Evanesco: Architectural Support for Efficient Data Sanitization in Modern Flash-Based Storage Systems

Myungsuk Kim*
morssola75@davinci.snu.ac.kr
Seoul National University

Jisung Park*
jisung.park@inf.ethz.ch
ETH Zürich & Seoul National University

Geonhee Cho
ghcho@davinci.snu.ac.kr
Seoul National University

Yoona Kim
yoonakim@davinci.snu.ac.kr
Seoul National University

Lois Orosa
lois.orosa@inf.ethz.ch
ETH Zürich

Onur Mutlu
omutlu@gmail.com
ETH Zürich

Jihong Kim
jihong@davinci.snu.ac.kr
Seoul National University

Abstract

As data privacy and security rapidly become key requirements, securely erasing data from a storage system becomes as important as reliably storing data in the system. Unfortunately, in modern flash-based storage systems, it is challenging to irrecoverably erase (i.e., sanitize) a file without large performance or reliability penalties. In this paper, we propose Evanesco, a new data sanitization technique specifically designed for high-density 3D NAND flash memory. Unlike existing techniques that physically destroy stored data, Evanesco provides data sanitization by blocking access to stored data. By exploiting existing spare flash cells in the flash memory chip, Evanesco efficiently supports two new flash *lock* commands (pLock and bLock) that disable access to deleted data at both page and block granularities. Since the locked page (or block) can be unlocked only after its data is erased, Evanesco provides a strong security guarantee even against an advanced threat model. To evaluate our technique, we build SecureSSD, an Evanesco-enabled emulated flash storage system. Our experimental results show that SecureSSD can effectively support data sanitization with a small performance overhead and no reliability degradation.

CCS Concepts. • **Hardware → External storage**; • **Security and privacy → Data anonymization and sanitization**.

Keywords. solid-state drives (SSDs), 3D NAND flash memory, security, privacy, data sanitization

*The first two authors contributed equally to this research.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378490

ACM Reference Format:
Myungsuk Kim, Jisung Park, Geonhee Cho, Yoona Kim, Lois Orosa, Onur Mutlu, and Jihong Kim. Evanesco: Architectural Support for Efficient Data Sanitization in Modern Flash-Based Storage Systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378490

1 Introduction

Securely erasing data when necessary is becoming one of the essential functions in a modern storage system as the amount of security-sensitive data processed by computer systems is rapidly increasing. For example, when sensitive personal data, such as private photos, social-networking-service messages, or confidential data (e.g., medical records or proprietary documents), is *intentionally* deleted, the deleted data should *never be recoverable*. However, modern file systems delete a file by *unlinking* it (i.e., changing the metadata to indicate that the space occupied by the file is empty), but they do not physically delete the content of the file in the storage system [1–3]. This is undesirable because, if one bypasses the file system to access the storage media directly, such insecurely erased data can be accessed, resulting in unauthorized disclosure of security-sensitive information.

To prevent such unauthorized information disclosure, this paper addresses the data-sanitization problem in flash-based storage systems. Formally, we define that a storage system S supports data sanitization for a set F of files if the following two conditions are satisfied for a file $f \in F$: (C1) S does not store any content of file f after file f is deleted, and (C2) S does not keep any old content of file f after file f is updated. Our main goal is to investigate how to design a flash-based storage system that can efficiently and reliably support secure data sanitization.

Unlike in a hard disk drive (HDD), data sanitization is quite difficult to support in a modern flash-based storage system due to the erase-before-write nature of flash memory. This property prevents NAND flash memory from supporting in-place updates to stored data. For example, in an HDD, overwrite-based techniques [4, 5] that overwrite the logical

sectors of a file are sufficient to erase the file in an irrecoverable fashion. On the other hand, in a flash-memory-based storage system, a simple overwrite-based data sanitization approach does not work. When the host system overwrites a file, the storage firmware writes the new data of the file in new *physical pages*. The old data remains until the storage firmware erases the block that includes the physical pages containing the old data of the file.[1] The uncontrolled presence of old data of deleted or updated files opens up ample opportunities for improper access to sensitive data. In this paper, we call this issue the *data versioning problem*.

As an alternative to the overwrite-based technique, which is not applicable to flash-based storage systems, prior works propose physical-sanitization techniques [6–11]. For example, erase-based techniques [6, 7] sanitize a file by erasing all the *physical pages* (not just the logical pages at the file-system level) that belong to the file. However, such an approach is difficult to adopt in practice due to its high performance overhead. Since an erase operation works at block granularity (which consists of hundreds of pages[1]), to erase a page of a block, all the valid pages in the same block should be moved to different block(s) before the entire block is erased, which incurs significant performance and lifetime overheads in a flash-based storage system. Moreover, a block of modern 3D NAND flash memory should be erased *lazily* due to reliability issues (see Section 5.4). That is, a block cannot be erased immediately when one or more of its pages need to be erased, which makes existing erase-based techniques difficult to employ in modern 3D NAND flash memory.

Reprogram-based techniques [7–11] aim to overcome the large performance overhead of erase-based techniques. These techniques reprogram (i.e., overwrite) the previously-written page to effectively destroy the old data in the page. Although reprogram-based techniques enable erase-free data sanitization, they have a similar performance problem as erase-based techniques when used for a flash-based storage system adopting *multi-level cell (MLC)* flash memory. Since multiple pages share the same physical flash cells (i.e., the same wordline) in MLC flash memory, when a single page p of a wordline w is to be erased, all the other valid pages associated with wordline w should be moved to a different wordline before page p is reprogrammed. Moreover, as the MLC technique advances to support more bits per cell using more advanced manufacturing technology (e.g., triple-level cell (TLC) [12, 13] or quad-level cell (QLC) [14, 15] flash memory), reprogram operations quickly degrade the reliability of flash memory as the cell-to-cell interference caused by them drastically increases [16–21]. For future MLC flash memory, frequent reprogram operations may be difficult to use in practice because their interference with neighboring wordlines cannot be easily controlled to meet the high-reliability requirement of flash memory.

In this paper, we advocate a new type of data-sanitization technique for modern flash-based storage systems. Motivated by the observation that physically destroying stored data has undesirable impact on the reliability of MLC flash memory, we focus on alternative data-sanitization mechanisms that provide security guarantees equivalent to the ones that physically erase sanitized data. We propose *Evanesco*,[2] a new data-sanitization technique that disables access to sanitized data. Evanesco avoids physically destroying the contents of sanitized data by implementing a blocking mechanism *within a flash chip*. This on-chip access control makes Evanesco secure even against a very strong attacker that can directly access raw flash chips (i.e., bypassing OS- or firmware-level access control) through all interfaces to access the flash chip.

Evanesco uses two new flash commands to block access to sanitized data: 1) `pageLock` (`pLock`) disables access to a page, and 2) `blockLock` (`bLock`) disables access to an entire block. With our proposed blocking mechanism, a read request to a sanitized (i.e., locked) flash page always returns data with all bits set to '0'. Since locked pages cannot be unlocked except after the corresponding block (where the locked pages exist) is physically erased, Evanesco provides security guarantees against advanced threat models (see Section 5). By exploiting spare flash cells that exist in modern NAND flash memory, Evanesco reliably manages the access permissions to each page and block inside the flash chip. Our evaluations using 160 state-of-the-art 3D TLC NAND flash chips show that Evanesco physically locks a page or a block without negatively affecting the reliability of other stored data.

To demonstrate the effectiveness of Evanesco, we build SecureSSD, an emulation-based prototype solid-state drive (SSD) that implements an extended flash memory model with `pLock` and `bLock`. SecureSSD guarantees that, as soon as a file is deleted, its data becomes physically locked in the flash chip such that accessing the deleted file is not possible. To satisfy the two data-sanitization requirements C1 and C2, the flash translation layer (FTL) of SecureSSD uses `pLock` and `bLock` when 1) a file is to be securely deleted and 2) when a flash management task (e.g., garbage collection (GC)) needs to move a valid page. SecureSSD uses new I/O interfaces to communicate with the host system so that a user can specify the security requirements of the stored data. By using `pLock` and `bLock` only for security-sensitive data, SecureSSD keeps the performance overhead of data sanitization at minimum.

We evaluate SecureSSD using four workloads collected from enterprise servers and mobile systems. Our experimental results show that SecureSSD meets the data-sanitization requirements while achieving high SSD efficiency. Over existing reprogram-based techniques, SecureSSD increases the

[1] In NAND flash memory, a page is the unit of read and write operations while a block, which consists of hundreds of pages (e.g., 576 pages), is the unit of erase operations. For more detail, see Section 2.1.

[2] Evanesco is a transfiguration spell used to *vanish* an object in the novel series Harry Potter [22].

input/output operations per second (IOPS) performance by up to 4.8× (2.9× on average) and reduces the number of block erasures by up to 79% (62% on average).

This paper makes the following key contributions:

- We propose Evanesco, a new low-cost data-sanitization technique for modern 3D NAND flash memory. Evanesco effectively sanitizes security-sensitive data by disabling access to the sanitized data until the corresponding block is physically erased. By using spare flash cells, Evanesco reliably manages the data-access permission inside a flash chip without negative effect on the performance and reliability of a storage system.
- We introduce SecureSSD, an Evanesco-enabled storage system that implements Evanesco at low performance overhead. By allowing a user to set the security requirements of written data with extended I/O interfaces, SecureSSD selectively sanitizes *only* security-sensitive data.
- We experimentally evaluate the reliability, performance, and lifetime of SecureSSD using 160 state-of-the art 3D TLC NAND flash chips. Our evaluations show that SecureSSD quickly sanitizes a page or a block without negatively affecting the reliability of the storage system. We compare SecureSSD to existing physical-sanitization techniques and show that SecureSSD significantly reduces the performance and lifetime overheads of data sanitization.

2 Background

To provide the necessary background for understanding how Evanesco works, we describe the basics of NAND flash memory, and we give an overview of flash-based storage systems.

2.1 Basics of NAND Flash Memory

NAND flash memory consists of flash cells, which store data, and peripheral circuits, which support flash commands such as read and write. A group of flash cells (typically 8K-16K cells) form a wordline (WL) and multiple WLs (typically 128-256 WLs) form a block. Figure 1 illustrates a typical flash block organization. In this block, there are n WLs and each WL consists of m flash cells. The flash cells on the same WL_k, which share the common WL_k signal from the row decoder module, are read or programmed together as a unit. A WL stores as many pages as the number of bits represented by a single flash cell (up to four pages with state-of-the-art QLC technology). All flash cells along the same column are connected in series to form a bitline (BL). BLs, which are shared by all the blocks in a flash chip, are connected to the page buffer, which is used for off-chip data transfer through the data-in/out circuitry. Two select transistors at the top and bottom of a BL comprise the source select line (SSL) and the ground select line (GSL) of a block, respectively. By applying proper voltages to the SSL and GSL of a block, we can activate the block to perform flash operations.

