallest erase count from blocks being used, and chooses another block (identified as N) which has the largest erase count from the free block pool. Then, the controller swaps block M and free block N by copying all the data from M to N. Last, the controller adds M to the free block pool, which will be erased during the idle time. The rationale for wear leveling is, the data stored in M is cold data, and therefore should be swapped to the block which has the largest erase count; additionally, those blocks with large erase count will be moved out from the free block pool, so that they will not be used when allocating new free blocks. This wear leveling implementation is advantageous, since it swaps a block being used with a free block for wear leveling, and data can be simply copied to the free block, without involving any immediate erasures. In

[1]Note that there may be an extreme case when the free blocks are rare in the flash memory and the I/O stream is continuous; in this case, the garbage collection may need to erase the victim blocks immediately to facilitate future I/Os. However, this extreme case only happens when a mobile device has reached its storage capacity. This is a low-probability event during the mobile device's lifetime and, in most of the device's lifetime, the garbage collection will not erase the victim blocks immediately.

ID	Content	Number of I/Os
1	*Financial1*, *Financial2*, *WebSearch1*	10,089,261
2	*Financial1*, *Financial2*, *WebSearch2*	13,613,622
3	*Financial1*, *Financial2*, *WebSearch3*	13,295,518

Table 1: Trace sets

Parameter	Value
Storage capacity	128 GB
Minimum free blocks percentage	15%
SSD_LIFE_TIME_THRESHOLD	0.80
Cleaning policy	wear-aware cleaning

Table 2: DiskSim configurations in our experiments

this wear leveling, duplicates are preserved in free block M, which will be erased during the idle time for performance optimization. Another common implementation is that, the controller swaps two blocks in use, and one block has the largest erase count, while the other has the smallest erase count. To accommodate block swapping, it needs to request a free block from the free block pool, and involves 2 block erasures[2]. In this wear leveling, the duplicates are preserved in the free block. In general, different static wear leveling strategies mainly concern on what blocks to be swapped [65] and when to swap the blocks [58]. Although different strategies may be used in wear leveling, the final step is always to swap the blocks, which typically produces duplicates as immediately erasing the obsolete blocks is inferior.

Duplicates created by bad block management. If a block turns "bad", the flash controller usually chooses an empty block from the free block pool, copies valid data from the bad block to this empty block, and adds the bad block to the bad block table. However, data originally stored in the bad block usually will not be sanitized (note that a regular flash storage medium does not handle secure deletion), and hence duplicates are preserved in the bad blocks.

3.2 Experimentally Confirming The Existence of Duplicates

We have experimentally confirmed the existence of duplicates in the flash memory using real-world workloads and a flash storage simulator.

For real-world workloads, we used the storage traces [14] from the UMass Trace Repository, which contain I/O traces from OLTP applications and search engine. The storage traces contain two financial traces (namely, *Financial1* and *Financial2*) and three web search traces (namely, *WebSearch1*, *WebSearch2*, and *WebSearch3*). Considering a server with an SSD may run multiple services simultaneously, we therefore combined the traces, generating a few different trace sets (see Table 1) for comparison. Note that, we need more write I/Os to invoke garbage collection and wear leveling, and the financial traces have much more write I/Os compared to that of the web search traces; therefore, each trace set contains both financial traces, but has a different web search trace.

For the flash storage simulator, we used DiskSim [27], a trace-driven disk simulator that includes modules for most secondary storage components. Originally DiskSim cannot simulate flash storage media. In 2009, Microsoft Research developed a package extends for DiskSim to enable SSD simulation.

[2]The process is as follows: 1) The data in the first block is copied to the free block; 2) The first block is erased; 3) The data in the second block is copied to the first block; 4) The second block is erased; 5) The data in the free block is copied to the second block; 6) The free block is added back to the free block pool, and will be erased during the idle time. Using RAM to help swapping blocks is not recommended, as RAM is vulnerable to power loss.

ID	Total # of duplicate pages	Size (GB)
1	286,393	2.18
2	320,810	2.44
3	286,887	2.19

Table 3: Experimental results

The configurations of the simulator are shown in Table 2. Some of the parameters in Table 2 are explained as follows:

- Minimum free blocks percentage: This is the minimal percentage of free blocks needed for the flash storage. If the number of free blocks drops and their percentage is below this minimum percentage, the garbage collection needs to be triggered. We chose a reasonable value 15% for our experiments.
- SSD_LIFE_TIME_THRESHOLD: This is a ratio between the remaining P/E cycles and the total P/E cycles of a flash block. When a flash block's remaining P/E cycles is less than $SSD\_LIFE\_TIME\_THRESHOLD * total\ P/E\ cycles$, the wear leveling should be invoked. We chose an empirical value 0.8 as the threshold in the experiments.
- Cleaning policy: This determines garbage collection and wear leveling strategy used in the simulator. We chose the wear-aware cleaning scheme, which is the optimal one having been implemented in the simulator.

Experimental process and results. We built DiskSim 4.0 with an SSD Model in a Ubuntu 14 virtual machine. We ran each trace set (Table 1) in the virtual machine, and calculated (i.e., by intercepting the FTL of the SSD simulator) the total number of duplicate pages generated in the flash memory during each run. Note that: 1) The simulator does not simulate bad block management which is more hardware-related and, therefore, the duplicate pages obtained in our experiments are generated by garbage collection and wear leveling only. 2) The total number of duplicates obtained is a mix of those generated by the garbage collection and those generated by wear leveling, and we were not able to explicitly distinguish them. This is because, the simulator uses a wear-aware cleaning scheme which mixes[3] the garbage collection and the wear leveling function.

The experimental results are shown in Table 3. We can observe that: First, a large number of duplicate pages (around 300K) were generated during running each trace set. This experimentally confirms that duplicates will be generated in the flash memory. Second, the number of duplicate pages generated is different when running different trace sets. This is reasonable as different trace sets have different I/Os, leading to different operations in the FTL.

[3]Specifically, when the garbage collection is triggered, the victim block having the least number of valid pages will be selected. Then, the simulator will calculate this block's remaining number of P/E cycles and, if it is less than a threshold, the wear leveling will be triggered as well.

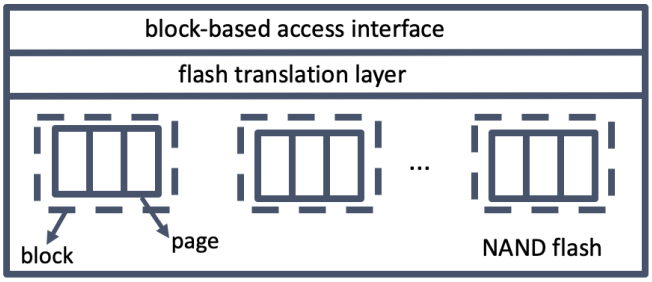

Figure 1: The architecture of a flash-based block device.

3.3 Compromising Existing Secure Deletion Schemes on Flash Memory by Leveraging Duplicates

Compromising the overwriting-based secure deletion schemes. Duplicates are created internally across the entire flash. Therefore, after target data have been securely deleted via block erasure [24, 25, 32, 84], scrubbing [46, 84, 85], or even the most recent analog scrubbing [43], duplicates of the deleted data likely remain in the flash memory. After secure deletion, the adversary can perform forensic analysis on the NAND flash, extracting duplicates of the deleted data for potential recovery.

