Enabling Secure and Efficient Data Loss Prevention with a Retention-aware Versioning SSD

Weidong Zhu
Florida International University
Miami, Florida, USA
weizhu@fiu.edu

Carson Stillman
University of Florida
Gainesville, Florida, USA
carson.stillman@ufl.edu

Sara Rampazzi
University of Florida
Gainesville, Florida, USA
srampazzi@ufl.edu

Kevin R. B. Butler
University of Florida
Gainesville, Florida, USA
butler@ufl.edu

Abstract

Cyberattacks resulting in data loss remain a critical concern in modern data protection. To mitigate such threats, data versioning has been introduced to recover compromised data by reverting the storage to a prior uncompromised state. However, most current versioning solutions are implemented at the host level (e.g., within the operating system), making them vulnerable to adversaries with escalated privileges who can compromise OS-level protections. Thus, device-level methods have been proposed to shift the versioning logic to hardware-isolated storage devices outside the untrusted OS. Unfortunately, these solutions suffer from limited retention times for historical data, narrowing the protection window and leaving systems exposed to persistent attacks. In this paper, we propose *LAST*, an inva**L**idation-**A**ware Ver**S**ioning sys**T**em for flash-based SSDs, that enables data versioning with enhanced awareness of data retention time, ensuring long-term availability of historical data with small performance impact. LAST modifies the SSD's flash translation layer (FTL) to retain the data invalidation order for tracking data retention time. Then, it leverages an ordered garbage collection (GC) that always reclaims versioned data with the longest retention time, as determined by the invalidation sequence. Therefore, this approach prevents the premature deletion of data with shorter retention, significantly extending the protection window and reducing the risk of data loss. Evaluated under various real-world workloads, LAST achieves a small latency overhead of 1.5% over a regular SSD while maintaining data history for up to 126.4 days with an average of 52.6 days. This significantly outperforms the average retention of current versioning methods by 61.4% at least and 165.9% at most, enhancing the protection window against data loss from cyberattacks.

CCS Concepts

• **Security and privacy → Malware and its mitigation**; **Systems security**; • **Computer systems organization → Secondary storage organization**.

Keywords

Malicious Attacks, Data Loss Prevention, Data Versioning, Flash-based SSDs

CCS '25, October 13–17, 2025, Taipei

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765135

ACM Reference Format:
Weidong Zhu, Carson Stillman, Sara Rampazzi, and Kevin R. B. Butler. 2025. Enabling Secure and Efficient Data Loss Prevention with a Retention-aware Versioning SSD. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765135

1 Introduction

Modern computer systems suffer from malicious attacks resulting in data loss or downtime, causing significant financial and operational disruption. For example, the 2021 ransomware attack on Colonial Pipeline encrypted essential systems, shutting down fuel delivery for days and leading to a severe gas shortage on the U.S. East Coast [39]. Beyond external attacks, malicious insiders with authorized access pose equally severe risks, causing catastrophic damage to data integrity, finances, and corporate reputation [29].

To combat these threats, various techniques, such as backups [35], process checkpoints [28], and snapshots [6, 87], have been widely used to preserve the previous storage state to recover the system from potential data corruptions. However, these techniques capture historical data only at discrete intervals, leaving gaps between snapshots. In contrast, a versioning system offering Continuous Data Protection (CDP) retains unlimited versions of data, allowing the storage to revert to any past state. This is particularly critical when facing adversaries who can corrupt data unpredictably at any time, as traditional methods may only capture snapshots of data after the compromise has occurred. Thus, we focus on CDP to protect users' data, consistent with prior versioning works [46, 59, 86, 96, 100].

An effective versioning system must address three critical aspects to ensure robust data protection:

(1) **Rootkit Resistance.** Recent attacks [45, 79, 86] have shown how retained data versions can be compromised by system breaches with privilege escalation. With these escalated privileges, attackers can disable the data protection processes and manipulate victim data. Therefore, a trustworthy versioning system should be enabled to defend against rootkit-level data corruptions.

(2) **Long Retention for Historical Data.** Providing long retention of historical data is desired by current versioning systems [15, 45, 79, 86, 96]. It guarantees data availability for remediating potential data compromise. For instance, advanced ransomware presents detection challenges due to its ability to mimic legitimate applications [103], indicating that the versioning system should provide longer data retention for the prolonged detection time. Moreover, some cyber attacks [102] can remain hidden in the victim's system for extended periods, even for months [73], before being

detected. Thus, data breaches may have occurred in the distant past, and providing long data retention can mitigate this threat.

(3) **Intact Historical Files.** Since computer data is typically stored in file form, it is crucial to ensure the completeness and integrity of versioned files. Partial loss or early deletion of historical files – called "shattered files" — can render recovered files unusable. For example, losing just the header information of a historical PDF file can make the entire file unusable.

Current versioning systems fall short in addressing these requirements, limiting their effectiveness in preventing data loss. Many current versioning approaches are employed in the host OS, called host-level methods [67, 76, 100]. However, the host system contains a large trusted computing base (TCB), leaving a large attack surface and making it vulnerable to privilege escalation vulnerabilities. In 2023 alone, 5870 vulnerabilities involving privilege escalation were reported (CVE-assigned) [14], highlighting significant risks for host-level solutions.

To remedy these limitations, others have proposed device-level versioning [46, 96] to provide a robust defense against OS compromise. These approaches are implemented within storage devices, isolated from the potentially compromised OS by limited interfaces. Flash-based SSDs are among the most popular storage devices used by existing device-level solutions both because of their prevalence and the fact they perform *out-of-place* writes rather than directly overwriting data on the medium, providing an inherent versioning functionality [96]. When data is written to the SSD, device-level approaches retain it until the storage space is insufficient to serve new writes, and garbage collection is triggered.

Despite the improvement, existing device-level approaches still suffer from critical limitations:

(1) **Degraded Data Retention.** While data versioning aims to maintain all the historical data in the device, the limited storage capacity indicates that the versioning system must reclaim the historical data periodically to release the storage space for future data operations. Without carefully selecting which historical data to erase, these methods can prematurely delete data that could be crucial for recovery. For example, reclaiming a historical data page with a short retention time degrades the protection of historical data when there is a historical data with longer retention time existed. While current versioning SSDs [79, 96] recognize this threat, they cannot track the complete retention time order of data versions. This limitation results in the early deletion of versioned data and a diminished capability to recover the system following a data breach.

(2) **Incomplete Historical Files.** Storage devices lack knowledge of file-level relationships between the data arrived at the storage and files because they operate at the block level. Moreover, "removed" but versioned data in files are typically generated in continuous time [37, 48] due to spatial locality, which indicates that data adjacent to a recently accessed address will likely be accessed shortly. Since existing versioning SSDs [46, 79, 96] cannot be fully aware of the data retention sequence, they can occasionally erase portions of a versioned file early, leading to shattered files.

(3) **Data Loss Due to Architectural Defects.** The DRAM data cache has been widely used in modern SSDs [18, 54, 88] to absorb I/O requests for high performance. However, the data cache in the SSD can cause the loss of data versions because *a write hit to the data cache can overwrite data that has been recently cached but not yet written to storage.*

To address these limitations, we propose LAST, a secure and efficient SSD-based versioning approach. The core idea behind LAST is simple but effective: when historical data reclamation is required, it always deletes data versions with the longest retention duration first. To achieve this target, LAST distinguishes between newly written data (*first-time*) and the data (*overwriting*) that overwrites existing data, storing each type separately in the storage based on their arrival times. Since the retention time of versioned data depends on when data is invalidated and when it is truly deleted from storage, the order in which overwriting data is stored reflects the invalidation sequence and thereby represents the retention time. Therefore, LAST leverages the order-maintained overwriting data to select the historical data with the longest retention time during the reclamation.

LAST also avoids data loss in the DRAM data cache due to the cache hit. Unlike the traditional overwrite-upon-hit caching method, LAST disables the in-place update operations in the data cache and stores the incoming data in an out-of-place manner. This ensures that the historical data cannot be deleted or overwritten in the cache. Additionally, since maintaining the historical data introduces additional storage overhead, LAST further introduces the deduplication technique to effectively remove the duplicated data copies in the storage, improving storage efficiency.

With these enhancements, LAST offers significant advantages over existing methods: *(1)* It maximizes the retention of historical data because the recently deleted versioned data can be preserved longer, avoiding premature removal. *(2)* Due to spatial locality, file-related historical data is typically generated consecutively. LAST reclaims file data based on its original invalidation order, ensuring files remain intact and avoiding shattered files. *(3)* LAST's cache design prevents accidental version loss inherent to conventional DRAM caching techniques.