Although the flash cell, shown in Figure 1, is structurally similar to a normal MOS transistor, it is unique in that its

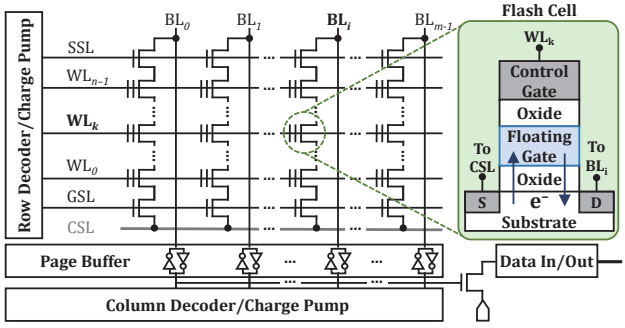


Figure 1. Organizational overview of a NAND flash block.

threshold voltage V_{th} can be adjusted by injecting (ejecting) electrons into (out of) the floating gate. Depending on the number of electrons in the floating gate, the flash cell works as an *off* or *on* switch under a given control gate voltage, thereby effectively storing data (encoded by V_{th}). For example, in single-level cell (SLC) flash memory, we can assign '0' state (i.e., data value '0') to high V_{th} in the flash cell, and '1' state (i.e., data value '1') to low V_{th} in the flash cell.

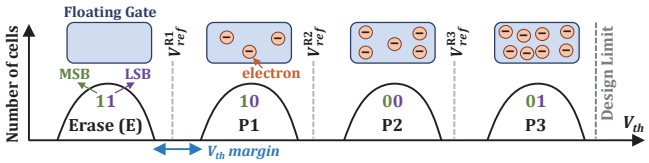
Flash Operations. The *program operation* changes the data value of a flash cell by increasing the cell's V_{th} (i.e., program operation can only change the data value from '1' to '0', assuming the SLC encoding just described). To increase the V_{th} of a flash cell, the program operation transfers electrons from the substrate into the floating gate of the cell via FN tunneling [23] by applying a high voltage (> 20V) to the WL. The *erase operation* sets the data value of a flash cell back to '1'. The erase operation applies a high voltage (> 20V) to the substrate (while all the WLs in the block are set to 0V) to remove electrons from the floating gate, which decreases the V_{th} of all flash cells in a block. Since the program operation can change the data value of a flash cell only from '1' to '0', all the flash cells of a page should be erased first to program data on the page (called *erase-before-program*). The erase operation works at block granularity because a high voltage is applied to the substrate that underlies the entire block.

To read the stored data from a page, the V_{th} levels of the flash cells on the WL are probed using a read reference voltage V_{ref}. In Figure 1, when WL_k is selected for read,[3] if the V_{th} of the i-th flash cell in WL_k is higher than V_{ref}, the i-th flash cell turns *off*, so the cell current of BL_i is blocked (i.e., the flash cell is identified as '0'). On the other hand, if the V_{th} of the i-th flash cell is lower than V_{ref}, the i-th flash cell turns *on*, so the cell current can flow through BL_i (i.e., the flash cell is identified as '1'). By sensing BLs from the selected WL_k, the stored data in the entire WL is read into the page buffer. Note that, if we can prevent the page buffer from buffering the data of a flash cell, or prevent the data in the page buffer from being transferred out of the flash chip

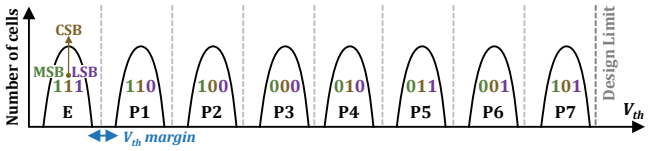
[3] Since no other WL in the same block (e.g., WL_{k-1} or WL_{k+1}) should affect the read operation on WL_k, the flash cells in all other WLs should behave like pass transistors. Their gate voltages are set to V_{READ} (> 6V), which is much higher than the highest V_{th} value of any flash cell [24].

via the data-out path, we would prevent access to the stored data in any WL.

Multi-level Cell Flash Memory. To improve the storage density of flash memory, the multi-level cell (MLC) technique is widely used. MLC technology was initially developed to store 2 bits per cell [25, 26], then extended to support 3 bits/cell (called TLC) [12, 13] and 4 bits/cell (called QLC) [14, 15]. Figure 2 illustrates V_{th} distributions for 2^m-state NAND flash memory that stores m bits within a single flash cell by using 2^m distinct V_{th} states, for $m = 2$ (MLC) and $m = 3$ (TLC). As m increases to store more bits in a flash cell, more V_{th} states should be squeezed into a limited V_{th} window, which is fixed at flash design time. Therefore, more careful management (e.g., smaller ISPP voltage steps [27]) is required for multi-level flash memory to form finer V_{th} states. Furthermore, as m increases, the V_{th} margin (i.e., the gap between two neighboring V_{th} states) inevitably becomes smaller, as shown in Figures 2(a) and 2(b). When the V_{th} margin is smaller, the V_{th} distributions of two neighboring states are more likely to overlap under various noise conditions (e.g., long retention times [28–31], cell-to-cell interference [16–21], and read disturbance [24, 32]), which degrades the reliability of NAND flash memory. For example, MLC NAND flash memory can tolerate up to 3,000 program and erase (P/E) cycles, while TLC NAND flash memory can tolerate only about 1,000 P/E cycles [33]. As a result, as m increases, many optimization techniques that are effective for smaller m's become less effective or even inapplicable due to the lower flash reliability. For a detailed review of NAND flash reliability and management techniques, we refer the reader to Cai *et al.* [34–36].



(a) m=2 : MLC flash memory.



(b) m=3 : TLC flash memory.

Figure 2. V_{th} distributions of 2^m-state NAND flash memory.

2.2 Overview of Flash-Based Storage Systems

A flash-based storage system employs special embedded firmware, called flash translation layer (FTL), which plays a key role in managing unique features and characteristics of flash memory and providing backward compatibility with traditional HDDs (which support overwrites). The FTL handles host writes in an append-only fashion (i.e., it always stores the new data of a logical page using a new physical page) to avoid a long-latency erase operation before a program operation on a page (due to the erase-before-program nature of flash memory). In order to link logical pages (from the operating system) to physical pages of the flash-based storage system, the FTL maintains a logical-to-physical (L2P) mapping table. Figure 3 shows an example of how the FTL manages a page write from the host system. When the file system overwrites logical page address (LPA) `0x00` whose current value is `A` (❶), the FTL stores `A'` at free physical page address (PPA) `0x03` (❷). The FTL then updates the L2P mapping table as well as the page status table: the L2P mapping of LPA `0x00` to PPA `0x03` and the valid status for PPA `0x03` (❸). Finally, the FTL *invalidates* the old PPA `0x00` that used to be mapped to LPA `0x00` (❹).

To maintain a sufficient number of free pages for future host writes, the FTL invokes a garbage collection (GC) process when the storage system is close to running out of free pages. The GC process reclaims free pages by erasing *victim* blocks. If a victim block contains valid pages, these valid pages should be moved to other block(s) and remapped in the L2P table before the victim block is erased. To reduce such live-data-copy overheads in the GC process, the host system uses a new block I/O command, called trim or discard [37]. Figure 3 shows an example of how a block is deleted. When the file system deletes LPA `0x01`, it issues a trim request to inform the storage device that the LPA's data is no longer necessary (①). The FTL then changes the related L2P mapping and page status (②) so that the GC process becomes aware that PPA `0x01` is no longer valid. When Block 0 is selected as a victim block for GC, only `A'` at PPA `0x03` needs to be copied to some other block (e.g., Block 1 in Figure 3).

The append-only strategy is an inevitable choice for an FTL to efficiently manage the underlying flash chips, but it renders a flash-based storage system vulnerable to potentially-malicious access to deleted or stale data. As shown in Figure 3, when the file system updates (❶~❹) or deletes (①~②) a file, the FTL *logically* invalidates the corresponding pages while leaving the physical data of the invalidated pages intact. Only the GC process *physically* erases the invalid pages.[4] Thus, a security vulnerability in a flash-based storage system arises from the temporal gap between when a page is *logically*

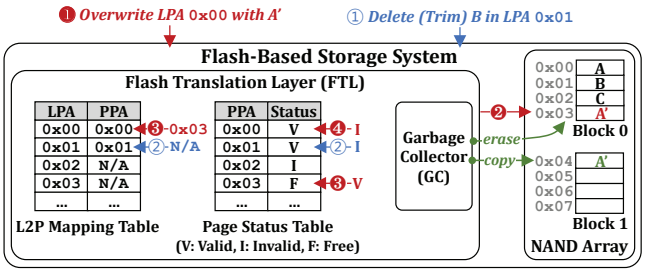


Figure 3. An illustrative example of how an FTL works.

[4]Note that, the host system cannot control either which block will be erased or when the block will be erased, with standard I/O interfaces.

invalidated and when the page is *physically* erased. An adversary that accesses raw flash chips before the invalidated page is physically erased can recover deleted or stale file data with a proper forensic tool [38].

3 Data Versioning: An Empirical Study

When a file system deletes or updates a file, multiple versions of old data of the file can remain in the storage system as the FTL always writes new data of the file in new physical pages. In this paper, we call this issue the *data versioning problem*. To better understand the data versioning problem in a flash-based storage system, we empirically measure how many invalid versions of a file exist in a flash-based storage system throughout the lifetime of the file under varying storage workload characteristics.

Version Trace Tool. We use a custom I/O tracing environment, VerTrace, that integrates an existing I/O profiling tool, IOPro [39], and an open storage emulation platform, FlashBench [40]. VerTrace annotates each physical page with 1) the name of the file to which it belongs and 2) the creation time of the file. VerTrace uses the MD5 hash function [41] to efficiently manage per-page annotation information, which is passed to the emulated storage model of FlashBench via an extended block I/O interface. We extend the emulated storage model of FlashBench to support a logger module that keeps track of the number $N^{page}_{valid}(f,t)$ of valid pages and the number $N^{page}_{invalid}(f,t)$ of invalid pages for a file f at time t. VerTrace works with the ext4 file system [42].

Benchmark Traces and Settings. We use three benchmark traces, Mobile, MailServer, and DBServer, each of which mimics the I/O activity in an Android smartphone, a mail server, and a database server, respectively.[5] For faster evaluation, we limit the maximum capacity of the emulated storage to 16 GiB. To avoid potential start-up bias of simulations and focus on steady-state behavior, each evaluation runs until the total written data size exceeds 64 GiB after we initially fill 75% of the storage capacity.