Compromising the encryption-based secure deletion schemes. Most of the existing secure deletion schemes for flash memory store keys in the local storage [24, 52–54, 72] and securely remove them upon secure deletion. Since keys are stored in flash blocks, and data (i.e. keys) stored in those blocks may be duplicated by the internal management of flash memory (Sec. 3.1 and 3.2). Therefore, even if the original keys have been purged during secure deletion, by performing forensic analysis on the NAND flash, the adversary can extract duplicates to restore the deleted keys which can be used to decrypt the data claimed to have been deleted. Note that uncovering cryptographic keys from NAND flash is feasible via digital forensics and memory acquisition [59, 60].

4 SYSTEM AND ADVERSARIAL MODEL

System model. We mainly consider *computing devices equipped with flash memory as external storage*, in the form of flash-based block devices via FTL. These include servers/workstations equipped with ATA/ SCSI/ NVMe SSDs, and mobile computing devices (smartphones, tablets, IoT devices) equipped with memory cards (e.g., UFS cards, eMMC cards, MicroSD cards). The flash-based block device (Figure 1) usually exposes a block-based access interface, such that data can be read and written in "blocks", and a block-based file system (e.g., EXT4, FAT32) can be deployed. Let N be the storage capacity (in terms of **data nodes** [72], each is the unit of file system I/Os) of a block device. Let i be the block address, and $0 \leq i \leq N-1$. The block-based access interface provides two entry-points:

– Block_Device_Read(i, $\&data$): read a data node from block address i, and store it to the memory address $\&data$.

– Block_Device_Write(i, $\&data$): write a data node (stored in the memory $\&data$) to block address i.

NAND flash inside the block device consists of n erase blocks, and each flash block contains s pages. The FTL is responsible to translate a block address i to a flash memory address (x, y), where x is the erase block index, and y is the page index.

Adversarial model. We consider a computationally bounded adversary. The adversary can have access to a victim computing device multiple times, obtaining a snapshot of the storage medium upon each access (i.e., a *multi-snapshot adversary*). Note that the access time of the adversary is controlled by the user [71], so that the user can sanitize the sensitive data before exposing the device to the adversary. Each snapshot being obtained by the adversary includes information from the storage medium, which can be the physical image of the raw NAND flash, obtainable by forensic data recovery tools [21, 50, 79]. For a given data node which needs to be securely deleted, the user can only allow the adversary to capture snapshots outside its lifetime, preventing the data node from being restored trivially by the adversary after deletion. This type of multi-snapshot adversary is common in real world. For example, an attacker breaks into a hotel room, obtaining a snapshot of the memory card of a victim mobile device, aiming to restore sensitive data deleted by the device's owner; the attacker gained access to the device before, obtaining an earlier snapshot of the device.

We assume that the adversary will not take advantage of the correlation among the content of the data. Content may be correlated with each other, which could be a threat to secure deletion. However, content correlation requires knowledge on the high-level semantics of the data which is not suitable to be investigated in the lower-layer storage media. Equivalently, we assume there is no correlation among the content of the data.

Definitions. We define the secure deletion guarantee as: *Given a data node, we say it is securely deleted from a computing device if the multi-snapshot adversary cannot recover it by obtaining and analyzing snapshots of the device outside its lifetime.*

5 RedFlash

To simplify the presentation, we assume the size of a data node is equal to that of a page, since a page is the read/write unit of NAND flash. This can be easily extended to a general case where the size of a data node is a multiple of the page size. In the following, a "duplicate" refers to the duplicate data stored in a flash page.

5.1 Design Overview

A straw-man scheme. A basic approach could be: 1) We immediately delete any duplicates produced by internal management of flash storage (e.g., garbage collection, wear leveling, and bad block management). In this way, we ensure that the adversary cannot utilize any duplicates to recover the deleted data. 2) Once a deletion request is issued by the user from upper layer, we will immediately delete the corresponding data node from flash memory using block erasure. This straightforward solution, however, is expensive because: 1) for those data nodes which have duplicates but have not yet been deleted, it is unnecessary to remove their duplicates, as the existence of those duplicates does not "hurt" the secure deletion guarantee; and 2) a block erasure causes write amplification and is expensive when deleting data of small size.

Our approach is to wait until a secure deletion request is issued and then 1) to delete the targeted data node, and 2) to locate and delete any duplicates across flash memory associated with this data

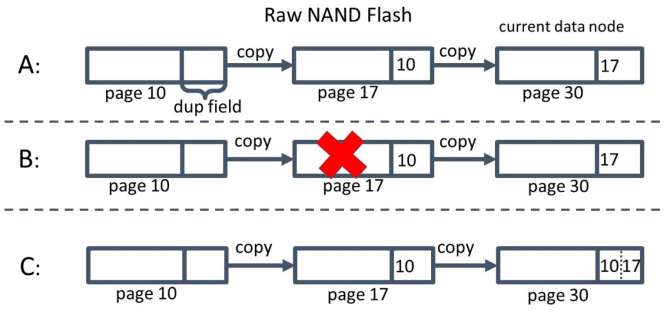

Figure 2: Duplicate chain.

node. A significant issue faced in our approach is how to efficiently locate duplicates associated with the data node being deleted. To address this issue, we follow two principles:

– *No searching*: One can simply search the duplicates across the entire flash memory in a brute-force manner. This however, should be avoided because: 1) searching the entire flash is prohibitively expensive; and 2) identical data may be present among files [61] originally (e.g., two files have identical data [77]), and searching duplicates based on the deleted data node may lead to mistakenly deleting data belonging to another file.

– *No RAM*: One can keep track of locations of all duplicates associated with each data node in RAM. This however, should be also avoided because: 1) RAM is volatile and suffers from failures like power-loss and is not a reliable source. 2) The embedded flash device is usually equipped with a limited amount of RAM.

Following the aforementioned design principles, we propose to "chain" each data node and its associated duplicates together across the flash (*no RAM*), generating a duplicate chain. In this way, once a data node is deleted, all the associated duplicates can be efficiently found by simply traversing the chain (*no searching*). We reserve an area in each flash page, called "dup field" , which is used to store information needed to maintain the duplicate chain. In Figure 2: A, we provide an example of a duplicate chain. Upon garbage collection, a valid data node stored in page 10 (note that we neglect the flash block index for now to simplify the presentation) is copied to a new page 17 of a new block; now page 17 stores the current data node, and page 10 stores a duplicate, and location 10 is written to the "dup field" of page 17 upon copying; after a while, wear leveling happens, the data node stored in page 17 is copied to another page 30 of a new block; now page 30 stores the current data node, and page 17 stores a duplicate, and the location 17 is written to the "dup field" of page 30 upon copying; *once the data node stored in page* 30 *is securely deleted, according to the "dup field" of page* 30*, we can efficiently locate page* 17*, and according to the "dup field" of page* 17*, we can efficiently locate page* 10. A few additional challenges need to be addressed as discussed as follows:

Challenge 1: Where can we store the "dup field" in a flash page?

Having observed that each flash page has an OOB area (Sec. 2.1), and only a few bytes of OOB have been used by the flash controller, we utilize the remaining unused space of the OOB to store the "dup field", instead of using extra other space in the corresponding page. This is advantageous because: Flash memory is update unfriendly, i.e., an update can only be performed after a block erasure is performed; therefore, once the OOB is initially written (e.g., 64 bytes out of 128 bytes have been used), it cannot be written again before the entire encompassing block is erased; therefore, the unused space of the OOB (e.g., 64 bytes) will be temporarily "freezed"; we use the unused space of the OOB to store the "dup field" during the initial write of the OOB, fully utilizing it before it is "freezed". Note that this will not affect regular functions of the flash controller, since this space is not used by other functions[4].