Our evaluation demonstrates that LAST significantly improves historical data retention, outperforming existing device-level solutions by 61.4% at minimum and 165.9% at maximum. This substantial improvement in retention is essential for effective recovery from sophisticated, long-hidden cyberattacks. In summary, this work makes the following contributions:

- We perform an in-depth analysis of data versioning in SSDs equipped with DRAM cache and characterize data retention in existing versioning SSDs to give evidence of why they are insufficient in preserving data versions.
- We propose LAST to protect the versioning system from OS compromises. It enables versioning data in the cache with a trivial overhead of 1.5% compared to a regular SSD.
- We provide ordered versioning with LAST that leverages data invalidation sequences to avoid shattered files and achieve long retention time, outperforming existing versioning SSDs by 61.4% at minimum and 165.9% at maximum.
- We implement a prototype of LAST in an SSD emulator [61]. Our evaluation demonstrates that LAST can improve data versions' availability with minimal performance degradation compared to existing versioning SSD methods.

2 Case Study of Data Loss

In this section, we explore critical attack scenarios that can lead to data loss. Building on these scenarios, we identify key insights that highlight essential factors for effective data loss prevention.

2.1 Encryption Ransomware

Ransomware is a high-profile malware that encrypts user files until a ransom is paid. These attacks can disrupt critical services across government agencies, businesses, and individual consumers. Even worse, ransomware has a low entry barrier for attackers. In 2024, the number of ransomware victims worldwide rose by 15% compared to the previous year [34], while global ransom payments increased by 35% [92]. Although traditional ransomware typically follows a straightforward pattern of file access and encryption, recent variants have shown increasingly sophisticated patterns:

(1) Recent research [27, 45, 94] demonstrates that some ransomware attacks can operate at the rootkit level, undermining typical detection and backup strategies. They can disable existing defense mechanisms or bypass detection entirely. For example, with escalated privileges, it might read or modify data directly on the device rather than interact with files [105]. As most existing solutions rely on file access behavior to identify threats [51, 52, 83], this approach circumvents detection even if the ransomware does not disable the operating system's defenses.

(2) Attackers can also adapt ransomware to outmaneuver specific defenses. For example, a common detection technique focuses on changes in data entropy, as encryption raises the randomness of file contents. However, emerging ransomware variants, such as CHAOS [72], exploit encoding methods (e.g., Base64) to lower entropy, making it harder for existing mechanisms to identify ransomware activity and prolonging the attack time before discovery.

Takeaway. Considering advanced ransomware that can operate at the rootkit level and adapt to evade detection, immediate detection and termination are not guaranteed. Therefore, an effective defense strategy must include a robust data recovery mechanism and store historical data for as long as possible to facilitate data recovery if an attack happens.

2.2 Wiper Malware

Unlike ransomware attacks, wiper malware is designed purely for destruction rather than seeking financial reward. Attackers often remain silent until the compromised system fails, causing severe operational disruptions. For instance, in 2022, Russian-based groups performed wiper attacks on Ukrainian government systems, interrupting critical infrastructure and business activities. Unlike many destructive cyberattacks, wiper attacks have distinctive characteristics:

(1) Since these attacks aim solely to destroy data, they frequently target high-value systems whose compromise has significant societal or national security consequences.

(2) Wiper attacks tend to be large in scale and highly coordinated, often initiated by governments or large cybercriminal organizations, such as the NotPetya malware [77] and Iran Wiper [71].

Takeaway. Wiper attacks pose a severe threat to critical infrastructures and organizations where data availability is critical. Once the attack is activated, it can erase data immediately. Defending against such attacks relies heavily on effective data recovery, making comprehensive versioning essential. Therefore, maintaining historical data provides the foundation for robust data loss prevention.

2.3 Insider Threats

While malware is the primary cause of data loss, malicious insiders with legitimate system access can lead to significant negative outcomes as well. Moreover, such attacks are especially difficult to counteract for the following reasons:

(1) Since insiders are typically trusted users, it may take a long time to discover such intentional data corruption. For example, a former IT employee of NCS conducted an insider attack for 2.5 years, wiping 180 virtual servers and causing $678,000 in financial losses. Moreover, organizations could only realize they have been compromised after substantial damage has already been done.

(2) Insiders typically possess extensive knowledge of the system withy high-level privileges. They can easily bypass security controls and blend malicious actions with normal activities. Consequently, insider-induced data loss is more difficult to detect.

Takeaway. Although insider threats are less common than malware, they pose a severe challenge for data loss prevention. Insider threats can remain hidden in a system for extended periods, and insiders' elevated privileges allow them to evade many host-level defenses. As with rootkit-level attacks, it is crucial to preserve historical data for possible data recovery while ensuring strong isolation measures that prevent unauthorized access to defense mechanisms.

2.4 Learned Lessons

Real-world attack scenarios underscore the urgency of designing robust and secure data loss prevention strategies that provide the following guarantees. *(1)* All historical data should be retained, as attacks can corrupt any part of data at any point in time. This approach maximizes the availability of historical data for potential recovery. *(2)* The protection system itself should be isolated from vulnerable modules in the host environment to avoid being compromised. *(3)* Since some threats can persist for a long period of time, historical data must be kept as long as possible to support recovery even long after an initial attack.

Based on these observations, this paper proposes a solution that maximizes historical data retention and defends against advanced attacks—including those with rootkit-level privileges, thereby providing robust data loss prevention.

3 Flash-based SSDs

This section provides necessary background information on flash-based SSDs.

Overview. Flash-based SSDs are replacing traditional hard disk drives (HDDs) for their high performance and energy efficiency [44]. Flash memory operates write and read requests at page granularity (e.g., 4KB). These pages are grouped into a flash block. Since flash memory erases data at block granularity, it performs overwrites in an out-of-place manner. When the ratio of free pages reaches a threshold (e.g., 20%), garbage collection (GC) employed in the flash translation layer (FTL) reclaims invalidated data by migrating valid pages to other free blocks and erasing them. Since flash memory has limited program/erase cycles [55], FTL includes wear-leveling and bad block management to improve the SSD's lifetime.

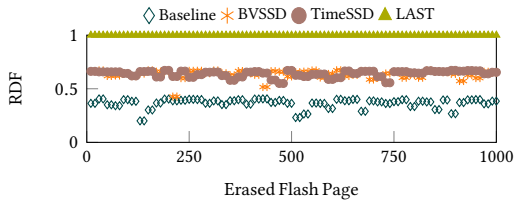

Figure 1: Retention Drop Factor (RDF) of erased pages when testing a random write workload of FIO. LAST reaches optimal retention time compared to existing versioning SSDs.

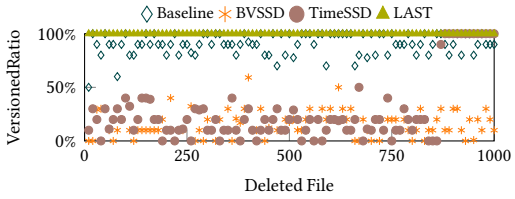

Figure 2: Retained version ratio of deleted files after a random write. LAST reaches 100% ratio compared to the Baseline SSD and current versioning SSDs (BVSSD and TimeSSD).

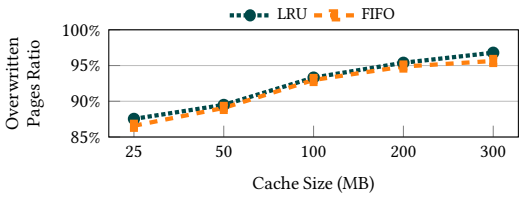

Figure 3: Average ratio of overwritten pages in the data cache at increasing cache sizes for LRU and FIFO policies. A larger cache removes more data versions due to its higher hit ratio.

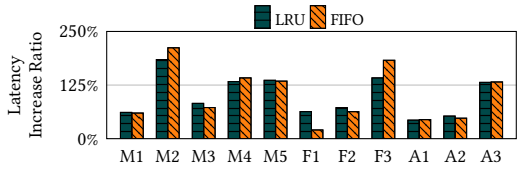

Figure 4: The ratio of latency increases when cache versioning is introduced in SSDs equipped with LRU and FIFO policies. We evaluate them by testing workloads shown in Table 2.

Flash Memory Data Layout. Flash-based SSDs are structured for high parallelism. They use bus channels to connect multiple flash chips that process I/Os independently. Each chip contains flash dies, offering die-level parallelism, and each die consists of planes made up of blocks. Finally, planes, the lowest level of parallelism in SSDs, can process I/Os simultaneously when requests target the same address.

Address Translation. Address translation interprets logical page addresses (LPAs) derived from logical block addresses (LBAs)in the OS to physical page addresses (PPAs) in flash memory using a mapping table [42, 48]. Page-level mapping has been widely used in light of its high performance [42, 74]. However, DRAM has a limited capacity – typically 0.1% [63] of flash memory – and it cannot house the entire mapping table. Thus, FTL uses a cached mapping table (CMT) to store those "hot" mapping entries in the DRAM, while the complete table is maintained in flash memory's *translation pages*, organized by a global mapping directory (GMD).

DRAM Data Cache. DRAM is also deployed as a data cache [18, 54] to absorb I/O requests for low-latency access. For example, the LRU policy is a classical caching algorithm used for SSDs [88], prioritizing storing the data that has been recently accessed. If an I/O request hits data in the cache, it will be serviced by the DRAM without accessing flash memory.