Metrics. The main goal of our evaluation is to identify 1) how many invalid versions of a file exist, and 2) for how long these invalid versions remain inside the storage device. To evaluate the impact of the file access pattern on data versioning, we classify files into two types depending on their write patterns. We call a file f a *uni-version (UV)* file if the snapshot (i.e., contents) of f at time t is a subset of the snapshot of f at time $(t+1)$. For example, if f is an append-only file or a write-once file, f is a UV file. If f is not a UV file, we call f a *multi-version (MV)* file. For example, if the file system deletes or overwrites f, f is not a UV file.

To quantize the data versioning behavior of a file, we use two metrics. First, we define the *version amplification factor (VAF)* of a file f as follows:

$$VAF(f) = \max_{t \in I}\{N^{page}_{invalid}(f,t)\} / \max_{t \in I}\{N^{page}_{valid}(f,t)\}$$

where I represents the entire execution time of the workload. The higher the $VAF(f)$ of a file f, the higher the number of invalid versions are present in the flash chips, which makes the storage system more vulnerable to malicious access. Intuitively, the $VAF(f)$ of a UV file f would be '0'. If f is an MV file, its $VAF(f)$ significantly varies depending on the access pattern of f, which reveals the amount of updated or deleted data of f remaining stale inside flash chips.

Second, we measure the total length $T_{insecure}(f)$ of insecure time intervals of a file f. We define that a file f is insecure at time t, if $N^{page}_{invalid}(f,t) > 0$. The longer the $T_{insecure}(f)$ of a file f, the more likely an adversary can recover an old version of file f. In a real system, $T_{insecure}(f)$ highly depends not only on the access pattern of f, but also on the system idle time. For example, $T_{insecure}(f)$ may be extremely long if there is a huge time gap between when a user deletes file f and when the user issues a sufficient number of writes to invoke GC (which physically erases the deleted data). Since the system idle time significantly varies depending on the user, which is difficult to model, we use *logical* time that increments by 1 for each 4-KiB host write.

Analysis Results. Table 1 summarizes three interesting observations about the data versioning behavior of the three benchmark traces. We calculate the average and maximum VAF and $T_{insecure}$ values of all the created files in each trace execution. $T_{insecure}$ values are normalized to the total number of writes needed to fill the entire capacity of an SSD.[6]

First, the $VAF(f)$ of a file f can be quite high (e.g., 7.8) when file f is heavily updated, as seen for MV files in DBServer. Even if a file is not deleted, such files with high VAF values can pose security vulnerabilities unless their old versions are properly sanitized.

Second, even uni-version files with no updates can have a large number of invalid versions as seen for Mobile (1.5 VAF value) and MailServer (1.0 VAF value). Since UV files do not update their own data, these invalid versions are the result of the extra copy operations during GC invocations.[7]

Third, $T_{insecure}$ values are quite large in UV files as well as MV files. For example, the average and maximum $T_{insecure}$

Table 1. A summary of our data versioning evaluations.

Workload	Uni-version (UV) files				Multi-version (MV) files			
	$VAF(f)$		$T_{insecure}(f)$		$VAF(f)$		$T_{insecure}(f)$	
	avg.	max.	avg.	max.	avg.	max.	avg.	max
Mobile	0.24	1.5	2.0×10^{-2}	0.43	1.0	2.0	0.41	2.3
MailServer	0.22	1.0	2.1×10^{-2}	1.7	0.93	2.4	0.50	2.5
DBServer	4.8×10^{-3}	0.24	0.52	2.6	3.2	7.8	3.5	3.5

[5] See Section 7 for a detailed description of each trace.

[6] If $T_{insecure}(f) = 1.0$, it indicates that invalid pages of a file f exist while the total capacity of a disk is written.

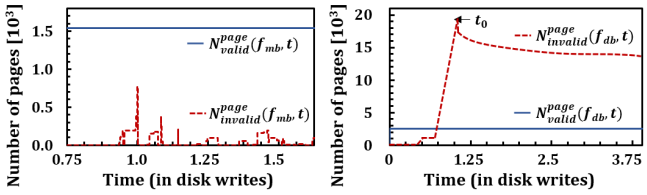
[7] As explained in Section 2.2, the GC process invalidates all the valid pages in a victim block, after copying these valid pages' data to other free pages. Until the victim block is erased, a UV file can have invalid pages stored in the victim block. Since a block is erased lazily (to minimize the negative reliability impact of erase operations; see Section 5.4), it may take a long time for invalid pages of the victim block to be physically erased.

values of MV files in DBServer are 3.5, which indicates that most of the MV files have one or more invalid versions for a very long time (while the host system performs 3.5 disk writes). Note that even UV files are insecure for a significant amount of time (e.g., in MailServer and DBServer) because GC victim blocks are erased lazily as explained in Section 5.4.

To highlight different data versioning patterns, we select two files, f_{mb} (from Mobile) and f_{db} (from DBServer), and compare their $N^{page}_{valid}(f,t)$ and $N^{page}_{invalid}(f,t)$ timeplots. Figure 4(a) shows the timeplot for the append-only file f_{mb}. Even though f_{mb} is a UV file with no updates, there are a fair number of invalid pages (up to 800 pages) due to GC invocations. Figure 4(b) shows the data versioning pattern of the heavily-updated MV file f_{db}. Before the GC process is invoked at time t_0, $N^{page}_{invalid}(f_{db},t)$ rapidly increases due to frequent updates while $N^{page}_{valid}(f_{db},t)$ remains constant. Although $N^{page}_{invalid}(f_{db},t)$ tends to decrease after t_0 because invalid pages are erased by subsequent GC invocations, the rate of decrease in $N^{page}_{invalid}(f_{db},t)$ is quite slow because 1) invalid pages of f_{db} are scattered to many blocks and 2) more invalid pages are generated from continuous updates to f_{db}.

Based on our empirical study, we identify two key requirements that a desired data sanitization technique should meet. First, the technique should support *per-page sanitization*. As mentioned above, the invalid pages of a file (e.g., f_{db} of DBServer) can be stored in the same block with other files' valid pages. If it is not possible to individually sanitize an invalid page, all the valid pages stored in the same block should be copied to other block(s) to sanitize the invalid page, which incurs significant performance and lifetime overheads.

Second, the effect of the page-level sanitization technique should be *immediate*. Our study shows that a single file may have a large number of invalid pages for a long time, even when the host system does not delete the file. If the storage system does not support immediate sanitization of invalid data (i.e., if it sanitizes a file only when the host system deletes the file), the storage system should keep track of all the physical pages used for each file because the FTL uses multiple physical pages to store the data of a single logical page when the file system updates the logical page or the GC process moves the data. Doing so not only requires additional I/O interfaces to send file-system information, but also significantly increases the metadata maintained inside the storage system.



(a) f_{mb}: a UV file in Mobile. **(b)** f_{db}: an MV file in DBServer.

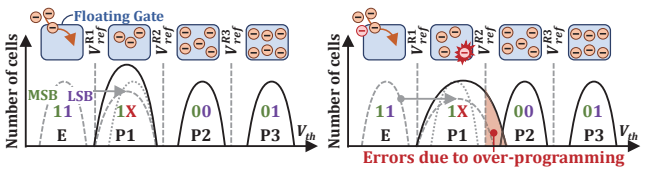
Figure 4. Data versioning under different write patterns.

4 Reprogram-Based Data Sanitization

Most existing data-sanitization techniques (e.g., [6–11]) destroy the stored data by intentionally changing the V_{th} of flash cells. For example, the scrubbing technique [10] increases the V_{th} of all flash cells in a WL so that the V_{th} distributions of different states are mixed together, which makes it impossible to identify the original data. However, in MLC NAND flash memory, this technique is not easy to adopt because it incurs a significant performance overhead to move valid pages out of the WL to be scrubbed. For example, consider TLC NAND flash memory that stores three pages (i.e., LSB (least-significant bit), CSB (central-significant bit), and MSB (most-significant bit) pages) in each WL. To sanitize one of the three pages, other valid page(s) should be moved to other free page(s). To do so, two extra read operations and two extra write operations may be needed.

To overcome the performance overhead of the scrubbing technique, prior work proposes a more efficient reprogram-based sanitization technique for MLC NAND flash memory [8]. Unlike scrubbing, this technique uses the one-shot programming scheme. The key claim of the one-shot reprogramming (OSR) technique is that, even in MLC NAND flash memory, a page can be safely destroyed by using a low program voltage, while the other page in the same WL does not suffer a critical reliability damage. Since no page copy is required during the reprogram process, OSR can achieve zero-copy overhead. Figure 5(a) illustrates an example case where OSR works as intended. In this example, the LSB page is to be sanitized while the MSB page is to remain as a valid page. To destroy the LSB page, OSR moves the V_{th} levels of the E-state cells (i.e., '11') to the right so that they are overlapped with the V_{th} levels of the P1-state cells (i.e., '10'). By doing so, the original LSB page cannot be correctly read with V^{R1}_{ref}, which effectively sanitizes the LSB page. Note that, in this example, the MSB page is not affected at all, and can be reliably read with V^{R2}_{ref}.

Although many flash cells would behave as shown in Figure 5(a), a significant number of flash cells may misbehave under OSR due to over-programming [17, 32, 35, 43]. Figure 5(b) shows such a case where OSR moves the V_{th} levels of the E-state cells too far (to the right) such that some of them overlap with the P2-state cells (i.e., '00'). When such over-programming errors occur, the MSB page cannot be reliably read with V^{R2}_{ref}, because the MSB values of over-programmed cells (which should be '1') are recognized as '0'.

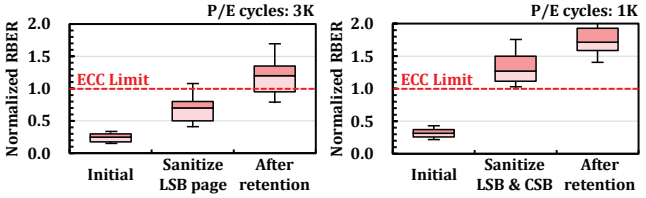


(a) Intended operation. **(b)** Overprogramming.

Figure 5. Two cases under the OSR technique.