Challenge 2: How can we handle a "broken" duplicate chain?

A duplicate in a chain may be reclaimed earlier and the entire chain could be broken (see Figure 2: B). Using the previous example, there was a chain 10 → 17 → 30. However, before a data node stored in page 30 is deleted, the flash block encompassing page 17 may be reclaimed by the flash controller earlier, and hence page 17 has been sanitized and the corresponding "dup field" has been lost. In other words, we now lose connection with page 10, which stores a duplicate and should be sanitized once the data node stored in page 30 is securely deleted (see Figure 2: B). Note that flash memory is update unfriendly, and therefore, once page 17 is lost, updating the chain from 10 → 17 → 30 to 10 → 30 by directly overwriting the "dup field" of page 30 is infeasible except performing an expensive block erasure. Our idea is to make the chain more robust. Specifically, when adding a new page to the chain, we redundantly store all the prior locations in the chain to the "dup field" of this new page. Following the previous example, "dup field" of page 17 will store location of page 10, but "dup field" of page 30 will store location of both page 10 and 17 (see Figure 2: C). In this way, even if page 17 is erased, the chain will not be broken, since the "dup field" of page 30 also stores the location of page 10.

Another issue for a "broken" chain (Figure 2: B) is, suppose page 17 has been reclaimed earlier and may now have been used to store new data, we should not sanitize page 17 again once the data node stored in page 30 is securely deleted. But how can the flash controller know whether a page still remaining in the chain has been deleted before. This can be mitigated by checking each page before deleting data from it. If a page has been deleted before, it may now store inconsistent information (e.g., content stored may be different or the "dup field" stores something inconsistent). For example, page 17 is supposed to store identical content and its "dup field" is supposed to store location of page 10, but if it does not, then page 17 definitely has been deleted before. Note that reading a flash page is an order of magnitude faster than writing a page (Sec. 2.1), and therefore extra overhead due to checking pages in the duplicate chain will not be significant upon secure deletion. Rarely, there could be a special case that, page 17 has been deleted before and then used to store new identical data and the information stored in "dup field" is accidentally consistent. In the previous example, both page 10 and page 17 have been deleted earlier, and then used to store data associating with a new data node with identical data, accidentally creating a new chain 10 → 17. To handle this special case, we need to store in the "dup field" of each page in the chain a unique chain ID. A timestamp upon creating the chain may be used as this ID, but it may not be practical since a micro-second precision may not be enough and a higher precision requires large space for storing it. Having observed that each chain is associated with a data

[4]Usually there are other layers under the FTL, e.g., MTD, which will deny a page write if the FTL tries to use more than the unused space of the OOB.

node, which is bounded to a block device address (Sec. 4), we use this corresponding block address as the chain ID. When a page was deleted and used to store data associated with a new data node, its chain ID is surely different since two different data nodes will not have an identical block device address even though the two data nodes may have identical content. Note that: 1) the block device address i will be passed to the FTL when Block_Device_Write(i, $\&data$) is called; and 2) the size of this address is not large, e.g., for a 4TB flash device, with a 4KB data node size, it is at most 30 bits.

A last issue for a "broken" chain is, there may exist a special case that a page's "dup field" stores its own location. For example, as shown in Figure 3, the block encompassing page 17 is reclaimed earlier but later page 17 is used again to store new data which associates with the same data node, and thus the "dup field" of page 17 now stores location 17. Both page 17 in the duplicate chain will have the same chain ID. To mitigate this issue, upon secure deletion, when traversing the duplicate chain (starting from the deleted data node and traversing back) to delete duplicates, if a page has been deleted before, it will not be deleted again.

Challenge 3: How can we handle the growth of "dup field"?

Since we store in the "dup field" of a page all its previous locations, the size of the "dup field" may grow over time. Fortunately, according to our experiments (see Sec. 7.1), the length of a duplicate chain is usually small in practice and hence the number of previous locations is usually small. Therefore, we use the OOB area of a page to store it. For example, for a 4KB page with 128-byte OOB, if there are 64 bytes used for storing the 'dup field", we can store up to 21 previous locations (assuming each location is represented by 3 bytes). In addition, the "dup field" may contain obsolete locations since some of the flash blocks holding duplicates have been reclaimed earlier, and those obsolete locations will be purged periodically to reduce the size of the "dup field". *In the worst case if the "dup field" is too large to be stored entirely in each OOB, we will use regular space from the page to store it.*

5.2 Design Details

We elaborate the design details of RedFlash, a secure deletion scheme which can efficiently *Re*move both the data and the corresponding *d*uplicates from *Flash* memory. RedFlash is incorporated into the FTL by modifying its existing functions: garbage collection, wear leveling, bad block management. We also need to modify the block access interface Block_Device_Write(i, $\&data$) with secure deletion support and, the secure deletion in the FTL is triggered when Block_Device_Write(i, NULL) is called.

The page copy operation. A core function of RedFlash is a Page_Copy operation, which is used to copy data from a source page to a destination page (Figure 4). In this operation, to copy data from a source address to a destination address, we construct "dup field" of the destination page as: the "dup field" of the source page, appended by the location of the source page. Upon data copying, the newly constructed "dup field" will be written to the OOB of the destination page.

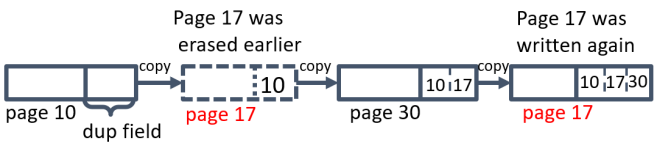

Figure 3: A special case for the robust duplicate chain.

Special functions of FTL. Special functions in the FTL will be modified as follows to support RedFlash.

– During *garbage collection*, the flash controller will pick a victim block, and copy data stored in all the valid pages of the victim block to a free block. When copying data from each valid page in the victim block to a page in the free block, we use Page_Copy.

– Upon *wear leveling*, the flash controller swaps a block (i.e., a cold block) which has the smallest erase count from blocks being used, with another block (i.e., a hot block) which has the largest erase count from the free block pool. This requires copy each page from the cold block to the hot block. When copying data from each page in the cold block to a page in the hot block, we use Page_Copy to replace the existing copy operation.

– *Bad block management* is designed to manage "bad" blocks. When a bad block is detected, the flash controller chooses an empty block from the free block pool, copies valid data from the bad block to this free block. When copying data from a valid page in the bad block, Page_Copy should be used.

Major operations of RedFlash. RedFlash is incorporated into a flash-based storage system by modifying the existing block access interface Block_Device_Write(i, $\&data$), as shown in Figure 5. When $\&data$ is not NULL, the system performs a regular write on

Page_Copy (i, j) /*copy data from page i to page j*/
(1) Read page i, including both content C_i and its "dup field" D_i
(2) Construct page j's "dup field"D_j as: $D_i || i$
(3) Write both C_i and D_j to page j, with D_j written to page j's OOB

Figure 4: The page copy operation in RedFlash

Block_Device_Write(i, $\&data$)
1. Translate i to a flash page location (x, y) by searching FTL's mapping table
2. If $data$ is NULL AND (x, y) is NULL, return FAILURE
3. If $data$ is NULL AND (x, y) is not NULL: /*secure deletion is performed on block address i*/
   (1) Read page (x, y), and extract its "dup field" D_0
   (2) If D_0 contains page locations $\{(x_i, y_i) | 1 \le i \le n\}$
   For $1 \le i \le n$:
   Read page (x_i, y_i), and extract "dup field" D_i
   Read chain ID from D_i
   If chain ID == i AND page (x_i, y_i) has not been deleted before, delete data from page (x_i, y_i)
   (3) Delete data from page (x, y)
   (4) Update FTL's mapping table by removing mapping for i
   (5) return SUCCESS
4. If (x, y) is not NULL, invalidate page (x, y) /*This is an overwrite on block address i*/
5. Request a new flash page (a, b)/*Write the new data to flash memory*/
6. Construct the "dup field" by filling its chain ID i
7. Write $data$ to page (a, b) and "dup field" to its OOB
8. Update FTL's mapping table by mapping i to page (a, b)
9. return SUCCESS