4 Motivation

In this section, we compare existing host-level and device-level versioning methods. We then examine how current versioning SSD solutions fail to track retention time orders, leading to data corruption. Moreover, we explore the challenges of data versioning in SSD caches. Finally, we outline the motivation behind proposing LAST to address these issues.

4.1 Host-Level vs Device-Level Versioning

Host-level versioning has been widely used for data history preservation and is often deployed in the OS, at the filesystem level [6] or the block layer [100]. However, such approaches have shown critical limitations.

In general, the OS exposes a larger attack surface to adversaries because of its large Trusted Computing Base (TCB). Attackers can exploit this surface to perform privilege escalation [4, 7] and gain root privilege. Then, they can disable host-level versioning and consequently compromise the data recovery. For example, ransomware actors can exploit a vulnerability in the anti-cheat driver of Genshin to escalate privileges, allowing them to disable antivirus processes [85]. In addition, host-level solutions introduce extra write traffic for versioning purposes. Thus, they also suffer from significant performance degradation [84, 87, 96].

In contrast, device-level solutions [46, 96] shift the versioning enforcement into the storage device to counteract OS compromise. Moreover, they achieve transparent data preservation without introducing extra I/O traffic. As a result, device-level approaches offer advantages in defending against privileged adversaries [86] while incurring trivial performance overhead [96] compared to host-level methods. Therefore, this work primarily focuses on device-level versioning methods.

4.2 Characterizing Data Retention in Versioning SSDs

Flash-based SSDs make a compelling platform for device-level versioning [45, 46, 79, 96] because of their out-of-place update property and inherent logging functionality.

Long data retention is crucial for mitigating system compromises, as discussed in Section 2.4. Thus, we assess data retention efficiency in an unmodified SSD (Baseline) and current versioning SSDs –

BVSSD [46] and TimeSSD[1] [96]. Implementation details are provided in Section 8.1. We define the retention drop factor (RDF) for each reclaimed flash page as:

$$RDF = RT_{Selected} / RT_{Longest} \quad (1)$$

where $RT_{Selected}$ represents the retention time of the selected erased data page, and $RT_{Longest}$ is the retention time of the longest-held page upon the page erasure. If the versioning system can always pinpoint the page with the longest RT, the RDF of the page is 1, as $RT_{Selected} = RT_{Longest}$ and $RDF = \frac{RT_{Selected}}{RT_{Longest}} = 1$. Otherwise, a page with a shorter RT might be selected, resulting in $RT_{Selected} < RT_{Longest}$ and $RDF < 1$.

In Figure 1, the RDFs of the SSDs range from 0.2 to 0.7. Specifically, Baseline, BVSSD, and TimeSSD achieve average RDFs of 0.35, 0.62, and 0.63, respectively. The small RDFs of BVSSD and TimeSSD are because they cannot identify the retention time of versioned data, causing the deletion of data with short retention time. Thus, current versioning SSDs exhibit degraded data retention, limiting the capability of recovering a data compromise.

Motivation 1: Current versioning SSD solutions do not optimally retain pages in the SSD because they cannot track the order of the retention time.

For a versioned file, removing its partial pages can result in the file being unusable. For example, a versioned executable file cannot be used after its recovery if some of the pages that make up the file are reclaimed. We call these files *shattered*. To evaluate the prevalence of this issue in state-of-the-art versioning SSDs, we define $VersionRatio$ for a versioned file:

$$VersionRatio = NUM_{Retained} / NUM_{TT} \quad (2)$$

where $NUM_{Retained}$ means the number of retained data pages of the file, and NUM_{TT} is the total number of data pages of the file. We create 1,000 files, each containing 1MB of data, on versioning SSDs and delete them after running a FIO [3] random write workload (200GB). We then run the same FIO workload again to evaluate the ratio ($VersionRatio$) of retained pages in each deleted file after execution. Since the deleted data (over 170GB) generated by a FIO workload exceeds the total reclaimed data (21.8GB) during GC, the versioned files should not be reclaimed if the SSD correctly prioritizes data based on retention time. However, as shown in Figure 2, existing versioning SSDs, which cannot fully track retention time, may prematurely erase some data pages in a deleted file (i.e., $VersionRatio < 100\%$), leading to potential file corruption.

Motivation 2: Current versioning SSDs do not fully maintain the invalidation order of data pages, leading to shattered files and decreased file usability after recovery.

4.3 Characterizing Cache Versioning in SSDs

DRAM data cache [18, 54, 88] has been widely used in flash-based SSDs to boost performance. Incoming data is first served by the cache before being evicted to the flash memory. However, upon a write hit to the cache, the cached data is overwritten. This compromises system security by allowing attackers to overwrite recently written data that remains in the cache and has not yet been saved (i.e., versioned) in flash memory.

To show this impact, we implement two classical caching policies (i.e., LRU and FIFO) into regular SSDs [61] equipped with a DRAM cache shown in Table 1 without cache versioning. Figure 3 shows that the data cache incurs significant versioned data removal when testing the workloads in Table 2. Moreover, the removal worsens at increasing cache size because it results in a higher hit ratio which in turn means more overwritten data.

Motivation 3: For SSDs equipped with DRAM data cache, they lose version history for all write-hits in the cache, degrading their capability to recover data.

An intuitive remedy for Motivation 3 is to perform out-of-place writes in the data cache upon a write hit. Thus, we implement a versioning SSD that operates write requests by writing all incoming data to a free space in the DRAM, even in the case of a write hit. Read requests operate in the same way as the SSD without cache versioning. We observe that when the data cache is filled, the write request incurs a data eviction. We then employ LRU and FIFO to versioning SSDs that incorporate cache versioning and compare their performance with regular SSDs equipped with the same caching policies. Figure 4 shows that versioning the data cache significantly increases the average latency over LRU and FIFO regular SSDs by 100.1% and 101%, respectively. This is because, even on a cache hit, writes are forced to write in a new DRAM space, and thus cache eviction cannot be avoided.

Motivation 4: Versioning the DRAM cache is challenging because it can significantly degrade the performance.

4.4 Why LAST?

In light of the aforementioned considerations, we are motivated to propose LAST for the following reasons.

First, attackers with escalated privileges can kill any data protection process in the OS. LAST remedies this by providing versioning enforcement inside the storage device, which is hardware-isolated from the OS, providing resistance against privileged attacks and outperforming host-level solutions.

Second, since a privileged adversary can disable malware detection, and compromises often take time to be discovered [73, 102], it is critical to enable long retention of versioned data to ensure availability. Therefore, we design LAST to prolong the preservation of retained data in the storage device.

Third, current versioning SSDs cannot identify the retention time order for versioned data pages, leading to suboptimal data retention and shattered files, as discussed in Section 4.2. While bloom filter (BF)-based methods[79, 96] were introduced to record retention time, they fail to track the complete data retention order for two reasons: (1) A BF only tracks the presence of elements (versioned data) but cannot provide the correlations (order) of the stored versioned data. (2) DRAM size might be too small to record the full data invalidations (as happens for TimeSSD [96] and RSSD [79]). It is necessary to delete the old BFs to release DRAM space, leading to the loss of data invalidations. In this work, we show how our LAST can track the complete data retention order for efficient data versioning.

[1] Although RSSD [79] is the most recent versioning work, it relies on the same retention time identification algorithm (i.e., bloom filter) as TimeSSD. Moreover, RSSD requires dedicated network hardware equipped on the SSD, which is not applied to current commercial SSDs. Our work aims for data versioning in regular SSDs, and thus we select TimeSSD as a comparison.

Finally, current versioning SSDs do not consider the DRAM data cache. If a data cache is employed, it can lead to version loss on write cache hits. Thus, we design LAST to overcome this limitation by versioning the data in the SSD's data cache while introducing minimal overhead.

5 Threat Model

In this work, we focus on adversaries who attempt to compromise stored data to make it inaccessible; for example, they can employ ransomware [45] or wiper [9] attacks to prevent users from accessing their data. Moreover, we assume that the host OS is vulnerable to attacks from adversaries that can fully compromise it and gain escalated privilege (root) [4, 7, 79, 96, 105]. Thus, the OS is untrusted, and adversaries can disable and manipulate data protection mechanisms deployed in the OS.

We trust the firmware of SSD [25, 79, 96] due to its reduced trusted computing base (TCB) and independent processor and memory, which provides strong isolation from the host OS. We also assume that the firmware cannot be modified in the SSD [79, 96], secured by methods like digital signature and secure boot [24, 25]. We neglect the data loss risk in the DRAM due to power failure because modern SSDs typically have capacitor- [21] or battery-backed [50] DRAM. Thus, we trust storage devices (i.e., FTL) and assume the DRAM data is preserved during power failure. We trust the hash algorithm used in our deduplication and disregard its collisions, as we employ BLAKE2 [26], which is secure with no known collisions. Preventing hash collisions [36, 98, 99] is an active research area outside the focus of this work. Other collision-prevention methods [36, 98] can be applied to our approach.