To identify how the reliability of multi-level cell NAND flash memory is affected by over-programming errors, we measure how *raw bit-error rate (RBER)* changes under OSR in real 3D MLC NAND flash memory and 3D TLC NAND flash memory chips. Figures 6(a) and 6(b) show RBER per 8,192 of flash cells on MSB pages i MLC and TLC NAND flash memory, respectively, under three different conditions: 1) right after (i.e., zero retention time) programming all the pages on a WL (left-most box plot), 2) right after sanitizing the other page(s) on the same WL using OSR (middle box plot), and 3) after a 1-year retention time (right-most box plot).[8] In MLC NAND flash memory, as shown in Figure 6(a), after the LSB page is sanitized by the OSR, 7.4% of the RBER values in MSB pages exceed the ECC limit, making these *valid* MSB pages *unreadable*. As explained in Figure 5(b), a large portion of the extra bit errors on MSB pages are over-programming errors due to an excessive V_{th} shift during OSR. To minimize over-programming errors, one solution might be to fine-tune the OSR parameters separately for each WL. However, since the exact amount of V_{th} shift under OSR can significantly vary depending on each WL's process-variation related characteristics (e.g., the physical location of each WL on the chip) [28, 30, 31, 44, 45], customizing the OSR parameters separately for each WL is extremely difficult. Figure 6(b) shows that the impact of OSR on the reliability of TLC NAND flash memory is even higher than that in MLC because TLC NAND flash memory has a narrower V_{th} margin between adjacent V_{th} states. For example, when both the LSB page and the CSB page are sanitized, *all* of the MSB pages become unreadable due to their high RBER values.

We also observe that the RBER value of a valid page after sanitizing other pages on the same WL greatly increases if the valid page experiences a long retention time. When we measure RBER with the industry standard requirement (i.e., 1-year retention requirement at 30°C [46]), most of the MSB pages in 3D MLC NAND flash memory and all of the MSB pages in 3D TLC NAND flash memory, cannot be reliably read. As shown in the right-most box plots in Figures 6(a) and (b), such valid pages' RBER values can be more than 1.5 times over the ECC limit (correction capability). Our evaluation results, therefore, clearly show that OSR is not a reliable solution for data sanitization in modern flash-based storage systems, since it leads to destruction of valid data that is not supposed to be sanitized.



(a) MLC NAND flash memory. **(b)** TLC NAND flash memory.

Figure 6. Changes in RBER of MSB pages under OSR.

[8] All measurements are normalized to the maximum RBER value below which an ECC module can correct errors.

5 Evanesco: Lock-Based Sanitization

We propose Evanesco, a new technique to sanitize a physical page *immediately* without negatively affecting the reliability of other stored data. In this section, we present our threat model and introduce two new flash commands, `pLock` and `bLock`, which enable Evanesco to sanitize a page and a block, respectively, at low cost.

5.1 Threat Model

We assume a very capable attacker who possesses all the required skills to recover deleted information from a modern flash-based storage system by reading flash cells through the interfaces to the NAND flash chip. The attacker can gain physical access to a full system, including a processor, DRAM, and a flash-based storage device. The attacker can deconstruct the storage device (e.g., de-soldering flash chips) without any damage on stored data, and directly access the raw flash chips through all known flash interface commands while bypassing the file system and the FTL.

If the storage system is encrypted, the attacker can obtain any necessary passwords and encryption keys to decrypt stored data. The attacker has comprehensive knowledge about the implemented encryption scheme, and can perform sophisticated attacks (e.g., a cold boot attack [47]) to retrieve the secret keys. The attacker can issue a court order or legal subpoena that obliges a user to reveal the used password.

We assume that the attacker cannot directly probe raw flash memory cells to retrieve stored data using highly-sophisticated tools such as a scanning electron microscope (SEM) [48]. Although such attempts were successful in very early 2D SLC NAND flash memory (with 350-nm technology node) [48], to our knowledge, they are very difficult in practice for modern 3D NAND flash memory due to several reasons. First, in order to have visual access to flash memory cells, the attacker needs to deprocess several tens of layers of flash chips. Since memory cells are organized in a cubic form in 3D NAND flash memory, it likely requires extreme effort to expose individual cells without damaging their electrical status (i.e., stored data). Second, the technology node of modern 3D NAND flash memory already has reached 20 nm [49], and manufacturers employ aggressive multi-level cell techniques (e.g., TLC and QLC NAND flash memory) to maximize storage density. To directly read stored data from such memory cells, an SEM should support an extremely high resolution to distinguish the contrast between cells with different values. We are not aware of any demonstration that allows an attacker to directly probe raw memory cells in modern 3D NAND flash memory. We speculate that, if such techniques exist, they require extremely expensive equipment and infrastructure.

5.2 Approach Overview

The key insight of Evanesco is that if we could block access to a flash page by controlling the on-chip access permission (AP) flag of the page, we could achieve the same effect of data sanitization without physically destroying the data stored in the page. In Evanesco, we support two types of AP flags inside a flash chip, a page-level AP (pAP) flag and a block-level AP (bAP) flag, controlled by two new flash commands, pageLock (pLock) and blockLock (bLock), respectively. The pLock <ppn> command locks physical page number ppn by setting the pAP flag of ppn to the *disabled* state. The bLock <pbn> command locks physical block number pbn by setting the bAP flag of pbn to the *disabled* state. When a page or a block of a flash chip is locked by pLock or bLock, respectively, the flash chip blocks any access to it. Since the pAP flag (or bAP flag) of the locked page (or locked block) can be reset to its default *enabled* state only after the block with the locked page (or the locked block itself) is erased (i.e., no *unlock* command exists for locked pages or blocks), once a page or a block is locked, its data becomes permanently inaccessible. When the locked page or block is re-enabled, its data already has been destroyed by an erase operation.

Figures 7(a) and 7(b) illustrate an operational overview of pLock and bLock, respectively. To sanitize physical page 0x22 (denoted as PP#0x22), the pLock <0x22> command (❶) sets the pAP flag of PP#0x22 to disabled ('D' in ❷). Future reads to PP#0x22 (❸) fail because the Evanesco-enhanced logic inside the flash chip checks if the pAP flag is enabled (❹) before transferring the page data out from a flash chip (❺). If a target page's pAP flag is enabled (e.g., as for PP#0x20), a read request to the page operates as a normal read operation. Similarly, as shown in Figure 7(b), when physical block 0x08 (denoted as PB#0x08) needs to be sanitized, the bLock <0x08> command (①) sets the bAP flag of PB#0x08 to disabled ('D' in ②). When the bAP flag is disabled, a read to any of the pages in PB#0x08 fails, including a read to PP#0x20 (③), because the Evanesco-enhanced logic first checks the bAP flag (④) before the pAP flag and prevents reading out page data from a disabled block (⑤), regardless of the pAP flag of the target page.

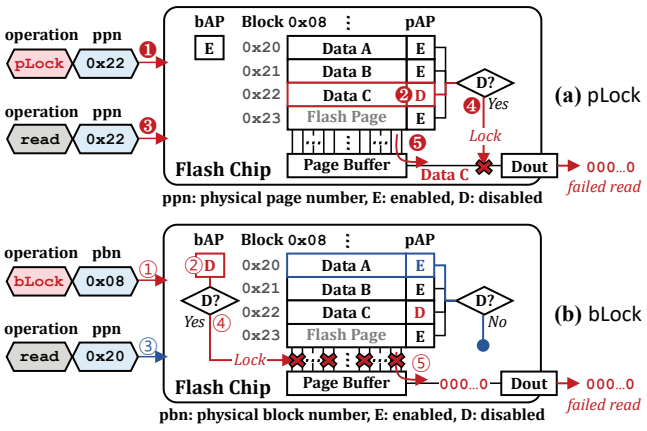
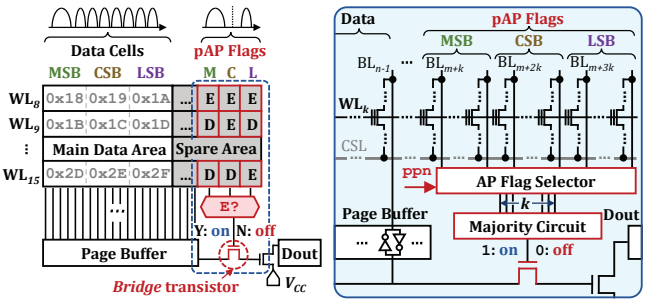


Figure 7. Operational overview of pLock and bLock.

5.3 PLock: Page-Level Data Sanitization

Organizational Overview. Figure 8(a) shows an organizational overview of our pLock implementation. In order to exploit the existing flash organization as much as possible, per-page pAP flags are implemented using flash cells available in the spare area[9] (i.e., the OOB (out-of-band) area) of each WL. For example, in the TLC flash memory illustrated in Figure 8(a), three pAP flags are placed in the spare area for the LSB, CSB, and MSB pages of each WL, respectively. Since the spare area is read concurrently with the main data area (i.e., a page), no special command is needed to read a pAP flag. If the pAP flag of a page is set to disabled , the bridge transistor, which connects the page buffer to data-out pins as shown in Figure 8(a), is turned off so that the flash chip outputs all-zero data. Otherwise, the flash chip outputs the requested data from the page.

To implement the pAP flags with spare flash cells, it should be possible to *selectively* program flash cells on the same WL because 1) the pAP flag of a page is set to disabled *after* the page is programmed on the main data area and 2) the pAP flag of each page on the same WL is set to disabled at different times. To support such selective cell programming, we exploit the SBPI (self-boost program inhibit) technique [27], which allows flash cells on the same WL to be selectively programmed by choosing different voltage settings for different BLs.[10] For example, when programming PP#0x19, we inhibit all three pAP flags on WL_8 with the SBPI technique so that the pAP flags are not programmed, and thus they stay in the default enabled state. When PP#0x19 needs to be sanitized,



(a) Organizational overview. **(b)** A pAP implementation.

Figure 8. Our pLock implementation.

[9] A flash page consists of a main data area for storing data and a spare area for storing page-specific information such as the logical page address and error-correcting code (ECC) [50] values. A typical 16-KiB page has up to 1 KiB as spare area.

[10] When a page in WL_k is programmed, its i-th cell within the page is selectively programmed depending on the value of BL_i. If BL_i is set to '0' (i.e., 0V), the i-th cell is programmed. If BL_i is set to '1' (i.e., V_{CC}), the i-th cell is not programmed (i.e., it is inhibited) because there is an insufficient voltage difference between the cell's floating gate and the substrate due to the channel boosting effect [27, 32, 43, 51]

pLock sets only the pAP flag of PP#0x19 to disabled by inhibiting all the data cells on WL_8 as well as the pAP flags of PP#0x18 and PP#0x1A.

Implementation. To support pLock with the organization shown in Figure 8(a), there are two main implementation challenges. First, programming a pAP flag should not cause reliability issues (e.g., due to interference) in the main data area and the spare area of the WL. Although the SBPI technique supports selective programming of flash cells on a WL, inhibited cells might be affected by a high program voltage applied to the WL. Second, a pAP flag should be programmed fast and read reliably. In particular, it should be guaranteed that there is no error in pAP flags under all flash operating conditions (e.g., long retention times, high P/E cycles, and process variability [28]). For example, if a disabled pAP flag value is mistakenly re-enabled after a long retention time, the associated locked page can be accessed again, which is unacceptable.