Figure 5: Major operations of RedFlash**. SUCCESS: 0; FAILURE: -1**

the block address i. If the block address i stores data before, this is an overwrite operation, and the corresponding flash page which stores the old data should be invalidated first (optionally, if the old data being overwritten need to be securely deleted, we can call Block_Device_Write(i, NULL) first). The new data will be stored to a new flash page, and before writing it, we need to initialize its "dup field" by embedding a unique chain ID, which is the block address i. When $\&data$ is NULL, the system performs secure deletion on the block address i, and the data node stored in the corresponding flash page should be deleted. The FTL will read this page, extracting its "dup field". If the "dup field" does not contain any locations, this data node does not have any duplicates, secure deletion ends. Otherwise, the FTL will check each location, read the corresponding page, and check each "dup field". The duplicate stored in a page will be deleted if its "dup field" contains a matched chain ID (i.e., i) and the duplicate stored in the page has not been deleted yet.

Efficiently deleting data stored in a flash page. Deleting data from a flash page is non-trivial considering flash memory is "update unfriendly" (Sec. 2.1). To enable efficiently deleting data from a flash page, we use scrubbing [85] for SLC NAND flash. For MLC NAND flash, simply performing scrubbing on the target page cannot work [46, 85], since multiple bits share one cell in MLC chips and, scrubbing one page will cause corruptions to other pages (i.e., paired pages) in the same block. Deletion on data stored in an MLC NAND flash page can be efficiently performed by adapting the scrubbing technique as: relocating the content in the paired pages, and performing scrubbing on both the target and the paired page. Note that the paired page can be easily located based on the type of flash memory [46]. The modifications for MLC can be summarized as: 1) When secure deletion is triggered from the user, for each page in the duplicate chain, the content of its paired-page will be read and written to a new flash page, and the mapping table will be updated correspondingly; 2) In Block_Device_Write(i, $\&data$) (Figure 5), step 3(3) will be updated as: "delete data from page (x,y) and its paired page using scrubbing"; step 3(4) will be updated as "Update FTL's mapping table by removing mapping for i and its paired page".

Extra considerations. 1) Handling duplicates from read disturb management: Read disturb management requires re-locating read-frequently blocks/pages elsewhere. The obsolete blocks/pages typically will not be reclaimed immediately for performance consideration, and data stored in them will be temporarily preserved and may be utilized to compromise the secure deletion guarantee. To mitigate this compromise, we can add the duplicates generated by read disturb management to the duplicate chain, and eliminate them upon secure deletion. 2) Mitigating data leakage at the voltage level: Even though data have been securely deleted at the digital level, flash memory cells may hold their "imprints" at the analog level. If the adversary can successfully extract such analog signal, it may compromise the secure deletion guarantee. Those remnants can be eliminated by sanitizing data at the lower voltage level by using analog scrubbing [43] instead of conventional scrubbing. 3) Other remnants which may affect secure deletion: The past existence of the deleted data may affect the layout of data storage and, even if the data are deleted, such an impact may be preserved in the flash memory [24], leading to the compromise of the secure deletion guarantee. This type of "remnants" can be taken care by randomizing the storage layout [24]. 4) Handing legacy flash devices: RedFlash needs to be integrated into the FTL. Therefore, to allow the legacy flash devices to use RedFlash, a potential solution could be upgrading the flash firmware via tools provided by manufacturers [44, 74]. 5) About the usage of OOB: The OOB area is often used to store the error correction code (ECC), and RedFlash will utilize the remaining space to avoid interfering the original functionality. For example, a 2KB flash page is associated with 64 bytes of OOB, and typically only 12 bytes are used to store ECC [73, 75]; in other words, 52 bytes can be used by RedFlash to store a "dup field" of the duplicate chain. This is usually sufficient regardless of the storage size, because: The size of the "dup field" is determined by the number of bits required for representing each page number as well as the length of the duplicate chain; the size of each page number grows when the storage capacity grows in a logarithmic way, which scales pretty well; the duplicate chain is usually short in length (Sec. 7.1) regardless of the storage size, as a good FTL design will not preserve too many copies at the same time to avoid wasting too much space. In the worst case if the OOB does not have enough space to store the "dup field", RedFlash can use some of the regular space in the corresponding flash page.

User steps. Secure deletion is usually performed by users staying in the user space. To invoke secure deletion incorporated in the FTL, the user application can intercept the existing "write (fp, buf, nbytes)" system call, where "fp" is the file pointer (determined by the file descriptor and the offset), "buf" is the memory address of the data being written, and "nbytes" is the size of the data being written. By setting "buf" to NULL, it can issue a secure deletion request on file data located at "fp" with length "nbytes". Based on "fp" and "nbytes", the file system can calculate the corresponding block addresses $i_1, i_2, \cdots, i_k$ (assuming there are k blocks storing the file data being deleted), and then invoke Block_Device_Write(i, NULL), where $i \in \{i_1, i_2, \cdots, i_k\}$.

6 SECURITY ANALYSIS

In the following, we first show that, for any given data node D, the straw-man scheme can ensure secure deletion of D against the multi-snapshot adversary. We then show RedFlash can achieve a security guarantee comparable to the straw-man scheme.

The straw-man scheme can ensure secure deletion of D. Since the adversary can obtain snapshots of raw NAND flash (Sec. 4), by performing forensic analysis on the captured snapshots, the adversary can only obtain two types of flash pages: type I - pages with all '1's (i.e., an empty page which is programmable); and type II - pages storing actual data/metadata. The adversary surely cannot learn anything about D from the type I pages. The adversary cannot learn anything about D from the type II pages either, because: First, the data/metadata stored in the type II pages of the captured snapshots will not be the duplicates of D, since they are completely sanitized from the NAND flash upon deleting D and the adversary can only capture snapshots outside the lifetime of D. Second, the data/metadata stored in the type II pages from the captured snapshots will not correlate to D based on our assumption (Sec. 4).

RedFlash **can achieve a security guarantee comparable to the straw-man scheme**. Compared to the straw-man scheme, the

adversary in RedFlash may obtain two additional types of pages in the captured snapshots: type III - pages with all '0's (this type of pages exists after scrubbing is performed on them); and type IV - pages storing data duplicates (together with "dup fields") of other data than D. We show in the following that utilizing the extra types of pages will not give the adversary additional advantages of recovering the deleted data node D.

From the type-III pages, the adversary surely cannot learn anything about D. From the type-IV pages, we discuss two cases: 1) From duplicates of other data rather than D, the adversary cannot learn anything about D either, since they are not correlated to D based on our assumption (Sec. 4). 2) From "dup fields", the adversary can derive duplicate chains. However, since the "dup fields" corresponding to D have been completely sanitized upon deleting D, the duplicate chains obtained here will have nothing to do with D and cannot be used to derive any knowledge about D.