We trust the manufacturer of the SSD, as assumed in prior works [25, 58, 96], to securely generate and embed a credential. The credentials are distributed to authenticated users and maintained securely using existing secure key techniques, such as USB security key [22]. Thus, we assume that there is a trusted security administrator to hold the credential securely, which can be used for authentication when version management, such as data recovery, needs to proceed. Upon the successful compromise of data by privileged adversaries, the security administrator can disconnect the SSD and plug it into another trustworthy computer to ensure secure data recovery [96].

6 LAST Design

In this section, we present an architectural overview of LAST, followed by the details of its critical components.

6.1 LAST Overview

LAST employs *Versioning Cache Management* to manage the DRAM data cache while tracking the data retention through the data invalidation sequence. To mitigate the overhead of cache versioning (see Section 4.3), LAST reserves a dedicated read cache for read requests. When the DRAM cache is full, Versioning Cache Management leverages OPEN to efficiently evict the cached data to flash memory while preserving the invalidation sequence in the flash memory. Thus, LAST can leverage *Order-aware GC* to reclaim data versions based on invalidation order, avoiding premature removal of versioned data. Moreover, LAST devises a lineage-preserved deduplication engine (*LPDedup*) to reduce data writes, enhancing flash memory lifetime. Finally, LAST offers efficient version management (*Version Manager*) for managing retained versions.

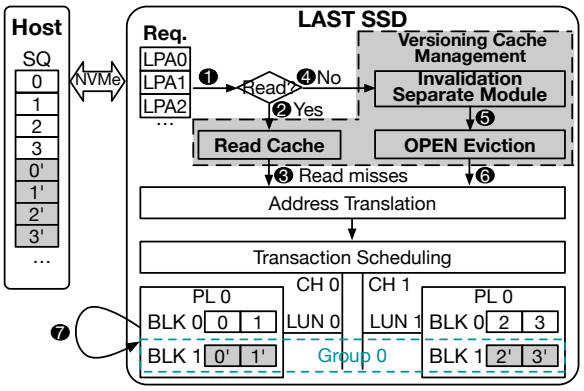

Figure 5: The I/O workflow of LAST. LAST leverages versioning cache management (grey area) for incoming I/O requests, and it manages data eviction from the DRAM to flash memory.

Figure 5 shows the I/O workflow of LAST. LAST judges the incoming I/Os to determine their request type ❶. For a read request ❷, LAST first queries the data cache. If the read hits, it will be served by the DRAM cache. Otherwise ❸, LAST leverages the address translation to locate the requested data on flash memory and read it to complete the read. For write requests, the invalidation separate module identifies the overwriting data and stores them separately in the DRAM for tracking their order ❹. When the data cache is filled ❺, LAST leverages OPEN to evict data from the DRAM cache to flash memory ❻. Finally, LAST proceeds GC through Order-aware GC and LPDedup to reclaim versioned data orderly while saving storage space ❼.

6.2 Versioning Cache Management

LAST devises cache versioning to retain the data invalidation order with trivial overhead, containing the following modules.

Read Cache. Since DRAM cache versioning can lead to significant performance degradation as discussed in Section 4.3, LAST maintains a read cache region for handling read requests, reducing read overhead. In Figure 6, LAST creates a read cache list to store the cached data for immediate access on read hits. If the read data is not in the read cache or other caches, LAST retrieves it from flash memory to finish the request and adds the data into the read cache. For data replacement, we use the classical LRU policy in the read cache as an exemplar, while other caching methods can be applied.

Multiple data versions exist in the SSD. To ensure reading the correct (newest) data, LAST leverages the L2P table, which records the mapping from the logical address to the latest physical data page. Thus, the read cache can retrieve the newest data version by querying the L2P table.

Invalidation Separate Module. Since data invalidation determines the retention time, LAST maintains the invalidation sequence of versioned data to identify their retention time order. To retain such sequence, LAST classifies the incoming write data into *first-time* and *overwriting* categories, where overwriting requests invalidate existing data. Thus, the invalidation order can be tracked by maintaining

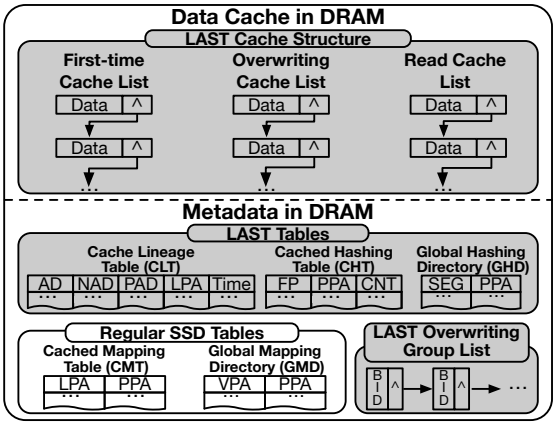

Figure 6: Introduced data structures (grey area) of LAST. LAST manages the data cache through link lists and introduces CLT and Overwriting Group List to store the invalidation order. CHT and GHD are used for LAST deduplication.

Algorithm 1 Interleaving Eviction of OPEN.

Input: N_CHs = Number of channels, N_LUNs = Number of flash chips
N_PLs = Number of planes, N_PGs = Numer of pages in a block
PGs_per_CH = Number of pages per channel
PGs_per_LUN = Number of pages per chip
PGs_per_PL = Number of pages per plane
$BLKs\_per\_PL$ = Number of blocks per plane
BID = Current block ID for data group

$NUM = N\_CHs * N\_LUNs * N\_PLs * N\_PGs$
$PG\_Set = \{\}$
if First-time region or overwriting region filled **then**
  PG_Set = {NUM of adjacent pages in the cache region}
end if
for PG in {0,1,...,N_PGs-1} **do**
  for PL in {0,1,...,N_PLs-1} **do**
    for LUN in {0,1,...,N_LUNs-1} **do**
      for CH in {0,1,...,N_CHs-1} **do**
        $Evic\_Data = PG\_Set[PG + PGs\_per\_PL*PL + PGs\_per\_LUN*LUN + PGs\_per\_CH*CH]$
        Flush $Evic\_Data$ to the address (CH, LUN, PL, BID, PG)
      end for
    end for
  end for
end for
$BID = (BID + 1)\ \%\ BLKs\_per\_PL$

the arrival of overwriting requests sequentially. Since deletion requests (e.g., Trim [11]) can also invalidate data, LAST transfers a deletion request to an overwriting operation by writing with 0s. In Figure 6, LAST creates first-time and overwriting cache lists to maintain first-time and overwriting data, respectively, in the data cache. The invalidation (retention) order can be maintained by individually caching overwriting data sequentially based on arrival time in the overwriting cache list. For a write request, LAST checks the LPA to PPA mapping table to determine its data type. If the accessed address does not exist in the cached mapping table (CMT) and has no entries in the full mapping table in flash memory, which can be queried through the global mapping directory (GMD), the write request is directed to the first-time cache. Otherwise, the data is sent to the overwriting cache. Data within these caches is organized into linked lists, with newer entries added to the end as shown in Figure 6, maintaining the invalidation order for effective version management.

Order-Preserving EvictioN (OPEN). OPEN handles data eviction for the first-time and overwriting cache regions, and it aims to store adjacent data pages from first-time or overwriting regions in the same flash block sequentially, mirroring their order in the data cache. However, this requires evicting data from the cache to a flash block sequentially, breaking the parallelism usage of flash memory [91] and leading to non-trivial performance overhead [104]. To avoid this, LAST leverages OPEN to organize data into eviction groups, each sized to match the parallelism of the flash memory. Then, OPEN selects blocks with the same block ID (BID) from each plane in flash chips to store the data in an eviction (e.g., overwriting) group. Unlike traditional caches that evict adjacent data into different flash blocks, OPEN selects interleaved data from an eviction group and evicts[2] them in parallel while maintaining invalidation order in a flash block, as illustrated in Algorithm 1.. This method also preserves the spatial locality of data, and versions with similar retention times are likely to be stored in the same data blocks, reducing GC overhead. Finally, OPEN creates an *overwriting group list* to track the invalidation sequence between evicted overwriting groups, and each overwriting group is related to a unique BID, which records the location of the data group in the flash memory.

Unlike traditional update-allowed cache, LAST can evict data without waiting for the cache regions to fill. OPEN leverages real-world workload idleness and high internal parallelism of flash memory to deploy background eviction. Real-world workloads often exhibit idle times [66], allowing the system to schedule background tasks [60, 96] for better performance. Moreover, flash memory can process multiple I/O requests simultaneously due to its internal parallelism. Thus, OPEN assesses idleness by monitoring flash memory bandwidth, differing from traditional idle time prediction [60, 96]. If the current write bandwidth exceeds a threshold (i.e., 20% of peak write bandwidth), OPEN stops background eviction; otherwise, it continues without waiting for cache regions to fill.