In our current design, we meet the first requirement by programming a pAP flag using the one-shot programming scheme with a lower program voltage, in addition to the SBPI scheme. Since flash cells for pAP flags need to distinguish between only two discrete states enabled and disabled, we treat flash cells for pAP flags as SLC cells. Unlike typical TLC data cells which need to store eight different states, SLC flash cells with two states can be reliably programmed with a low program voltage, avoiding a reliability degradation due to the over-programming problem (Section 4). Furthermore, the one-shot programming scheme reduces the frequency and duration of applying a program voltage to a WL. Therefore, using the SBPI scheme with a low-voltage one-shot programming scheme minimizes the impact of programming a pAP flag on the reliability of the inhibited flash cells. Note that using the one-shot programming scheme also has the benefit of having a relatively short latency.

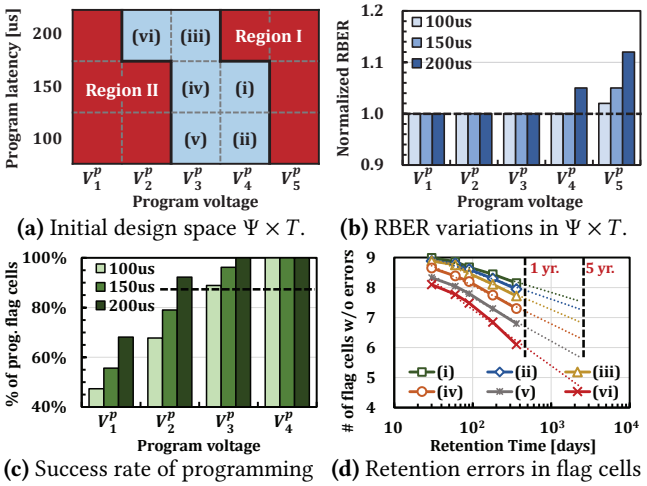
To guarantee error-free management of pAP flags, we employ a simple k-modular redundancy scheme that allocates k flash cells for each pAP flag. As shown in Figure 8(b), the k-bit majority circuit computes the pAP flag value of each page from k flash cells. As we show in the following subsection, with a sufficiently high k, we can manage pAP flags reliably without requiring a complicated ECC module.

Design Space Exploration. To determine good design parameters for our proposed pLock implementation in Figure 8, we conduct comprehensive reliability and performance evaluations using 160 state-of-the-art (48-layer) 3D NAND flash chips. To minimize the potential distortions in the evaluation results, for each test scenario, we evenly select 120 test blocks from each chip at different physical block locations, and test all the WLs in each selected block. We test a total of 3,686,400 WLs (11,059,200 pages) to obtain statistically significant experimental results. Using an in-house custom test board, we evaluate various performance and reliability metrics while varying the number of P/E cycles (from 0 to 1,000) and retention-time requirements (from 0 to 5 year). Due to the page limit, we discuss only the key results under worst-case reliability conditions.[11]

In the design shown in Figure 8, there are three key design parameters that we need to decide: 1) program voltage V_{prog}^{pLock} and 2) program latency t_{pLock} used for programming the flag cells, and 3) k, the number of flash cells per pAP flag. To find the best combination of $(V_{prog}^{pLock}, t_{pLock})$, as shown in Figure 9(a), we start from an initial design space $\Psi \times T$ where $\Psi = \{V_1^p, V_2^p, V_3^p, V_4^p, V_5^p\}$ $(V_{i+1}^p - V_i^p = 0.5V)$ and $T = \{100\mu s, 150\mu s, 200\mu s\}$. We define the initial design space $\Psi \times T$ via a preliminary evaluation on the performance and reliability of the pLock implementation in Figure 8.

First, we evaluate how the reliability of data cells is affected by a different combination of $(V_{prog}^{pLock}, t_{pLock}) \in \Psi \times T$ for the pLock implementation. Although the SBPI technique enables pLock to inhibit data cells while programming flag cells, the RBER of data cells can increase due to *program disturbance*.[12] As shown in Figure 9(b), the higher the program voltage or the longer the program latency, the higher the RBER of data cells due to program disturbance during pLock. Based on the result shown in Figure 9(b), we exclude four combinations in Region I (Figure 9(a)) from further consideration because they increase the RBER of data cells.

Second, we evaluate if a flag cell can be reliably programmed for a $(V_{prog}^{pLock}, t_{pLock})$ combination of the remaining design space (i.e., $\Psi \times T$– Region I). As shown in Figure 9(c), several combinations cannot reliably program a flag flash cell due to low program voltage or short program time. For example, with combination $(V_1^p, 100\mu s)$, pLock can program



(a) Initial design space $\Psi \times T$. **(b)** RBER variations in $\Psi \times T$.

(c) Success rate of programming flag cells. **(d)** Retention errors in flag cells when $k = 9$.

Figure 9. Design space exploration results for pLock.

[11] Our test procedure follows the JEDEC standard [46] recommended for commercial-grade flash products.

[12] Even if all the data cells in a WL are inhibited during pLock, the high program voltage applied to the WL can affect the V_{th} levels of the data cells. This undesired phenomenon is called program disturbance [32, 43, 52].

only 47.3% of flag cells successfully. Based on the result in Figure 9(c), we exclude five combinations in Region II, leaving six candidate combinations, (i) ~ (vi), as shown in Figure 9(a).

As the last step in our design space exploration, we evaluate how the number of retention errors changes when k flag cells are grouped to represent a single pAP flag. We test two retention-time requirements at 30 °C after 1K P/E cycles, 1-year and 5-year retention times, while varying k from 5 to 11. Figure 9(d) shows the evaluation results when k = 9 (which we use as the final k value). The number of retention errors is significantly affected by which (V_{prog}^{pLock}, t_{pLock}) combination we use. For example, for the 5-year retention-time requirement, combination (vi), (V_2^p, 200 μs), leads to 5 retention errors in 9 flag cells, while combination (i), (V_4^p, 150 μs), leads to at most 2 errors. When combined with the 9-bit majority circuit, combination (vi) cannot guarantee that a pAP flag is correctly managed throughout the required retention time. Out of the six candidate combinations, we select combination (ii), (V_4^p, 100 μs), which meets a high retention-time requirement with the shortest t_{pLock}, as the final design parameter along with 9 flag cells to represent a pAP flag.

5.4 BLock: Block-Level Data Sanitization

Need for Block-Level Sanitization. The pLock command enables per-page sanitization at low cost, but its performance overhead may become nontrivial if a large number of pages need to be sanitized at the same time. For example, if a user wants to securely delete a 1-GiB file (e.g., a video file) from a flash-based storage system with 16-KiB page size, 65,536 consecutive pLock commands are needed, which can introduce significant delay in the flash-based storage system. A block-level data sanitization mechanism could mitigate the performance overhead of a large number of pLock commands: a single bLock command can sanitize all the pages in a block at once with low latency.

There is an even more fundamental reason to support such a bLock command in modern 3D NAND flash memory. In recent 3D NAND flash memory, due to structural characteristics, the reliability of data stored in a block strongly depends on the time gap between when the block is erased and when data is programmed to the block [53, 54]. This time gap is called an *open interval*. The shorter the open interval, the more reliable the storage of data. Figure 10 illustrates how RBER increases as the length of an open interval increases. When the open interval is the largest we tracked, RBER is 30% larger than when the open interval is zero. To avoid the reliability problem of an open interval, a block should be erased *lazily*, i.e., the erase of the block should happen just before programming data on the block. Therefore, bLock is essential to effectively sanitize data in an entire block without reliability issues.

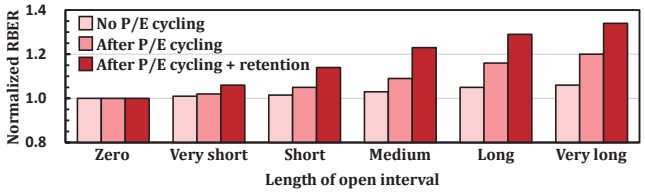
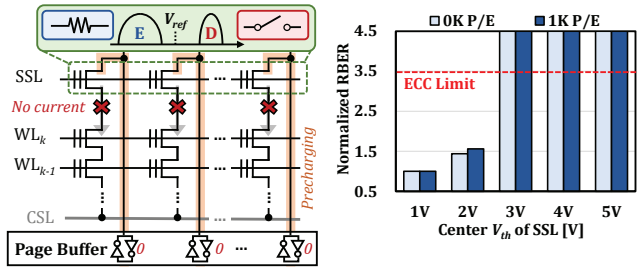


Figure 10. RBER vs. open interval length.

Implementation. We implement bLock by leveraging a new feature of 3D flash organization. We allocate per-block bAP flags in the SSL of each flash block. As explained in Section 2.1, there is an SSL at the top of each block, which is used to select the active block during flash operations. Unlike 2D flash memory where normal transistors are used for SSL transistors, 3D flash memory uses normal flash cells as SSL transistors [25, 55], which allows us to program (and erase) the SSL of a block as a normal WL.[13] Therefore, by sufficiently increasing the V_{th} levels of SSL *cells* (i.e., programming the SSL just like programming a normal WL), we can turn the SSL cells of a block into *off switches*, which effectively inhibit all read requests to the block. Since there is no way to erase only SSL cells using the standard flash interfaces, bLock can efficiently sanitize an entire block.

Figure 11(a) shows the operational overview of our bLock implementation. To disable a block (i.e., to set its bAP to `disabled`), bLock shifts the V_{th} levels of SSL cells to higher than V_{READ}, which effectively disconnects all the flash cells below such SSL cells from the page buffer. Since no current can flow through BLs, the page buffer data for all the flash pages in a block is fixed to '0' irrespective of the actual page data. As shown in Figure 11(b), we find that when the center V_{th} level of an SSL exceeds 3V, a read operation to any of the pages in the corresponding block fails due to the introduction of enough bit errors beyond the correction capability of ECC.



(a) Operational overview. **(b)** RBER vs. center V_{th} of SSL.

Figure 11. Our bLock implementation.

Design Space Exploration. To implement bLock in practice, two requirements should be satisfied. First, bLock should move the V_{th} levels of SSL cells sufficiently so that all SSL cells are completely turned off during a read operation. Second, before physically erasing a flash block, SSL cells should reliably keep their high V_{th} levels during the entire lifetime.

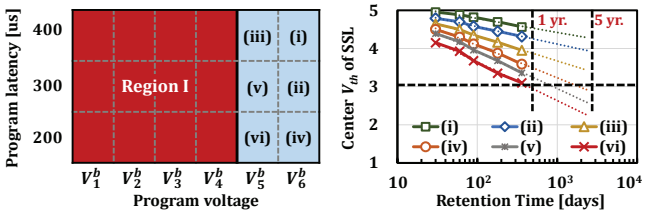
[13] Using a normal flash cell to implement an SSL transistor is inevitable in 3D NAND flash memory because inserting a normal transistor in a vertically-stacked 3D flash organization is more difficult than inserting a flash cell.