7 EXPERIMENTAL EVALUATION AND SIMULATION

7.1 Experimental Results for SLC Flash

Real-world implementation and experimental setup. We implemented RedFlash into OpenNFM [28], an open-source NAND flash controller framework, which has implemented the major operations of an FTL in C. We have modified a few major functions, including garbage collection, wear leveling, and bad block management in the OpenNFM (refer to Sec. 5.2 for details of the new implementations for those functions). In addition, we have intercepted the *FTL_Write* function so that it can handle both the regular write requests invoked by Block_Device_Write(i, &$data$) and the secure deletion requests invoked by Block_Device_Write(i, NULL).

To perform experimental evaluation over the SLC NAND flash, we ported [82] RedFlash to LPC-H3131[5], an electronic development board equipped with 180MHz ARM micro-controller, 512MB SLC NAND flash (consisting of approximately 4,000 erase blocks, and 64 pages in each block), and 32 MB SDRAM. After RedFlash is ported, the electronic board can be used as a secure USB device supporting secure deletion. The device was attached to a host computing device (Firefly AIO-3399J [9], Six-Core ARM 64-bit processor, up to 1.8GHz, 4GB RAM, Ubuntu 16) and used as external storage like any regular USB devices. We used benchmark tool fio [35] to evaluate the throughput, which was running in the host computing device and performing I/Os on the attached USB device (formatted using FAT32). When evaluating the throughput, we did not run the the real-world workloads used in Sec. 3.2, as they fit the large-size disk only, but our LPC-H3131 only has 512MB flash storage.

Evaluating lengths of duplicate chains. When adding a new page to a duplicate chain, RedFlash needs to store all locations of prior duplicates in the chain to "dup field" of the new page. Since we only have limited space to store the "dup field", to avoid overwhelming the "dup filed", each duplicate chain should not be too long. We therefore measured the lengths of the duplicate chains by stressing the flash device as follows: we kept writing data to the flash device and, once the device was completely filled, we deleted all the data; we repeated the aforementioned process until the accumulated data being written reach 50GB. Under different wear leveling threshold, we obtained the length of the longest duplicate chain. The results are shown in Table 4. We can observe that the length of the duplicate chain does not exceed 3 under different wear leveling thresholds. The results indicate that a duplicate chain is usually not too long in practice. This is reasonable, since a long duplicate chain indicates that there are too many invalid pages simultaneously present in the flash memory, which is usually avoided by a good FTL design.

Wear leveling threshold	Length
10	3
20	3

Table 4: The length of the longest duplicate chains.

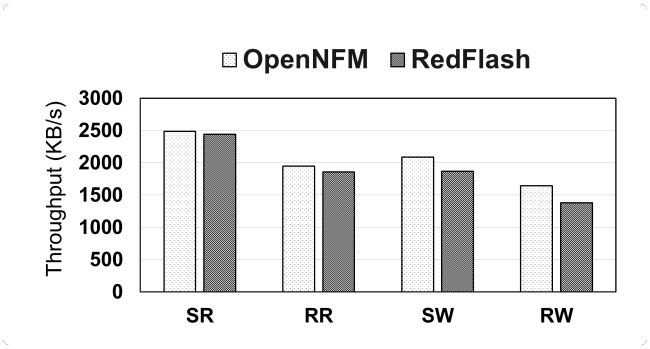

Figure 6: Comparison of I/O throughput between OpenNFM and RedFlash, obtained from fio benchmark under different read/write patterns. SR: sequential read; RR: random read; SW: sequential write; RW: random write.

Throughput. To assess how RedFlash affects throughput of a regular (non-secure) flash storage device, we used original OpenNFM as a baseline for comparison. Note that we did not compare RedFlash with existing secure deletion schemes for flash memory, since none of them can handle duplicates towards ensuring secure deletion. The threshold for wear leveling was set as 10 (i.e., when the difference of P/E cycles of two blocks exceeds 10, wear leveling is triggered) which can ensure good wear leveling effectiveness. The results are shown in Figure 6. The benchmark evaluated different I/O patterns including sequential read/write and random read/write. We can observe that, 1) RedFlash has little influence on read operations. This is because, RedFlash does not need to modify any logic relating to reads. 2) RedFlash has influence on write operations. The write throughput of RedFlash decreases around 10% compared to that of OpenNFM. This is due to additional operations added to various functions of FTL, causing additional overhead to writes. This concludes that compared to a regular non-secure flash device (for SLC flash), RedFlash achieves the secure deletion guarantee at the cost of a small additional overhead.

We also assessed the write throughput of RedFlash using real-world I/O workloads. We chose another workload traces which can match the storage capacity of the LPC-H3131. The traces were a portion of SNIA (Storage Networking Industry Association) Nexus 5 Smartphone Traces [5], as listed in Table 5. We did not perform

[5] As a low-end electronic board, LPC-H3131 only has 1-2MB/s throughput. This will not affect most of our experimental results except throughput. However, our evaluation on throughput is to show additional overhead compared to the baseline scheme, by running both of them in this low-end board. This additional overhead will not be significantly different when changing to a high-end board with 100MB/s throughput.

Name	Read(MB)	Write(MB)
log176_booting	736	223
log111_email	12	45
log166_webBrowsing	16	77
log156_download	1	712
log186_twitter	55	129
log191_facebook	27	67
log260_faceBookHandOuts	50	129
log201_googleMap	46	147
log235_radioFacebook	19	131
log225_musicTwitter	80	215
log225_musicFacebook	162	270
log230_musicMessage	50	179
log220_musicWebBrowse	120	162
log121_movie	123	4
log152_youtube	1	27
log161_cameraVideo	248	1981
log171_angryBird	24	67

Table 5: Real-world traces collected from SNIA

each trace individually; instead, we created a few trace sets, each of which contains a few different traces. This is because: 1) each trace itself is not large enough to write enough data to the flash storage device to invoke special functions in the FTL like wear leveling; 2) a real user usually runs multiple different workloads in his/her computing device, and grouping multiple workload traces will be closer to real-world scenarios. To simulate different users' behaviors when using their computing devices, we combine the traces in Table 5, creating 4 trace sets, each of which is a collection of traces, representing a typical user profile:

- User profile #1: a user which mainly uses his/her computing device for work, and the corresponding trace set #1 is: {*log176, log111, log166, log156*}.
- User profile #2: a user which mainly uses his/her device for social networking, and the corresponding trace set #2 is: {*log176, log186, log191, log260, log201, log235*}.
- User profile #3: a user which mainly uses his/her device for listening music, and the corresponding trace set #3 is: {*log176, log225(twitter), log225(facebook), log230, log220*}.
- User profile #4: a user which mainly uses his/her device for video and gaming, and the corresponding trace set #4 is: {*log176, log121, log152, log161, log171*}.

Each record in a SNIA trace contains block address and size of the data being read/written, but does not contain the actual content due to the privacy concerns. We therefore created a large content file containing randomly generated data as the data source. We fixed the wear leveling threshold as 10. To execute each write record in the trace, we sequentially read a different chunk of data from the content file and write it to the storage medium. Note that: 1) the size of the chunk is determined by the write size of the record, and 2) we neglect the block address of each record. The results are shown in Figure 7, which double confirm that RedFlash does not cause too much degradation in write throughput when running real-world I/O workloads.

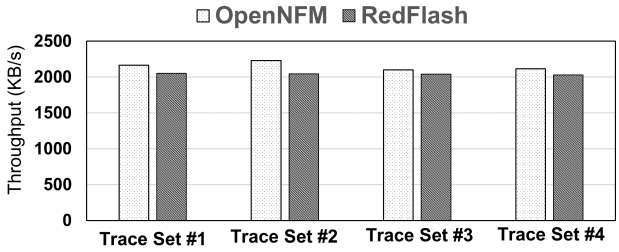

Figure 7: Comparison of write throughput between OpenNFM and RedFlash, obtained by running real-world workload traces.