The cache flush command [101] is widely used in the storage device to ensure data consistency. It compels to flush the data from the cache to the flash memory. LAST enables such enforcement in the SSD. Upon a flush command, LAST forces the data eviction from the first and overwriting caches to flash memory with the same method as the background eviction.

6.3 Order-aware GC

LAST initiates GC when the ratio of free pages is lower than a threshold (e.g., 20%). LAST chooses versioned data to erase sequentially based on their retention times. Therefore, LAST needs to (1) search the overwriting data to determine the retention time and (2) pinpoint the location of versioned data.

Figure 7 shows the GC workflow. Overwriting groups are sequentially organized in the overwriting group list, with the head group representing the earliest data invalidation. LAST selects this *group* for garbage collection ①. Time lineage information is stored in the out-of-band (OOB) area of each flash page [96] to track data versions

[2]Eviction prioritizes the filled cache region matching incoming I/O types (overwriting or first-time). If the device is idle, first-time or overwriting cache regions are processed in FIFO sequence.

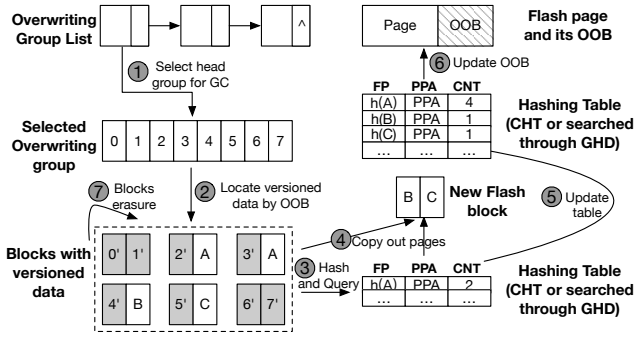

Figure 7: Example workflow of GC and deduplication in LAST. Pages (0' to 7') will be reclaimed as the data in the head group invalidates them. If LAST's deduplication identifies identical content, multiple data versions share the same flash page. Since the hash value of page A exists in the hashing table, it will not be migrated, and its reference count will increase by 2. The fingerprints of B and C do not exist in the hashing table; they will be migrated to a new flash block, and their hashing metadata will be inserted into the table.

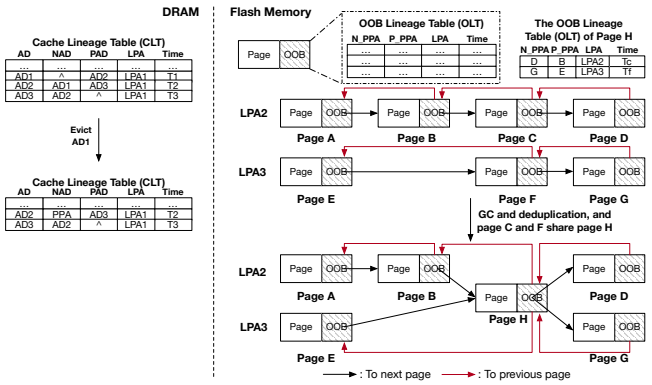

Figure 8: Example of LAST data lineage management. In CLT, when a data page (e.g., AD1) is evicted to flash memory, the metadata of the page (AD2) that overwrites the evicted data will be updated by changing its NAD to the new flash address (PPA). In OLT, each entry indicates a lineage. The OOB of page H includes two entries to indicate the previously recorded lineage information on pages C and F after deduplication.

(details in Section 6.5). LAST traverses the OOBs of data pages in the head group to locate versioned data for reclamation ②. Then, it copies the valid and versioned data pages to free blocks, updates their OOBs, and erases the selected blocks.

6.4 Lineage-preserved Deduplication

LAST utilizes lineage-preserved deduplication (LPDedup), which operates during GC, to save storage space. LPDedup preserves the invalidation order by maintaining the ordered layout of data within overwriting groups. In Figure 7, LAST computes the fingerprint using BLAKE2 [26], a fast hash algorithm used in deduplication SSDs [98]. It then searches a hash table to verify if the data exists ③. Each hash table entry is a key-value pair that uses the fingerprint as the index, followed by the address and reference count, which indicates the number of pages that share the fingerprint. Since DRAM cannot store all fingerprints and only a small fraction (10%-20% [31]) are highly duplicated, LPDedup maintains a cached hash table (CHT) in DRAM for recently accessed fingerprints, while the full table resides in flash memory and is managed by a global hashing directory (GHD). LAST splits the hashing space into segments with equation $Seg = H(P) \bmod n$, where $H(P)$ is the hash value of a data page P and n is the number of required data pages for the hashing table. Fingerprints in a flash page share the same Seg, and the GHD maps each Seg to its physical page address, allowing for querying of the full hashing table.

When migrating a page during GC, if its fingerprint is not found in CHT or the hashing table stored in flash memory, LAST copies the data to a new flash block ④ and adds the fingerprint to the CHT ⑤. Otherwise, it skips copying the data and updates or inserts – upon a miss in CHT– its hashing information in the CHT. Finally, LAST updates the metadata within the OOB ⑥ in an out-of-place manner by adding a new entry in its OLT using a partial page program[3] [75] for lineage management (see Section 6.5), following erasing the selected flash blocks ⑦. Since erasing a page requires adjusting the reference count (CNT) in the hash table or deleting the entry if the CNT reaches zero, LAST updates the CHT if it's cached. If not, it retrieves the hash mapping from flash memory, updates the entry, and writes it back to a new page.

6.5 Metadata of Versions

LAST records the updating correlations of the data stored within the DRAM cache and flash memory through a Cache Lineage Table (CLT). CLT contains the reflection between a data page in the cache and its metadata, which includes data creation time, the DRAM address (AD), the address (NAD) of the next data page updated by the current data, and the address (PAD) of the previous data page that updates the current data. When a data page is evicted from the cache to flash memory, LAST updates the CLT to maintain the lineage between the data page in the data cache and flash memory shown in Figure 8.

Since LAST introduces deduplication, multiple versioned or valid pages can share a physical flash page. Thus, LAST must identify the correct lineage from the duplicated data page, which can comprise multiple data lineages. In Figure 8, LAST creates an OOB Lineage Table (OLT) in the OOB area of each flash page to store metadata. The OLT of each flash page OOB comprises multiple entries representing different versioning chains to store the lineage information. Since the space of the OOB area is limited, LAST only allows ten lineage entries in an OLT, and we evaluate its sufficiency in Section 8.2.

6.6 Version Manager

LAST provides a lightweight tool for version management.

Version probe. To query the versioned data, users need to provide the address (LPA) and the creation time of the data. We thus provide

[3]For the SSD not supporting partial programming, LAST allows to redesign the OOB metadata management by storing the OLT information into flash memory directly and reserves a DRAM portion for updating lineage information, similar to the management of hash table.

Table 1: Parameters of the SSD used in our evaluation.

Parameters	Value	Parameters	Value
SSD Capacity	512GB	OOB Area	409B
Page Size	4KB	OOB Read	0.02ms
Pages/Block	256	Page Read	0.04ms
Blocks/Plane	16384	Page Write	0.2ms
Planes/Chip	1	Block Erase	2ms
Chips/Channel	8	Data Cache	300MB
Channels	4	Metadata DRAM	200MB

two interfaces for versions' probe: *TimeProbe(LPA)* and *PGProbe(LPA, time)*. *TimeProbe* will return all the timestamps of versioned data using the *LPA*. Based on the *LPA*, LAST can query the address mapping table to acquire the PPA of the latest data. Then, LAST will query the CLT and OLT to get the timestamps of versions. Thus, LAST could return the timestamps to users through *TimeProbe*. Once users receive the timestamps of versions, they can query the versioned data through the interface *PGProbe* with the *LPA* and *time*. *PGProbe* will look at the L2P table to locate the latest data and then query the lineage tables (i.e., CLT and OLT) to return the versioned data to users based on the *time*.

Version rollback. Users may backtrack their data to a past version. Thus, LAST provides an interface - *RollbackPG(LPA, time)* - to enable the rollback of an individual data page. Specifically, LAST queries the mapping table to locate the latest PPA. Then, based on the PPA, LAST searches the CLT or OLTs using the *time* to find the target version. In the end, LAST updates the L2P table. We further propose an interface *RollbackAll(time)* to enable an SSD-wide rollback. When a user requests to recover the SSD to a prior time t after an attack, LAST needs to revert the entire disk to the previous time point t. First, LAST queries all the entries in the L2P table. Then, LAST traverses all of the versioned pages through CLT and OLTs to get the PPAs of the latest data pages before time t. Finally, LAST updates the L2P table with new PPAs to finish the rollback.