Note that, in bLock, we do not need to consider the interference between an SSL and other WLs in the same block because an SSL is already physically separated from other WLs in the same block via a dummy WL (which is inserted between an SSL and WLs) to prevent potential unintentional programming of an SSL during a normal flash operation.

In our bLock implementation, we consider two design parameters related to bAP flags: program voltage V_{prog}^{bLock} and program latency t_{bLock} which are used for programming bAP flags (i.e., SSL cells). To minimize latency, we use the one-shot program scheme for bLock. To find a good combination of (V_{prog}^{bLock}, t_{bLock}), we start from an initial design space $\Psi \times T$ where $\Psi = \{V_1^b, V_2^b, ..., V_6^b\}$ ($V_{i+1}^b - V_i^b = 1.0V$) and $T = \{200\mu s, 300\mu s, 400\mu s\}$, as shown in Figure 12(a).

First, we evaluate if each combination can reliably program an SSL so that the center V_{th} level of the SSL is maintained above 3V with the one-shot programming technique. Based on our evaluation, we exclude the candidate combinations in Region I (in Figure 12(a)) from further consideration because they cannot move the center V_{th} level of SSL transistors to higher than 3V with a desired latency.

Second, we evaluate how the V_{th} levels of SSL cells change under a given retention-time requirement. We test two retention-time requirements at 30 °C after 1K P/E cycles, 1-year and 5- year retention times. Figure 12(b) shows that the center V_{th} level of an SSL significantly vary depending on different (V_{prog}^{pLock}, t_{pLock}) combinations. For example, the center V_{th} level of an SSL programmed with combination (i), (V_6^b, 400μs), is predicted to be more than 4V even after 5 years, while the center V_{th} level of an SSL programmed with combination (vi), (V_5^b, 200μs), is predicted to be lower than 3V before 1 year. Thus, combination (vi) is not reliable (and neither are combinations (iv) and (v)). Considering both t_{bLock} and retention reliability, we select combination (ii), (V_6^b, 300μs), as our final design parameters.



(a) Initial design space $\Psi \times T$. **(b)** Impact of retention time on center V_{th} of SSL.

Figure 12. Design space exploration results for bLock.

5.5 Implementation Overhead

Area Overhead. The proposed pLock implementation requires one 9-bit majority circuit per flash chip in addition to 27 flag cells per each WL. Since we implement the flag cells using the unused flash cells in the spare area of a WL, no space overhead exists for supporting the pAP flags. For the 9-bit majority circuit, approximately 200 transistors are needed [56]. Considering the size of the typical peripheral circuit area in modern flash memory, the area impact of this majority circuit is insignificant. The area overhead of the bridge transistors is also negligible because only one bridge transistor is needed for each data-out path. For example, only 8 bridge transistors are needed for a typical ×8 I/O NAND flash chip.

Latency Overhead. In our implementation, t_{pLock} and t_{bLock} are 100μs and 300μs, respectively. Compared to the page-program latency (t_{PROG}) and block-erasure latency (t_{BERS}), the latency overhead of pLock and bLock is very small. For 3D TLC NAND flash memory, t_{pLock} is less than 14.3% of t_{PROG} (700μs), and t_{bLock} is less than 8.6% of t_{BERS} (3.5ms) [13].

6 SecureSSD: System Integration

In order to take full advantage of pLock and bLock at the system level, we design an Evanesco-enabled flash-based storage system, called SecureSSD, which efficiently supports data sanitization at low cost by interacting with a host computing system. Although pLock and bLock provide low-cost data sanitization at the flash-chip level, it would unnecessarily degrade both SSD and system performance if they are used for sanitizing security-insensitive data. To avoid this, SecureSSD allows the user to specify the *security requirements* of written data through an extended I/O interface, so that the Evanesco-aware FTL in SecureSSD uses pLock and bLock only when invalidating security-sensitive data.

Figure 13 shows how SecureSSD manages written data according to the data's security requirements. To support data sanitization for Evanesco-*unaware* systems in a backward compatible manner, SecureSSD, by default, treats all written data as security sensitive. When an Evanesco-aware application does *not* require high security for a file (e.g., bar in Figure 13), it opens the file with a new access mode flag O_INSEC. Opening a file with O_INSEC indicates that the file data can have multiple versions in the SSD and deletion is not secure. For a write request to a file opened with the O_INSEC flag, a block I/O request to SecureSSD is flagged with a new

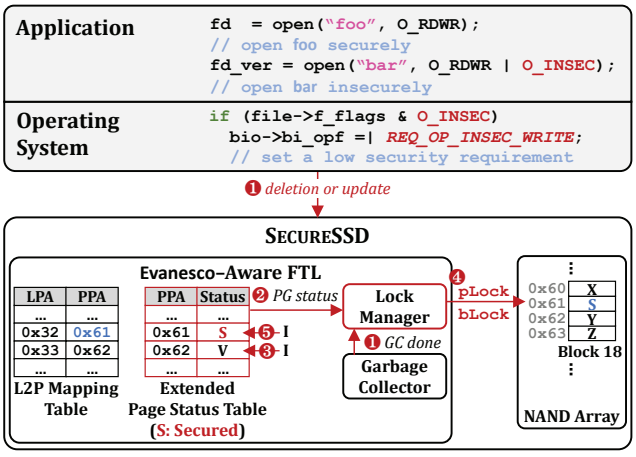


Figure 13. Operational overview of SecureSSD.

operation flag REQ_OP_INSEC_WRITE so that SecureSSD is aware that the written data is security insensitive.

To keep track of the security requirement of each page, the Evanesco-aware FTL in SecureSSD employs an extended page status table and a lock manager. A page in SecureSSD can be in one of four states: free, valid, invalid, or *secured*. For a default write (e.g., a write to LPA 0x32 of foo in Figure 13), the FTL updates the L2P mapping for the requested LPA with a free PPA (e.g., 0x61), and sets the page status to secured. In contrast, for a security-insensitive write (e.g., a write to LPA 0x33 of bar), the FTL sets the page status of the corresponding PPA (e.g., 0x62) to valid instead of secured.

When a PPA needs to be invalidated, e.g., due to a file update/deletion from the host or a copy operation in the GC process (❶ in Figure 13), the lock manager first retrieves the status of the PPA from the extended page status table (❷). If the status of the PPA is not secured, the FTL only updates the status to invalid (❸) as a regular Evanesco-unaware FTL would do. If the status of the PPA is secured, the lock manager *immediately* invokes a pLock or bLock command depending on the status of the other pages in the same block (❹). For example, when sanitizing a single secured page, the lock manager issues a pLock command. On the other hand, when 1) all the remaining pages in a block need to be sanitized (e.g., during GC or due to a trim request to contiguous secured pages) and 2) the estimated latency for sanitizing the pages with pLock is longer than t_{bLock}, the lock manager issues a bLock command to minimize the performance overhead due to data sanitization. After that, the FTL updates the status of the securely-invalidated page(s) to invalid (❺).

7 System-Level Evaluation

Methodology. We implement SecureSSD on FlashBench [40] with an Evanesco-enabled emulated flash model. Although FlashBench supports up to 512-GiB capacity, we limit its SSD capacity to 32 GiB for fast evaluation. We configure SecureSSD with two channels, each of which has four 3D TLC NAND flash chips. Each chip has 428 blocks and each block has 576 16-KiB pages (i.e., 192 WLs). We set flash operation timing parameters for t_{READ}, t_{PROG}, and t_{BERS} to 80μs, 700μs, and 3.5ms, respectively. Based on our design space exploration results, we set t_{pLock} and t_{bLock} to 100μs and 300μs, respectively.

We use four different benchmark traces. Three traces, MailServer, DBServer, and FileServer, are generated with the Filebench benchmark tool [57]. One trace, Mobile, is collected from an Android smartphone (Samsung Galaxy S2 [58]). Table 2 summarizes three main I/O characteristics of the evaluated benchmarks: 1) read to write ratio, 2) write pattern, and 3) write size. We use a custom trace replayer that sends each write request to an SSD with its security requirements and aligns the data boundary of each write request to multiples of 16 KiB (i.e., the physical page size).

Table 2. I/O characteristics of our four benchmarks.

Benchmark	read:write	File write pattern	Write size
MailServer	1:1	create/append/delete e-mails	16–32 KiB
DBServer	1:10	overwrite data files and log files	16–256 KiB
FileServer	3:4	create/append/delete files	32–128 KiB
Mobile	1:50	create/delete pictures	0.5–8 MiB

We compare SecureSSD (secSSD) with two baseline SSDs, erSSD and scrSSD, which exploit existing physical-sanitization techniques to support *immediate* data sanitization.[14] As with secSSD, erSSD and scrSSD manage write requests in a secure fashion, only if the requests have a high security requirement. When a secured page needs to be invalidated, erSSD *erases* the entire block that contains the secured page[15] while scrSSD performs *scrubbing* on the WL that contains the secured page. When the target block or the target WL has other valid pages, erSSD and scrSSD copy the valid pages to other free pages. In scrSSD, we set the scrubbing latency to 100μs assuming that the one-shot programming scheme is used to minimize performance overhead. To understand the benefits of pLock and bLock, we also evaluate secSSD$_{nobLock}$ which works in the same fashion as secSSD but without bLock.

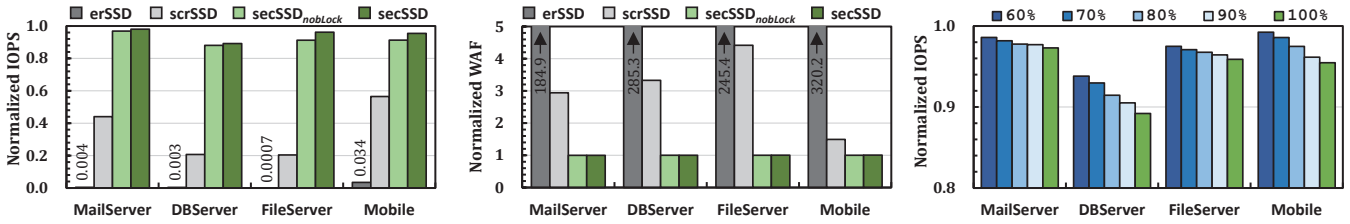
Evaluation Results. To evaluate the performance of SecureSSD, we measure input/output operations per second (IOPS) performance and write amplification factor (WAF) values for each SSD. All values are normalized to ones from an SSD with no data sanitization support. Figure 14(a) compares IOPS values of different SSDs under each workload. SecSSD significantly outperforms erSSD and scrSSD under every workload. erSSD performs poorly, achieving less than 4% of the IOPS level of the baseline SSD. Although scrSSD significantly outperforms erSSD, it achieves only 34% of the performance of the baseline SSD. In contrast, secSSD achieves 94.5% of the performance of the baseline SSD.