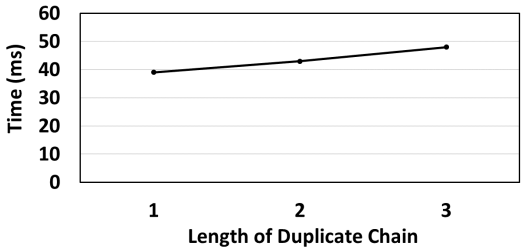

Figure 8: Time (ms) for securely deleting a data node in SLC flash.

Wear leveling threshold	WLI
10	1.87%
20	2.22%

Table 6: The WLI value of RedFlash under different wear leveling thresholds.

Overhead for secure deletion. We evaluated the time needed for securely deleting a data node (stored in a page) when the length of a duplicate chain varies. Note that when measuring the secure deletion time, we did not run extra workloads in the device, and therefore, the time is purely for secure deletion without affected by other workloads. The result is shown in Figure 8. We can observe that, the time needed for secure deletion is linear to the length of the duplicate chain. This is reasonable, since RedFlash will remove both the targeted data and the corresponding duplicates upon secure deletion. The longer the duplicate chain, the more data needed to be deleted upon secure deletion.

Wear leveling. The service life of flash memory is mainly determined by effectiveness of wear leveling. We thus assess impact of RedFlash on the wear leveling. We use Hoover economic wealth inequality indicator [72], which calculates an appropriately normalized sum of difference of each measurement to the mean. Assuming there are n erase blocks with erase count $e_1, e_2, ...e_n$, respectively, we calculate the wear leveling inequality (WLI for short) value as: $WLI = \frac{1}{2}\sum_{i=1}^{n} || \frac{e_i}{E} - \frac{1}{n} ||$, where $E = \sum_{i=1}^{n} e_i$. Intuitively, WLI indicates the fraction of erasures that should be re-assigned to other blocks to ensure completely even wear [72] and a smaller WLI value will imply a better wear leveling effectiveness.

To calculate WLI value of RedFlash, we continuously wrote data to attached USB device incorporating RedFlash, and erased the device once filled. We repeated aforementioned process until the

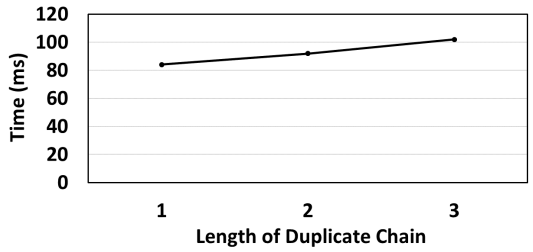

Figure 9: Time (ms) for securely deleting a data node in MLC.

total amount of data written reached 50 GB. The wear leveling threshold is 10 and 20 for each run, respectively. The results for WLI are shown in Table 6. We can observe that WLI values are around 2% under different wear leveling thresholds. These small values indicate good wear leveling effectiveness of RedFlash. We can also observe that, the WLI value is smaller when the wear leveling threshold is smaller. This is because: for a smaller wear leveling threshold, wear leveling will be triggered more frequently, and erasures are eventually distributed more evenly across the flash.

7.2 Simulation Results for MLC Flash

The major performance difference between RedFlash on SLC and MLC flash is deleting data stored on a flash page. We therefore also assess the performance of RedFlash on securely deleting a data node from MLC via simulations.

Simulation setup. We still relied on the testbed (LPC-H3131, which runs RedFlash and is attached to the host computing device via USB) built for SLC. The difference is, when the secure deletion on a data node is invoked, we simulate each read/program operation using a delay function, rather than actually perform it (LPC-H3131 is equipped with SLC only, rather than MLC). The amount of time being delayed for each read/program is corresponding to some typical value from the datasheet of MLC flash [3]. In this manner, we can measure the time needed for securely deleting a data node from MLC flash.

Overhead for secure deletion on MLC. We measured the time needed for securely deleting a data node when the length of its corresponding duplicate chain varies. Note that the time is purely for secure deletion without affecting by other workloads. The results are shown in Figure 9. We can observe that: 1) For MLC, the time needed for securely deleting a data node is linear with the length of its duplicate chain. The reason is, the time needed to securely remove data from an MLC flash page is pretty constant and, the amount of data needed to be deleted for securely deleting this data node is linear with the length of its duplicate chain. 2) Under the same length of duplicate chain, secure deletion on MLC requires more time than that of SLC. This is because: in MLC, RedFlash will not only scrub pages associated with the duplicate chain, but also the paired pages (relocating data stored on them before deletion is performed), which brings extra overhead.

8 RELATED WORK

We briefly summarize existing secure deletion approaches. Thorough surveys about secure deletion approaches can be found in various sources [71, 91].

Secure deletion on cloud and database systems. A large number of existing research works tried to securely dispose of data outsourced to the clouds [62, 63, 70, 80, 81, 83, 86]. This is achieved by encrypting the outsourced data, and disposing of keys such that the encrypted data become inaccessible. Several other schemes investigated how to securely remove data from databases [11, 12] and storage systems [19, 22, 37, 90]. Especially, Neuralyzer [89] and EphPub [23] leveraged domain name system caching to ensure successful deletion of data when a prefixed lifetime expires.

Secure deletion on file systems/block devices. Several existing works studied the secure deletion problem in the file system layer [10, 15, 20, 48, 67, 69] or the block device layer [66]. In general, over-writing [15, 48] and encryption with ephemeral key [20, 67] are used to securely remove data from the file systems/block devices. They cannot ensure secure data deletion on flash storage media because: due to the out-of-place update, data deleted from upper layers (e.g., the file system) will be temporarily preserved in the flash memory until garbage collection is performed.

Secure deletion on flash memory. Lee et al. [53, 54] proposed a secure deletion scheme by forcing current and previous keys of a file to be stored in the same flash block, so that a file can be deleted by a single block erasure. Reardon et al. [72] proposed data node encrypted file system (DNEFS), in which they divided the entire flash memory into two areas: a data storage area and a key storage area. They encrypted each data node with a unique key, and collocated keys in the key storage area. Secure deletion is achieved by deleting keys in a batch. Since keys are condensed in a small area, one block erasure can eliminate multiple keys. Peters et al. proposed DEFY [68], which transformed data using all-or-nothing transform (AONT), creating a small message expansion. Data can be efficiently deleted by simply deleting the small expansion due to the all-or-nothing property.

Having observed that programming a single "1" bit to "0" is possible (the reverse operation is not allowed), Wei et al.[85] proposed scrubbing to efficiently delete data from a flash page. Wang et al. [84] minimized scrubbing overheads by organizing data in such a way that creates scrubbing-friendly patterns. Hasan et al. [43] observed data leakage at the voltage level after the scrubbing is performed, and proposed analog scrubbing as a remediation. Analog scrubbing can be used to replace the scrubbing operation in RedFlash if we want to eliminate data leakage at the voltage level of SLC flash. TrueErase [32, 33] proposed a full-storage-data-path framework that performs per file secure deletion and works with common file systems and solid-state storage. The data in the flash memory are deleted via block erasure. HiFlash [25], NFPS [46] and TedFlash [24] explored another type of remnants, namely, the impact of the past existence of the deleted data on the flash memory layout, which is orthogonal to our work.

9 CONCLUSION

Data duplicates produced by unique internal functions of flash translation layer may lead to compromise of secure deletion guarantee.