6.7 Implementation

We implement LAST on FEMU [61], a prevalent QEMU-based emulator widely used for SSD-related research [43, 47, 95]. In Table 1 [61, 78], the SSD consists of 512 GB flash memory and 500 MB DRAM (0.1% of flash memory [65]) where we allocate 200 MB for the metadata and 300 MB for the data cache. For the metadata, we allocate 100 MB for CMT [64] and 8 MB – sufficient for tracking the mappings of all the pages – for the GMD. Moreover, we allocate 16 MB [31] for the CHT, 14 MB for the GHD, 0.2 MB for the Overwriting Group List, and 1.3 MB for CLT. Thus, the total metadata introduced by LAST will be 31.5 MB, and we will validate their reasonability in Section 8.2. In addition, we allocate 16% data cache for the first-time cache, 16% data cache for the overwriting cache, and 68% data cache for the read cache. Moreover, we allow 10 entries in the OLT. We modify the FTL to create a backdoor, allowing users to send version management commands and receive results. Finally, we scale the SSD processor's clock rate to 0.5 GHz from our 3.2GHz host CPU by multiplying the table query time of I/Os in FTL with 6.4.

7 Security Analysis

Security Improvements. LAST advances security over existing in-device versioning in three ways: *(1)* LAST retains data upon its arrival in the storage device, whereas current versioning SSDs store data in their DRAM cache, which can lead to data loss. LAST thus provides more data versions to remediate system breaches. *(2)* LAST maximizes the SSD's ability to preserve versioned data as it avoids erasing versions with short retention time, which can significantly prolong the overall retention time, as shown in Section 8.5. For example, stealthy ransomware like Vipasana[4] writes minimal data, delaying detection. LAST's retention-aware design preserves data for extended periods, enabling recovery even after prolonged stealth operations. The longer fail-safe time window is critical for system recovery. *(3)* LAST avoids creating shattered files, as shown in Section 8.5, which ensures the integrity of versioned files for better recovery. In contrast, existing versioning SSDs can lead to shattered files, making their recovered data unusable, as discussed in Section 4.2.

Attacks on Versioning SSDs. Our LAST platform is resistant to compromise for the following reasons: *(1)* If attackers write and delete a large amount of data in a short time and force GC to overwhelm the SSD [79], the abnormal activities can be easily noticed by users [96] because the performance and available storage capacity will be significantly decreased, impeding normal applications. Note that we do not address anomaly detection in this work, as it is an active research area with extensive literature [27, 52, 57, 105], which is orthogonal to our data versioning target. *(2)* LAST achieves a long retention time of up to 126.4 days as discussed in Section 8.5. Malware often seeks to finish an attack quickly [45, 96]. If the malware attempts a "low and slow" attack of slow updates to the SSD, the retention time will remain high, leading to a high risk of being detected. *(3)* Flash memory capacity continues to increase due to new NAND flash technologies (e.g., QLC SSD [23]); for example, the largest SSD can be 100TB [17]. A large SSD makes the slowly-writing GC attack difficult as it will take an impractically long time to force GC.

How Long Should Historical Data Be Retained? In the worst-case scenario, attackers can remain undetected within a victim system for extended periods. This suggests that historical data should ideally be stored indefinitely. However, due to limited storage capacity and associated costs, it is not feasible to maintain infinite data versions. Therefore, LAST seeks to maximize historical data protection within storage capacity constraints. To achieve this, LAST employs an approximate-optimal data versioning strategy that avoids removing recently retained historical data, focusing instead on reclaiming data that have been stored the longest. In addition, to mitigate the retention limitations of currently versioning SSDs, employing higher-capacity SSDs can be beneficial – particularly given the emergence of more affordable, large-capacity SSD devices [17, 23].

Broader Security Engagement. LAST benefits from trusted computing primitives. For example, *(1)* to ensure the authenticity of SSD's firmware, a trusted platform module [40] can be integrated into the SSD to ensure the integrity of SSD firmware before boot, as it can maintain cryptographic keys securely and provide remote attestation. *(2)* Trusted execution environments (TEE) could provide an extended trusted computing boundary for complex operations in the SSD. Thus, integrating TEE with LAST can provide advanced version management using file-level semantic information [105].

[4] Hashtag: 8d2c4c192772985776bacfd77f7bc4d9.

Table 2: The characteristics of evaluated traces.

	Name	ID	Write Ratio	Daily Write
MSR	hm_0	M1	73.7%	2.9 GB/Day
	prxy_0	M2	96.9%	7.7 GB/Day
	rsrch_0	M3	90.7%	1.5 GB/Day
	wdev_0	M4	79.9%	1 GB/Day
	mds_0	M5	88.1%	1 GB/Day
FIU	mail	F1	58.8%	141.5 GB/Day
	web	F2	78.6%	1.8 GB/Day
	homes	F3	99.1%	3.8 GB/Day
ALI	dev_1	A1	99%	4.4 GB/Day
	dev_2	A2	99%	11.6 GB/Day
	dev_3	A3	95%	11 GB/Day

LAST can be integrated into forensic workflows, supporting the analysis of system activity. For example, in the event of a cyberattack, LAST preserves historical filesystem metadata (e.g., inodes) and data content for a long time in a trusted manner without being compromised by privileged attackers. This can be used by security administrators to reconstruct the chain of the attack and identify its root cause after the occurrence of the attack.

8 Evaluation

This section answers the following research questions: **(Q1)** How much space will versioning metadata consume? **(Q2)** What is the performance of LAST? **(Q3)** How does LAST affect the lifetime of the SSD? **(Q4)** How long can the versions be retained in LAST? **(Q5)** Can LAST eliminate or alleviate shattered versions? **(Q6)** How efficient the version manager is? We evaluate the space consumption of metadata **(Q1)** in Section 8.2. LAST's performance **(Q2)** is assessed in Section 8.3. The impact on SSD lifetime **(Q3)** is examined in Section 8.4. Finally, we answer **(Q4)**, **(Q5)**, and **(Q6)** in Section 8.5.

8.1 Experimental Setup

Environmental Setup. We use an Intel Xeon E3-1245 v5 @ 3.50GHZ 8-core processor with 64GB DRAM. Ubuntu 20.04.5 with kernel 5.13.4 is deployed as the host OS. In addition, we allocate a 50GB QCOW2 image file and install Ubuntu 18.04 along with kernel 4.15.0 to build a guest system on FEMU; we allocate 4GB DRAM to the guest with four vCPUs.

Comparison Selection. We implement multiple current regular and versioning SSDs for comparison.

(1) Baseline. This is a regular FEMU SSD equipped with LRU [49] policy without cache versioning.

(2) Baseline-BGE. This is a variant of Baseline with the background eviction, similar to LAST.

(3) LAST variants. LAST-NoRCA is a variant of LAST without the read cache. Similarly, LAST-NoBGE is created by removing background eviction, and LAST-NoDedup is obtained by disabling the deduplication feature.

(4) Versioning, BVSSD, and TimeSSD. Since current versioning SSDs overlook cache versioning, we borrow LAST's interleaving and background eviction strategies to an unmodified FEMU SSD (Versioning), BVSSD [46] and TimeSSD [96] for a fair comparison. All incoming data writes are first served by the DRAM cache in a logging fashion without removal. Finally, we re-implement the design of BVSSD and TimeSSD in the flash memory as follows. BVSSD sets a global GC threshold to monitor the ratio of free pages instead of at the chip level while using a greedy algorithm to select data blocks during GC. We implement TimeSSD Bloom filters (BFs) using an open-source library [2]. Each BF stores 100,000 invalidated PPAs with 0.01% false positives rate and occupies 234KB of memory [1]. We allocate 64MB of DRAM – as in TimeSSD – for BFs and compression buffers, allowing up to 280 BFs and tracking 106.8GB of invalidated data. Moreover, we implement TimeSSD to compress data during GC at a 20% [96] ratio using LZF [5] and use a greedy algorithm for block selection during GC. Finally, the SSD parameters are consistent with LAST, including 300MB data cache and 200MB metadata.

Workloads and Evaluation Method. We run experiments using the FIO benchmark [3] and real-world workloads from MSR [70], FIU [56], and Alibaba [12], as detailed in Table 2. The Alibaba traces provided enough data to trigger GC. However, the write sizes from MSR and FIU are too small to initiate GC. Thus, we duplicate MSR and FIU traces with an incremental offset (2GB) to the addresses at each duplication [96].

To evaluate performance (Figure 10 and Figure 14), we first fill the cache using FIO with a sequential write workload. Then, we run 1 million traces as a warm-up before conducting formal experiments, maintaining a 10:1 ratio of warm-up to formal traces [104]. However, the warm-up traces do not trigger GC in the experiments of Figure 14. We run the duplicated MSR and FIU traces and the full Alibaba traces to make the SSD close to full before starting the warmup-formal performance evaluations. For other experiments (i.e., Figure 16-**??**) that require GC, we employ only the duplicated traces and the full Alibaba traces. Note that all experiments run for four times.

Deduplication Setup. We use BLAKE2 hash algorithm and set the hashing latency to 10us, consistent with previous deduplication SSDs [98]. We evaluate the ratio of detected replicated data in the FIU workloads as the deduplication ratio, and their ratios of *mail*, *web*, and *homes* are 89.3%, 58.3%, and 33.5%, respectively. Then, we use the deduplication ratios when running the FIU traces. For MSR and ALI traces that do not have hash information, we set their deduplication ratio to a lower value (20%), matching the compression ratio in TimeSSD [96].