The performance gap between secSSD versus erSSD/scrSSD is mainly due to the large number of additional copy operations present in erSSD and scrSSD. As shown in Figure 14(b), the WAF values of erSSD and scrSSD are substantially higher, by up to 320× and 4.41× (251× and 2.6× on average) over the baseline SSD, respectively. In contrast, secSSD achieves almost equivalent WAF as the baseline SSD. Note that the amplified writes in erSSD and scrSSD can greatly degrade the SSD lifetime as well as the IOPS performance due to more frequent GC invocations.

Even though both secSSD$_{nobLock}$ and secSSD can sanitize a page without copying other valid pages stored in the same wordline, secSSD$_{nobLock}$ has lower performance

[14] All three SSDs we evaluate guarantee that $\forall t : N_{invalid}^{page}(f, t) = 0$, for a file f with a high security requirement.

[15] Since we are interested in comparing the I/O performance of erSSD and secSSD, we assume that there is no reliability issue due to the open block problem (see Section 5.4) in erSSD, i.e., erSSD can immediately erase a block without any reliability issue.



(a) IOPS of different SSDs (higher is better). **(b)** WAF of different SSDs (lower is better). **(c)** IOPS vs. secure data fraction.

Figure 14. Performance of SecureSSD under four different workloads.

than secSSD, in particular, under workloads with large-size writes (as in FileServer and Mobile). This is because the more pages are invalidated at the same time, the more opportunities for secSSD to sanitize an entire block by using bLock. We compare the number of pLock operations performed in secSSD and secSSD$_{nobLock}$ under each workload: the results show that the use of bLock operations reduces the number of pLock operations by up to 57% (28% on average) in secSSD. As a result, secSSD further improves the IOPS performance by up to 5.4% (3.1% on average) over secSSD$_{nobLock}$, as shown in Figure 14(a).

Finally, we measure the IOPS performance of secSSD under each workload while varying the fraction of securely-managed data, as shown in Figure 14(c). The fewer the secured pages, the higher the performance secSSD could achieve by using pLock and bLock only when invalidating security-sensitive data. When managing 60% of total written data in a secure fashion, the performance of secSSD is only up to 6.2% (2.8% on average) lower than that of the baseline SSD. Although selective sanitization has a higher performance impact under write-intensive workloads, DBServer and Mobile, secSSD exhibits the lowest performance in DBServer. This is because DBServer issues a large number of small updates to securely managed files such that secSSD has little opportunity to perform bLock and to exploit the internal parallelism of the SSD for pLock operations.

8 Related Work

To our knowledge, Evanesco is the first mechanism that supports low-cost data sanitization by disabling access to data within the flash chip. We briefly discuss closely-related prior work that aims to support data sanitization in flash-based storage systems by destroying or encrypting data.

Physical Destruction of Stored Data. The most fundamental approach to sanitizing data is to physically destroy the data in the storage medium. Diesburg *et al.* [6] propose a framework that enables a file system to notify an SSD to immediately erase blocks containing security-sensitive data. However, such an erase-based approach can significantly degrade the performance and lifetime of SSDs, introducing a large number of data copies. To sanitize data without requiring block erasure, several studies [7, 9–11] propose techniques based on scrubbing, which destroys the V_{th} values of cells in a target page in a WL. Scrubbing a WL in SLC flash memory is relatively simple as there is only one page in a WL. However, for MLC flash memory, scrubbing techniques need to copy other valid pages in the same WL to some other WLs before destroying the V_{th} values of the target page in a WL. To mitigate the performance overhead of scrubbing in MLC flash memory, Lin *et al.* [8] propose the one-shot reprogramming technique that enables the destruction of the V_{th} values of individual pages in a WL separately from each other. However, as shown in Section 4, the one-shot reprogramming technique is not easily applicable to modern 3D NAND flash storage systems, since it cannot meet the reliability requirements due to significant over-programming errors.

Data Encryption. Multiple works use data encryption to implement low-overhead data sanitization techniques for flash-based storage systems [3, 59–61]. The schemes encrypt security-sensitive data with a cryptographic algorithm, such as AES [62], and delete the encryption key when the data needs to be sanitized. Since it is almost impossible to obtain original data without the encryption key, the encrypted data can be effectively destroyed just by deleting the encryption key. However, encryption may not always be desirable due to performance overheads or resource constraints, and it also requires a complicated key management to satisfy stringent security requirements. If the encryption key is compromised [47, 63], this solution becomes ineffective.

Evanesco does not suffer from encryption overheads. It can also be used as a technique that is complementary to data encryption. For example, when encryption keys are mistakenly leaked, Evanesco can still guarantee that security-sensitive data is sanitized.

9 Conclusions

We introduce Evanesco, a new chip-level data sanitization technique for modern flash-based storage systems. Evanesco supports immediate per-page sanitization at low cost with two new flash commands, pLock and bLock, that disable access to a page or a block, respectively, using access control mechanisms implemented on the flash chip. By leveraging spare cells in existing flash memory organization, Evanesco effectively makes sanitized data in a flash chip inaccessible, with only a small resource overhead. Using state-of-the-art

3D NAND flash chips, we validate that pLock and bLock can quickly disable a target page and block without compromising reliability of stored data. To fully exploit pLock and bLock, we design an Evanesco-enabled flash storage system, SecureSSD, which efficiently and securely manages security-sensitive data by interacting with a host system using extended I/O interfaces. Our experimental results show that SecureSSD can delete security-sensitive files immediately and irrecoverably while providing a comparable performance to an SSD with no data sanitization support.

Based on both chip-level and system-level evaluations, we conclude that Evanesco is an effective low overhead data sanitization mechanism for modern flash memory based SSDs. We believe the basic ideas of Evanesco are applicable to other memory technologies (e.g., PCM [64–68], STT-MRAM [69, 70], RRAM [71]) and encourage future work to explore similar mechanisms in emerging memories.

Acknowledgments

We would like to thank anonymous reviewers for the feedback and comments. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-11. The ICT at Seoul National University provided research facilities for this study *(Corresponding Author: Jihong Kim)*.

References

[1] Joel Reardon, David Basin, and Srdjan Capkun. SoK: Secure data deletion. In *S&P*, 2013.

[2] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or death at block-level. In *OSDI*, 2004.

[3] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *USENIX Security*, 2012.

[4] Steven Bauer and Nissanka B. Priyantha. Secure data deletion for Linux file systems. In *USENIX Security*, 2001.

[5] Colin Plumb. shred(1) - Linux man page. https://linux.die.net/man/1/shred.

[6] Sarah Diesburg, Christopher Meyers, Mark Stanovich, Michael Mitchell, Justin Marshall, Julia Gould, and An-I Andy Wang. TrueErase: Per-file secure deletion for the storage data path. In *ACSAC*, 2012.

[7] Kyungmoon Sun, Jongmoo Choi, and Sam H. Noh. Models and design of an adaptive hybrid scheme for secure deletion of data in consumer electronics. *IEEE TCE*, 2008.

[8] Ping-Hesien Lin, Yu-Ming Chang, Yung-Chun Li, Chien-Chung Ho, and Yuan-Hao Chang. Achieving fast sanitization with zero live data copy for MLC flash memory. In *ICCAD*, 2018.

[9] Wei-Chen Wang, Chien-Chung Ho, Yuan-Hao Chang, Tei-Wei Kim, Kuo, and Ping-Hsien Lin. Scrubbing-aware secure deletion for 3-D NAND flash. *IEEE TCAD*, 2018.

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

[11] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. NFPS: Adding undetectable secure deletion to flash translation layer. In *ASIACCS*, 2016.

[12] H. Nitta, T. Kamigaichi, F. Arai, T. Futatsuyama, M. Endo, N. Nishihara, T. Murata, H. Takekida, T. Izumi, K. Uchida, T. Maruyama, I. Kawabata, Y. Suyama, A. Sato, K. Ueno, H. Takeshita, Y. Joko, S. Watanabe, Y. Liu, H. Meguro, A. Kajita, Y. Ozawa, Y. Takeuchi, Hara T., T. Watanabe1, S. sato, H. Tomiie, Y. Kanemaru, R. Shoji, C.H. Lai, M. Nakamichi, K. Owada, T. Ishigaki, G. Hemink, D. Dutta, Y. Dong, C. Chen, G. Liang, M. Higashitani, and J. Lutze. Three bits per cell floating gate NAND flash memory technology for 30nm and beyond. In *IRPS*, 2009.

[13] Woopyo Jeong, Jae-woo Im, Doo-Hyun Kim, Sang-Wan Nam, Dong-Kyo Shim, Myung-Hoon Choi, Hyun-Jun Yoon, Dae-Han Kim, You-Se Kim, Hyun-Wook Park, Dong-Hun Kwak, Sang-Won Park, Seok-Min Yoon, Wook-Ghee Hahn, Jin-Ho Ryu, Sang-Won Shim, Kyung-Tae Kang, Jeong-Don Ihm, In-Mo Kim, Doo-Sub Lee, Ji-Ho Cho, Moo-Sung Kim, Jae-Hoon Jang, Sang-Won Hwang, Dae-Seok Byeon, Hyang-Ja Yang, Kitae Park, Kye-Hyun Kyung, and Jeong-Hyuk Choi. A 128 gb 3b/cell v-nand flash memory with 1 gb/s i/o rate. *IEEE JSSC*, 2015.

[14] Cuong Trinh, Noboru Shibata, Takeshi Nakano, Mikio Ogawa, Jumpei Sato, Yoshikazu Takeyama, Katsuaki Isobe, Binh Le, Farookh Moogat, Nima Mokhlesi, Kenji Kozakai, Patrick Hong, Teruhiko Kamei, Kiyoaki Iwasa, J. Nakai, Takahiro Shimizu, Mitsuaki Honma, Shintaro Sakai, Toshimasa Kawaai, Satoru Hoshi, Jonghak Yuh, Cynthia Hsu, Taiyuan Tseng, Jason Li, Jason Hu, Ming T. Liu, Shahzad Khalid, Junliang Chen, Mitsuyuki Watanabe, Hung-Szu Lin, Junhui Yang, K. McKay, Khanh Nguye, Tuan Pham, Y. Matsuda, K. Nakamura, Kazunori Kanebako, Susumu Yoshikawa, W. Igarashi, A. Inoue, T. Takahashi, Y. Komatsu, C. Suzuki, Kousuke Kanazawa, Masaaki Higashitani, Seungpil Lee, T. Murai, K. Nguyen, James Lan, Sharon Huynh, Mark Murin, Mark Shlick, Menahem Lasser, Raul Cernea, Mehradad Mofidi, K. Schuegarf, and Khandker Quader. A 5.6 MB/s 64Gb 4b/cell NAND flash memory in 43nm CMOS. In *ISSCC*, 2009.