In this work, we have both theoretically and experimentally confirmed the existence of such data remnants. We propose RedFlash, a secure deletion scheme which can efficiently remove both data and the corresponding duplicates across the entire flash. Security analysis and experimental evaluation show that RedFlash can achieve the secure deletion guarantee, at the cost of a small performance degradation, compared to a regular (non-secure) flash controller.

ACKNOWLEDGMENTS

This work was supported by US National Science Foundation under grant number 1938130-CNS, 1928349-CNS, and 2043022-DGE. We would like to thank our shepherd, Haehyun Cho, and anonymous reviewers for their insightful suggestions for improving this paper.

REFERENCES

[1] Technical Note - Garbage Collection in Single-Level Cell NAND Flash Memory. https://www.micron.com/-/media/client/global/Documents/Products/Technical%20Note/NAND%20Flash/tn2960_garbage_collection_slc_nand.ashx.
[2] Jffs2. https://www.sourceware.org/jffs2/, 2003.
[3] Samsung K9GAG08B0M Datasheet . https://www.datasheet.directory/index.php?title=Special:PdfViewer&url=https%3A%2F%2Fdatasheet.iiic.cc%2Fdatasheets-0%2Fsamsung%2FK9GAG08U0M-PIB0.pdf, 2007.
[4] Memory Technology Device (MTD) Subsystem for Linux. http://www.linux-mtd.infradead.org/nand-data/nanddata.html, 2011.
[5] SNIA I/O Trace Data Files. http://iotta.snia.org/traces/, 2011.
[6] Dod manual 5220.22-m. https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodm/522022M.pdf?ver=2017-04-17-134632-467, 2016.
[7] Regulation (eu) 2016/679 of the european parliament and of the council. https://eur-lex.europa.eu/eli/reg/2016/679/oj, 2016.
[8] F2fs, 2019. https://www.kernel.org/doc/Documentation/filesystems/f2fs.txt.
[9] Firefly AIO-3399J . https://en.t-firefly.com/product/industry/aio_3399, 2021.
[10] Bikash Agrawal, Raymond Hansen, Chunming Rong, and Tomasz Wiktorski. Sdhdfs: Secure deletion in hadoop distributed file system. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 181–189. IEEE, 2016.
[11] Ahmed A Ataullah, Ashraf Aboulnaga, and Frank Wm Tompa. Records retention in relational database systems. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 873–882. ACM, 2008.
[12] Sumeet Bajaj and Radu Sion. Ficklebase: Looking into the future to erase the past. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 86–97. IEEE, 2013.
[13] Sumeet Bajaj and Radu Sion. Hifs: History independence for file systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1285–1296. ACM, 2013.
[14] Bruce McNutt Ken Bates. Umasstracerepository-search engine i/o. http://traces.cs.umass.edu/index.php/Storage/Storage, 2002.
[15] Steven Bauer and Nissanka Bodhi Priyantha. Secure data deletion for linux file systems. In *Usenix Security Symposium*, volume 174, 2001.
[16] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. Key recovery attacks of practical complexity on aes-256 variants with up to 10 rounds. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 299–319. Springer, 2010.
[17] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full aes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 344–371. Springer, 2011.
[18] Daniel Boteanu and Kevvie Fowler. Bypassing self-encrypting drives (sed) in enterprise environments. In *Proceedings of the Black Hat Europe Conference*, 2015.
[19] Fabiano C Botelho, Philip Shilane, Nitin Garg, and Windsor Hsu. Memory efficient sanitization of a deduplicated storage system. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 81–94, 2013.
[20] Alexandre Melo Braga and Alfredo H Gallinucci Colito. Adding secure deletion to an encrypted file system on android smartphones. In *Proc. SECURWARE*, pages 106–110, 2014.
[21] Marcel Breeuwsma, Martien De Jongh, Coert Klaver, Ronald Van Der Knijff, and Mark Roeloffs. Forensic data recovery from flash memory. *Small Scale Digital Device Forensics Journal*, 1(1):1–17, 2007.
[22] Christian Cachin, Kristiyan Haralambiev, Hsu-Chun Hsiao, and Alessandro Sorniotti. Policy-based secure deletion. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 259–270. ACM, 2013.
[23] Claude Castelluccia, Emiliano De Cristofaro, Aurelien Francillon, and Mohamed-Ali Kaafar. Ephpub: Toward robust ephemeral publishing. In *2011 19th IEEE International Conference on Network Protocols*, pages 165–175. IEEE, 2011.
[24] Bo Chen, Shijie Jia, Luning Xia, and Peng Liu. Sanitizing data is not enough!: towards sanitizing structural artifacts in flash media. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 496–507. ACM, 2016.
[25] Bo Chen and Radu Sion. Hiflash: A history independent flash device. *arXiv preprint arXiv:1511.05180*, 2015.
[26] Siddharth Choudhuri and Tony Givargis. Deterministic service guarantees for nand flash using partial block cleaning. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 19–24. ACM, 2008.
[27] CMU. Disksim simulator. https://www.pdl.cmu.edu/DiskSim/index.shtml, 2008.
[28] Google Code. Opennfm. https://code.google.com/p/opennfm/, 2011.
[29] United States Congress. Health Insurance Portability and Accountability Act. http://www.hhs.gov/ocr/privacy/index.html, 1996.
[30] Patrick Derbez, Pierre-Alain Fouque, and Jérémy Jean. Improved key recovery attacks on reduced-round aes in the single-key setting. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 371–387. Springer, 2013.
[31] Ming Di Leom. *Remote Wiping in Android.* PhD thesis, University of South Australia, 2015.
[32] 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.
[33] Sarah Diesburg, Christopher Meyers, Mark Stanovich, An-I Andy Wang, and Geoff Kuenning. Trueerase: Leveraging an auxiliary data path for per-file secure deletion. *ACM Transactions on Storage (TOS)*, 12(4):18, 2016.
[34] Yajuan Du, Wei Liu, Yuan Gao, and Rachata Ausavarungnirun. Observation and optimization on garbage collection of flash memories: The view in performance cliff. *Micromachines*, 12(7):846, 2021.
[35] Freecode. fio. http://freecode.com/projects/fio, 2014.
[36] Simson L Garfinkel and Abhi Shelat. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security & Privacy*, 99(1):17–27, 2003.
[37] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. Vanish: Increasing data privacy with self-destructing data. In *USENIX Security Symposium*, volume 316, 2009.
[38] Le Guan, Shijie Jia, Bo Chen, Fengwei Zhang, Bo Luo, Jingqiang Lin, Peng Liu, Xinyu Xing, and Luning Xia. Supporting transparent snapshot for bare-metal malware analysis on mobile devices. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 339–349. ACM, 2017.
[39] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, volume 44. ACM, 2009.
[40] Keonsoo Ha, Jaeyong Jeong, and Jihong Kim. A read-disturb management technique for high-density nand flash memory. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 13. ACM, 2013.
[41] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.
[42] 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.
[43] Md Mehedi Hasan and Biswajit Ray. Data recovery from "scrubbed" nand flash storage: Need for analog sanitiza-tion. In *The 29th Usenix Security Symposium*, 2020.
[44] Intel. Intel ssd firmware update tool. https://www.intel.com/content/www/us/en/download/17903/intel-ssd-firmware-update-tool.html, 2021.
[45] Kee-Hoon Jang and Tae Hee Han. Efficient garbage collection policy and block management method for nand flash memory. In *2010 2nd International Conference on Mechanical and Electronics Engineering*, volume 1, pages V1–327. IEEE, 2010.
[46] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Nfps: Adding undetectable secure deletion to flash translation layer. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 305–315. ACM, 2016.
[47] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Deftl: Implementing plausibly deniable encryption in flash translation layer. In *Proceedings of the 24th ACM conference on Computer and communications security*. ACM, 2017.
[48] Nikolai Joukov and Erez Zadok. Adding secure deletion to your favorite file system. In *Third IEEE International Security in Storage Workshop (SISW'05)*, pages 8–pp. IEEE, 2005.
[49] Sanghyuk Jung and Yong Ho Song. Link-gc: a preemptive approach for garbage collection in nand flash storages. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1478–1484. ACM, 2013.
[50] Keonwoo Kim, Dowon Hong, Kyoil Chung, and Jae-Cheol Ryou. Data acquisition from cell phone using logical approach. *Proceedings of the world academy of science, engineering and technology*, 26, 2007.
[51] Richard Kissel, Matthew A Scholl, Steven Skolochenko, and Xing Li. Sp 800-88 rev. 1. guidelines for media sanitization, 2006.