8.2 Metadata Size Testing

We evaluate only the reference counts of FIU in Figure 9, as other traces do not include hash fingerprints. Over 96% of fingerprints in FIU traces have reference counts of no more than nine. Since an OLT supports 10 entries, it is sufficient for storing lineage. Moreover, the OOB area consumes a maximum of 280 bytes for the OLT, fitting within the typical 409B size (10% of a flash page [62]). Thus, the OOB area is adequate for storing lineage metadata. In the worst case, if LAST directs the data to a flash page with a full OLT (i.e., 10), the page will not undergo deduplication and will instead be written to a free page.

Current versioning SSDs use BFs to track data invalidation [79, 96], requiring substantial DRAM; for example, a 512 GB SSD needs 306.8MB DRAM [79, 96]. This requirement increases with larger SSDs as more invalidation information must be stored. To avoid overwhelming DRAM, prior works only reserve partial invalidations by allocating 64MB [96] to the BF. However, this leads to the loss of invalidation information, degrading the ability to reconstruct versioned data and harming availability. In contrast, LAST needs only

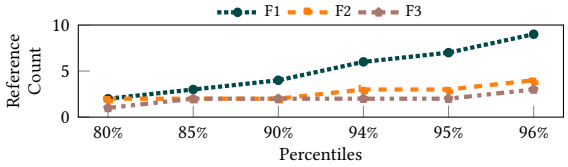

Figure 9: Reference count of FIU traces at tail percentiles.

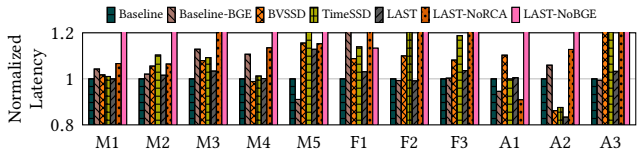

Figure 10: Normalized latency of Baseline SSDs, current versioning SSDs, and LAST.

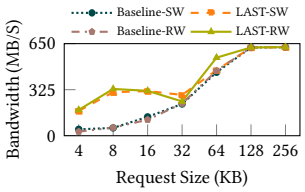

Figure 11: Write bandwidth of LAST at various request sizes.

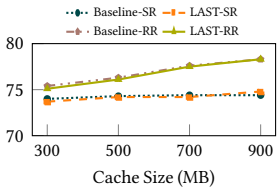

Figure 12: Read bandwidth of LAST at various cache sizes.

31.5MB of DRAM for metadata, as it directly records the invalidation sequence in the data layout of flash memory.

8.3 Performance Testing

Performance without GC Triggered. In Figure 10, LAST increases average latency over the Baseline SSD by 1.5%. LAST introduces trivial latency overhead for the following reasons. *(1)* LAST creates a read cache for incoming read requests. *(2)* OPEN proactively releases the first-time and overwriting caches in the background to avoid cache eviction. *(3)* Hash operations occur during GC, outside the critical I/O path. Baseline-BGE increases the latency over LAST by 9.3%, as Baseline-BGE proactively evicts data from the cache, decreasing read performance. Thus, current DRAM-cached SSDs typically do not adopt background eviction in their design [41, 64, 89]. Additionally, LAST-NoBGE increases latency by 139.3% over Baseline due to unavoidable data eviction from cache versioning. Without read cache (LAST-NoRCA), LAST's latency for the F1 (mail) workload increases over Baseline 41.9x. For other workloads, LAST-NoRCA increases latency over regular SSDs by 30%. These results indicate that the read cache is critical for SSD's performance, especially for workloads like F1 that have a higher ratio of read requests.

For versioning SSDs, we evaluate only BVSSD and TimeSSD, as BVSSD shares the same caching policy as Versioning, yielding identical performance without GC. Figure 10 shows that LAST decreases the average latency over BVSSD and TimeSSD by 7.7% and 10.3%, respectively. LAST achieves better performance due to its read caching region. Moreover, TimeSSD's Bloom Filter (BF) computations occur in the critical I/O path; the BF processing latency (1us) is amplified by the DRAM cache, leading to higher overhead.

Write Bandwidth. We evaluate the write throughput of LAST and Baseline when running sequential (SW) and random write (RW) workloads in FIO at various request sizes. Figure 11 shows that LAST achieves a better bandwidth than Baseline when request sizes are under 32KB. LAST performs better at lower incoming throughput. When the incoming throughput reaches the threshold (20% of the highest bandwidth), LAST stops the background eviction. Thus, LAST approximates the bandwidth of Baseline after 32KB.

Read Bandwidth. Figure 12 shows the bandwidth of LAST and Baseline when increasing the DRAM cache size. Since LAST creates a read cache, we evaluate the sequential read (SR) and random read (RR) workloads using FIO. LAST works efficiently when the SSD offers hundreds of megabytes of data cache, which is a practical size of modern SSDs [13, 18, 19, 90].

Cache Partition. We evaluate the data cache partition in Figure 13. Increasing the write cache ratio from 35% to 80% results in a 2.1% increase of random write (RW) bandwidth, indicating that larger write caches provide minimal performance benefits. This is because background eviction efficiently flushes data to flash memory. In contrast, random read (RR) bandwidth is decreased by 8.2% as the write cache expands, reducing the capacity of the read cache. This decrease becomes even more detrimental for workloads with a higher hit ratio. In F1 (mail), the access latency of LAST with a 35% write cache is 22.9% higher than with a 65% write cache. Thus, our implementation allocates the minimum required DRAM space to write caches, while reserving the majority of DRAM space for the read cache.

Performance during GC. In Figure 14, when GC is triggered, LAST incurs a latency overhead of 5.1% over the Baseline SSD. Baseline achieves better performance as it neglects cache versioning. However, LAST mitigates the overhead through background eviction and deduplication techniques. Moreover, the hash latency (10us) is far less than the latencies of flash write (200us) and erase (2ms). Thus, the hash overhead is trivial, and LAST incurs minimal overhead over regular SSDs.

Figure 14 also shows that LAST reduces the average latency over Versioning, BVSSD, and TimeSSD by 23.6%, 23.7%, and 72.7%, respectively. LAST outperforms existing versioning SSDs, as LAST reserves a read caching region for read requests, and it retains the spatial locality of versioned data in a data block, helping to concentrate versioned data pages into fewer flash blocks for less data migration. Additionally, LAST decreases the latency over LAST-NoDedup by 4.2%, as the deduplication can decrease the data migration during GC.

GC Execution Time. We evaluate the execution time of a GC operation in Figure 15. LAST increases the average GC latency over Versioning and BVSSD by 3.8% and 6%, respectively, but reduces it over TimeSSD by 94.1%. LAST does not significantly prolong GC execution because it maintains the spatial locality of data in flash memory, minimizing page migrations. Moreover, LAST's deduplication reduces page migrations during GC. Our results show that disabling deduplication (LAST-NoDedup) increases GC latency by 50.1%, demonstrating the effectiveness of our deduplication.

8.4 Lifetime Testing

Write amplification (WAF) is the ratio of data written inside storage to the data written by users, where a large WAF indicates a worse storage lifetime. Figure 16 shows that LAST increases average WAF over Versioning and BVSSD by 1.5%. Moreover, LAST-NoDedup increases

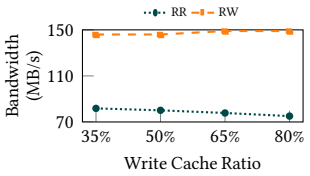

Figure 13: Bandwidth of LAST at different write cache ratios.

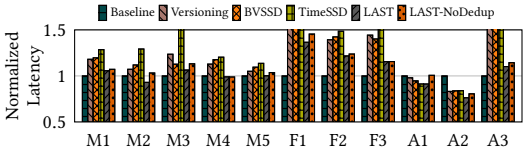

Figure 14: Normalized latency of Baseline, current versioning SSDs, and LAST during GC.

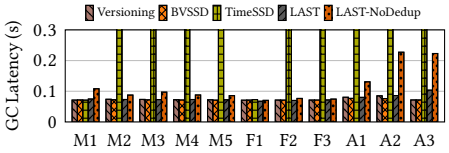

Figure 15: Average GC execution time of LAST and versioning SSDs during GC.

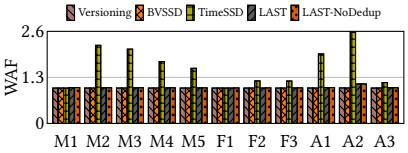

Figure 16: Write Amplification Factor (WAF) of LAST and versioning SSDs.

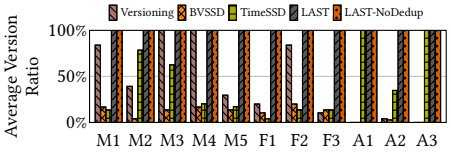

Figure 17: Ratio of available versions of created files in LAST and versioning SSDs.