[15] Jung H. Yoon, Ranjana Godse, Gary Tressler, and Hillery Hunter. 3D-NAND scaling and 3D-SCM—implications to enterprise storage. In *Flash Memory Summit*, 2017.

[16] Ki-Tae Park, Myounggon Kang, Doogon Kim, Soon-Wook Hwang, Byung Yong Choi, Yeong-Taek Lee, Changhyun Kim, and Kinam Kim. A zeroing cell-to-cell interference page architecture with temporary LSB storing and parallel MSB program scheme for MLC NAND flash memories. *IEEE JSSC*, 2008.

[17] Yu Cai, Onur Mutlu, Erich F. Haratsch, and Ken Mai. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *ICCD*, 2013.

[18] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis, and modeling. In *DATE*, 2013.

[19] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F. Haratsch. Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques. In *HPCA*, 2017.

[20] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. Enabling accurate and practical online flash channel modeling for modern MLC NAND flash memory. *IEEE JSAC*, 2016.

[21] Jisung Park, Jaeyong Jeong, Sungjin Lee, Youngsun Song, and Jihong Kim. Improving performance and lifetime of NAND storage systems using relaxed program sequence. In *DAC*, 2016.

[22] Vanishing spell. https://harrypotter.fandom.com/wiki/Vanishing_Spell.

[23] J. Maserjian and N. Zamani. Behavior of the Si/SiO2 interface observed by Fowler-Nordheim tunneling. *AIP Journal of Applied Physics*, 1982.

[24] Yu Cai, Yixin Luo, Saugata Ghose, and Onur Mutlu. Read disturb errors in MLC NAND flash memory: Characterization, mitigation, and recovery. In *DSN*, 2015.

[25] Kazushige Kanda, Noboru Shibata, Toshiki Hisada, Katsuaki Isobe, Manabu Sato, Yui Shimizu, Takahiro Shimizu, Takahiro Sugimoto, Tomohiro Kobayashi, Naoaki Kanagawa, Yasuyuki Kajitani, Takeshi Ogawa, Kiyoaki Iwasa, Masatsugu Kojima, Toshihiro Suzuki, Yuya Suzuki, Shintaro Sakai, Tomofumi Fujimura, Yuko Utsunomiya, Toshifumi Hashimoto, Naoki Kobayashi, Yuuki Matsumoto, Satoshi Inoue,

Yoshinao Suzuki, Yasuhiko Honda, Yosuke Kato, Shingo Zaitsu, Hardwell Chibvongodze, Mitsuyuki Watanabe, Hong Ding, Naoki Ookuma, and Ryuji Yamashita. A 19 nm 112.8 mm^2 64 Gb multi-level flash memory with 400 Mbit/sec/pin 1.8 V toggle mode interface. *IEEE JSSC*, 2012.

[26] Ki-Tae Park, Sangwan Nam, Daehan Kim, Pansuk Kwak, Doosub Lee, Yoon-He Choi, Myung-Hoon Choi, Dong-Hun Kwak, Doo-Hyun Kim, Min-Su Kim, Hyun-Wook Park, Sang-Won Shim, Kyung-Min Kang, Sang-Won Park, Kangbin Lee, Hyun-Jun Yoon, Kuihan Ko, Dong-Kyo Shim, Yang-Lo Ahn, Jinho Ryu, Donghyun Kim, Kyunghwa Yun, Joonsoo Kwon, Seunghoon Shin, Dae-Seok Byeon, Kihwan Choi, Jin-Man Han, Kye-Hyun Kyung, Jeong-Hyuk Choi, and Kinam Kim. Three-dimensional 128 Gb MLC vertical NAND flash memory with 24-WL stacked layers and 50 MB/s high-speed programming. *IEEE JSSC*, 2014.

[27] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, Jung-Hyuk Choi, Jang-Rae Kim, and Hyung-Kyu Lim. A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme. *IEEE JSSC*, 1995.

[28] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *DATE*, 2012.

[29] Yu Cai, Yixin Luo, Erich F. Haratsch, Ken Mai, and Onur Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *HPCA*, 2015.

[30] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. Improving 3D NAND flash memory lifetime by tolerating early retention loss and process variation. In *SIGMETRICS*, 2018.

[31] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. HeatWatch: Improving 3D NAND flash memory device reliability by exploiting self-recovery and temperature-awareness. In *HPCA*, 2018.

[32] Rino Micheloni, Luca Crippa, and Alessia Marelli. *Inside NAND flash memories*. Springer, 2010.

[33] Micron. Micron 3D NAND flash memory. https://www.micron.com/-/media/client/global/documents/products/product-flyer/3d_nand_flyer.pdf, 2016.

[34] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrian Cristal, Osman S. Unsal, and Ken Mai. Error analysis and retention-aware error management for NAND flash memory. *Intel Technology Journal*, 2013.

[35] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proc. IEEE*, 2017.

[36] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Reliability issues in flash-memory-based solid-state drives: Experimental analysis, mitigation, recovery. In *Inside Solid State Drives*. Springer, 2018.

[37] Giryoung Kim and Dongkun Shin. Performance analysis of SSD write using TRIM in NTFS and EXT4. In *ICCIT*, 2011.

[38] Marcel Breeuwsma, Martien De Jongh, Coert Klaver, Ronald Van Der Knijff, and Mark Roeloofs. Forensic data recovery from flash memory. *SSDDFJ*, 2007.

[39] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *USENIX ATC*, 2017.

[40] Sungjin Lee, Jisung Park, and Jihong Kim. Flashbench: A workbench for a rapid development of flash-based storage devices. In *RSP*, 2012.

[41] Ronald Rivest. The MD5 message-digest algorithm. https://tools.ietf.org/html/rfc1321, 1992.

[42] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Linux Symposium*, 2007.

[43] Seiichi Aritome. *NAND flash memory technologies*. John Wiley & Sons, 2015.

[44] Chun-Hsiung Hung, Meng-Fan Chang, Yih-Shan Yang, Yao-Jen Kuo, Tzu-Neng Lai, Shin-Jang Shen, Jo-Yu Hsu, Shuo-Nan Hung, Hang-Ting Lue, Yen-Hao Shih, Shih-Lin Huang, Ti-Wen Chen, Tzung Shen Chen, Chung Kuang Chen, Chi-Yu Hung, and Chih-Yuan Lu. Layer-aware program-and-read schemes for 3D stackable vertical-gate BE-SONOS NAND flash against cross-layer process variations. *IEEE JSSC*, 2015.

[45] Yi Wang, Lisha Dong, and Rui Mao. P-Alloc: Process-variation tolerant reliability management for 3D charge-trapping flash memory. *ACM TECS*, 2017.

[46] JEDEC. Solid-state drive (SSD) requirements and endurance test method, 2010.

[47] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and W. Edward Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security*, 2008.

[48] Franck Courbon, Sergei Skorobogatov, and Christopher Woods. Reverse engineering flash EEPROM memories using scanning electron microscopy. In *CARDIS*, 2016.

[49] Tech insights. Intel/Micron 64L 3D NAND analysis. https://www.techinsights.com/blog/intelmicron-64l-3d-nand-analysis, 2017.

[50] Rino Micheloni, Alessia Marelli, and Kam Eshghi. *Inside Solid State Drives*. Springer, 2013.

[51] Myungsuk Kim, Jaehoon Lee, Sungjin Lee, Jisung Park, and Jihong Kim. Improving performance and lifetime of large-page NAND storages using erase-free subpage programming. In *DAC*, 2017.

[52] Alessandro Torsi, Yijie Zhao, Haitao Liu, Toru Tanzawa, Akira Goda, Pranav Kalavade, and Krishna Parat. A program disturb model and channel leakage current study for sub-20 nm NAND flash cells. *IEEE TED*, 2010.

[53] Christian Monzio Compagnoni, Andrea Ghetti, Michele Ghidotti, Alessandro S. Spinelli, and Angelo Visconti. Data retention and program/erase sensitivity to the array background pattern in deca-nanometer NAND flash memories. *IEEE TED*, 2009.

[54] Chulbum Kim, Doo-Hyun Kim, Woopyo Jeong, Hyun-Jin Kim, Il Han Park, Hyun-Wook Park, JongHoon Lee, JiYoon Park, Yang-Lo Ahn, Ji Young Lee, Seung-Bum Kim, Hyunjun Yoon, Jae Doeg Yu, Nayoung Choi, NaHyun Kim, Hwajun Jang, JongHoon Park, Seunghwan Song, YongHa Park, Jinbae Bang, Sanggi Hong, Youngdon Choi, Moo-Sung Kim, Hyunggon Kim, Pansuk Kwak, Jeong-Don Ihm, Dae Seok Byeon, Jin-Yub Lee, Ki-Tae Park, and Kye-Hyun Kyung. A 512-Gb 3-b/cell 64-stacked WL 3-D-NAND flash memory. *IEEE JSSC*, 2017.

[55] Samsung. Samsung V-NAND technology: Yield more capacity, performance, endurance and power efficiency. https://studylib.net/doc/8282074/samsung-v-nand-technology, 2014.

[56] Zbysek Gajda and Lukas Sekanina. Reducing the number of transistors in digital circuits using gate-level evolutionary design. In *GECCO*, 2007.

[57] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX Login*, 2016.

[58] Samsung Electronics. Galaxy S2 black. https://www.samsung.com/uk/smartphones/galaxy-s2/GT-I9100LKAXEU/, 2011.

[59] Dan Boneh and Richard J. Lipton. A revocable backup system. In *USENIX Security*, 1996.

[60] Jaeheung Lee, Junyoung Heo, Yookun Cho, Jiman Hong, and Sung Y. Shin. Secure deletion for NAND flash file system. In *SAC*, 2008.

[61] Junghee Lee, Kalidas Ganesh, Hyuk-Jun Lee, and Youngjae Kim. FeSSD: A fast encrypted SSD employing on-chip access-control memory. *IEEE CAL*, 2017.

[62] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2013.

[63] Tilo Müller, Tobias Latzo, and Felix C. Freiling. Self-encrypting disks pose self-decrypting risks: How to break hardware-based full disk encryption. *Technical Report Friedrich-Alexander University of Erlangen-Nuremberg*, 2012.

[64] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

[65] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA*, 2009.

[66] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA*, 2009.

[67] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 2010.

[68] H-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. Phase change memory. *Proceedings of the IEEE*, 2010.

[69] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S. A. Wolf, A. W. Ghosh, J. W. Lu, S. J. Poon, M. Stan, W. H. Butler, S. Gupta, C. K. A. Mewes, Time Mewes, and P. B. Visscher. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics*, 2010.

[70] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *ISPASS*, 2013.

[71] H-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming-Jinn Tsai. Metal–oxide RRAM. *Proceedings of the IEEE*, 2012.