[52] 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.

[53] 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*, pages 1710–1714. ACM, 2008.

[54] 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.

[55] Lei Lei, Quanwei Cai, Bo Chen, and Jingqiang Lin. Towards efficient re-encryption for secure client-side deduplication in public clouds. In *International Conference on Information and Communications Security*, pages 71–84. Springer, 2016.

[56] Ming Di Leom, Kim-Kwang Raymond Choo, and Ray Hunt. Remote wiping and secure deletion on mobile devices: A review. *Journal of forensic sciences*, 61(6):1473–1492, 2016.

[57] Jingwei Li, Chuan Qin, Patrick PC Lee, and Jin Li. Rekeying for encrypted deduplication storage. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 618–629. IEEE, 2016.

[58] Jianwei Liao, Fengxiang Zhang, Li Li, and Guoqiang Xiao. Adaptive wear-leveling in flash-based memory. *IEEE Computer Architecture Letters*, 14(1):1–4, 2014.

[59] Carsten Maartmann-Moe, Steffen E Thorkildsen, and André Årnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. *digital investigation*, 6:S132–S140, 2009.

[60] Carlo Meijer and Bernard Van Gastel. Self-encrypting deception: weaknesses in the encryption of solid state drives. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 72–87. IEEE, 2019.

[61] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14, 2012.

[62] Zhen Mo, Yan Qiao, and Shigang Chen. Two-party fine-grained assured deletion of outsourced data in cloud systems. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 308–317. IEEE, 2014.

[63] Zhen Mo, Qingjun Xiao, Yian Zhou, and Shigang Chen. On deletion of outsourced data in cloud computing. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 344–351. IEEE, 2014.

[64] Tilo Müller, Tobias Latzo, and Felix C Freiling. Self-encrypting disks pose self-decrypting risks. In *the 29th Chaos Communinication Congress*, pages 1–10, 2012.

[65] Muthukumar Murugan and David HC Du. Rejuvenator: A static wear leveling algorithm for nand flash memory with minimized overhead. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2011.

[66] Kaan Onarlioglu, William Robertson, and Engin Kirda. Eraser: Your data won't be back. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 153–166. IEEE, 2018.

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

[68] Timothy M Peters, Mark A Gondree, and Zachary NJ Peterson. DEFY: A deniable, encrypted file system for log-structured storage. In *22th Annual Network and Distributed System Security Symposium, NDSS*, 2015.

[69] Zachary NJ Peterson, Randal C Burns, Joseph Herring, Adam Stubblefield, and Aviel D Rubin. Secure deletion for a versioning file system. In *FAST*, volume 5, pages 143–154, 2005.

[70] Arthur Rahumed, Henry CH Chen, Yang Tang, Patrick PC Lee, and John CS Lui. A secure cloud backup system with assured deletion and version control. In *2011 40th International Conference on Parallel Processing Workshops*, pages 160–167. IEEE, 2011.

[71] Joel Reardon, David Basin, and Srdjan Capkun. Sok: Secure data deletion. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 301–315. IEEE, 2013.

[72] 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.

[73] Samsung. K9f4g08u0a datasheet (pdf). https://pdf1.alldatasheet.com/datasheet-pdf/view/135880/SAMSUNG/K9F4G08U0A.html.

[74] Samsung. Samsung ssd magician7 software. https://semiconductor.samsung.com/consumer-storage/magician/, 2022.

[75] SkyHighMemory. What types of ecc should be used on flash memory? http://www.skyhighmemory.com/download/applicationNotes/001-99200_AN99200_What_Types_of_ECC_Should_Be_Used_on_Flash_Memory.pdf.

[76] Avinash Srinivasan, Jie Wu, Panneer Santhalingam, and Jeffrey Zamanski. Deaddrop-in-a-flash: Information hiding at ssd nand flash memory physical layer. *SECURWARE 2014*, page 79, 2014.

[77] Mark W Storer, Kevin Greenan, Darrell DE Long, and Ethan L Miller. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 1–10, 2008.

[78] Raja Subramani, Haritima Swapnil, Niharika Thakur, Bharath Radhakrishnan, and Krishnamurthy Puttaiah. Garbage collection algorithms for nand flash memory devices–an overview. In *2013 European Modelling Symposium*, pages 81–86. IEEE, 2013.

[79] Steven Swanson and Michael Wei. Safe: Fast, verifiable sanitization for ssds. *San Diego, CA: University of California-San Diego*, 2010.

[80] 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.

[81] Yang Tang, Patrick PC Lee, John CS Lui, and Radia Perlman. Fade: Secure overlay cloud storage with file assured deletion. In *International Conference on Security and Privacy in Communication Systems*, pages 380–397. Springer, 2010.

[82] Deepthi Tankasala, Niusen Chen, and Bo Chen. A step-by-step guideline for creating a testbed for flash memory research via lpc-h3131 and opennfm. 2020.

[83] Shin Tezuka, Ryuya Uda, and Kenichi Okada. Adec: Assured deletion and verifiable version control for cloud storage. In *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, pages 23–30. IEEE, 2012.

[84] Wei-Chen Wang, Chien-Chung Ho, Yuan-Hao Chang, Tei-Wei Kuo, and Ping-Hsien Lin. Scrubbing-aware secure deletion for 3-d nand flash. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2790–2801, 2018.

[85] 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, 2011.

[86] Jinbo Xiong, Ximeng Liu, Zhiqiang Yao, Jianfeng Ma, Qi Li, Kui Geng, and Patrick S Chen. A secure data self-destructing scheme in cloud computing. *IEEE Transactions on Cloud Computing*, 2(4):448–458, 2014.

[87] Yaffs. Yaffs. http://www.yaffs.net/, 2002.

[88] Ming-Chang Yang, Yu-Ming Chang, Che-Wei Tsao, Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. Garbage collection and wear leveling for flash memory: Past and future. In *Smart Computing (SMARTCOMP), 2014 International Conference on*, pages 66–73. IEEE, 2014.

[89] 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 Conference on Data and Application Security and Privacy*, pages 14–25. ACM, 2016.

[90] Lingfang Zeng, Shibin Chen, Qingsong Wei, and Dan Feng. Sedas: A self-destructing data system based on active storage framework. In *APMRC, 2012 Digest*, pages 1–8. IEEE, 2012.

[91] Qionglu Zhang, Shijie Jia, Bing Chang, and Bo Chen. Ensuring data confidentiality via plausibly deniable encryption and secure deletion–a survey. *Cybersecurity*, 1(1):1, 2018.