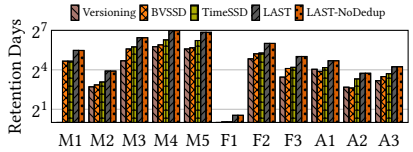

Figure 18: Average retention of erased pages in LAST and versioning SSDs.

the WAF value over LAST by 0.2%. LAST incurs trivial lifetime degradation. LAST can directly locate the flash block that contains expired data through the data *group* in the overwriting group list. Since data spatial locality is preserved in each data *group*, the expired data pages are likely to be maintained in fewer blocks. Thus, LAST achieves negligible lifetime overhead over BVSSD. Moreover, LAST decreases the WAF over TimeSSD by 37%. Since TimeSSD is not aware of the number of expired pages, which are not recorded by valid BFs, in selected blocks during GC, they may contain a significant amount of unexpired data, which should be migrated, burdening the SSD's lifetime.

8.5 Version Availability and Manager Testing

Shattered Version. We test the ratio of retained versions of 1,000 deleted files after GC in versioning SSDs and LAST. Using a method similar to that in Section 4.2, we split the trace into two halves. Then, we ran the first half, deleted the 1,000 files, and ran the second half. Notably, the traces do not access the created files. In Figure 17, both LAST and LAST-NoDedup maintain 100% versioned files, whereas Versioning, BVSSD, and TimeSSD only retain 42.8%, 10.1%, and 41.6% of versioned data, respectively. LAST reclaims versioned data based on the invalidation order, avoiding early deletion of versioned data. In contrast, existing versioning SSDs cannot track retention times, leading to premature removal of versioned files.

Retention Time. Figure 18 shows that LAST prolongs the average retention time of versioned data over Versioning, BVSSD, and TimeSSD by 165.9%, 96.5%, and 61.4%, respectively. LAST retains data by up to 126.4 days (M4) with an average of 52.6 days, whereas existing versioning SSDs can only achieve 19.8 days, 26.8 days, and 32.6 days on average, respectively. LAST avoids erasing versions with a short retention time. In contrast, BVSSD has no versioning policy to prolong the lifetime of deleted data. TimeSSD cannot track the invalidation sequence of each versioned flash page or infer the invalidation sequence of expired BFs due to limited DRAM capacity, as discussed in Section 8.2.

Version Manager. We evaluate the effectiveness of version manager in Table 3. TimeProbe, PGProbe, and RollbackPG operate at page granularity, and they can be performed quickly with a few milliseconds to finish per-page management. In M4 and M5 workloads, they have a much longer operating time because the queried data was updated many times (e.g., 27,000 times in M4 of TimeProbe), leading to heavy queries in the OOB area. Finally, LAST can rollback the SSD to a previous state in no more than 80 seconds through *RollbackALL*. These results show that LAST achieves fast version management.

Table 3: Execution time of LAST version manager.

	M1	M2	M3	M4	M5	F1	F2	F3	A1	A2	A3
TimeProbe (ms)	0.6	93.7	1	549.1	777.4	0.3	0.3	3.5	6.3	6.4	4.6
PGProbe (ms)	0.4	52.9	0.9	407.7	574.2	0.07	0.2	2.9	5.3	5.3	3.5
RollbackPG (ms)	0.4	66.9	0.9	453.3	643.5	0.03	0.3	2.1	5.3	5.3	3.4
RollbackAll (s)	31.2	20	49.2	36.3	37.8	14.4	19.3	29.3	49.3	49.3	78.4

9 Discussion

Consistency in the Data Cache. To ensure the consistency of the data stored in DRAM cache, modern SSDs typically employ power loss prevention (PLP) measures (e.g., supercapacitor [21]) to keep the DRAM cache powered upon a power failure for milliseconds [81] or even seconds [82] to flush data from the data cache to flash memory. Therefore, LAST adopts the existing PLP methods employed in modern SSDs to ensure data consistency in the data cache.

Caching Algorithms. The caching methods of SSDs are classified into write-through [90] and write-back [18, 41, 54, 64, 88, 89] policies. However, the write-back scheme is predominantly employed in SSDs for two reasons. *(1)* The write-through policy concurrently writes data into the data cache and flash memory, exposing the latency of the slow flash memory for data writes. In contrast, the write-back policy serves I/Os with low-latency DRAM on a cache hit. *(2)* The write-through policy degrades the lifetime of flash memory by writing data into it even when the data hits the data cache. Thus, LAST considers the write-back cache due to its prevalent use in modern SSDs.

Extensibility. LAST is compatible with other versioning techniques. For example, since LAST deduplication computes hash fingerprints with flash page granularity, we could deploy compression at page level as used in TimeSSD [96] to further decrease space consumption. Moreover, LAST can be used in RSSD to select versions with long

retention time sequentially and store them in the cloud storage instead of in the local SSD.

While LAST employs data versioning with the write-back policy, LAST can also be incorporated into the SSD equipped with the write-through data cache. Specifically, we can create two write pointers for incoming data, where each write pointer is assigned with a data *group* (see Section 6.2). Therefore, we can monitor the incoming data to write the first-time and overwriting data into the flash memory using two write pointers separately. We leave this as an extension for our future work.

Impact on Other FTL Modules. LAST modifies the caching and GC modules of the SSD's firmware. However, it remains compatible with other FTL functionalities. For example, LAST leverages current address mapping methods [42, 48] to manage the mapping table. Wear-leveling can operate at the granularity of data *groups* (see Section 6.2), and LAST uses the current wear-leveling approach [30] to distribute wear evenly across flash blocks. Finally, LAST is compatible with bad block management, which transparently replaces broken data pages with good ones [53] for address translation.

Limitations and Future Directions. While LAST significantly improves the availability of versioned data to alleviate the risk of data loss, it still suffers from the following limitations. *(1)* LAST maintains versioned data at the storage level, which lacks semantic context (e.g., provenance), limiting forensic utility. An interesting direction is to integrate semantic inference for enhanced recovery and forensic efficiency. *(2)* With limited storage capacity, retention time may degrade under heavy workloads. A promising direction is analyzing how real-world workloads affect retention and exploring integration with high-capacity storage (e.g., QLC SSDs).

10 Related Work

Host-level Versioning. Partial versioning retains a subset of data states such as using snapshots [10, 97], logging [80], or selective backups [32, 69, 93]. Ext3Cow [76] provides a file versioning and snapshot. Subramanian et al. [87] proposed ioSnap to efficiently snapshot system state within flash-based storage. However, they both preserve a limited data lineage and cannot eliminate the possibility of data loss.

Eidetic versioning [33, 46, 69] is a technology that can record and recall any past data. Peabody [68] is a full-versioning system for HDD that exposes the disk as an iSCSI target in the block layer. However, privileged attackers can destroy Peabody backups, and the drive cannot provide recovery service. Devecsery et al. developed an eidetic system [38] for hard drives to recover past data within an OS by leveraging information flows between processes. However, it strongly relies on the software stack, making it vulnerable to privileged attackers.

Device-level Versioning. Device-level methods deploy data versioning in the storage device. S4 [86] provides log-structured metadata versioning. However, it neglects the caching versioning and cannot track data retention. FlashGuard [45] prevents the deletion of potential victim data from privileged ransomware. However, it cannot protect data from non-read deletion, such as wiper attacks [8]. RSSD [79] offloads versions to remote cloud providers as they are cheaper and provide more storage space. However, it requires the SSD to provide in-storage ethernet, increasing financial cost. Moreover, the SSD price is continuously dropping [20], mitigating the cost of local SSDs compared to cloud storage. Finally, the cloud storage enlarges the TCB with the risk of data loss [16] and cannot be trusted in some cases [58].

11 Conclusion

LAST is an ordered versioning system that retains data history in the storage and allows users to manage the maintained versions securely. LAST considers DRAM in versioning SSDs and stores the overwriting data independently to track the invalidation sequence for the reclamation of GC. We evaluate LAST across multiple real-world workloads and compare it with existing SSDs; LAST ensures high availability of versioned data without significant overhead.

Acknowledgement

We would like to thank anonymous reviewers for their insightful feedback, helping us to improve our work. This work was partially supported by NSF CNS-1815883 and CNS-2055014.

References

[1] Bloom Filter Calculator. https://hur.st/bloomfilter/.
[2] C++ Bloom Filter Library. http://www.partow.net/programming/bloomfilter/index.html.
[3] Flexible I/O Tester. https://github.com/axboe/fio.
[4] How to Escalate Privileges in Linux, Privilege Escalation Techniques. https://systemweakness.com/how-to-escalate-privileges-in-linux-privilege-escalation-techniques-70c92499ae45.
[5] LibLZF. http://oldhome.schmorp.de/marc/liblzf.html.
[6] NILFS: Continuous Snapshotting Filesystem for Linux. https://nilfs.sourceforge.io/en/index.html.
[7] Understanding Privilege Escalation and 5 Common Attack Techniques. https://www.cynet.com/network-attacks/privilege-escalation/#:~:text=Privilege%20escalation%20is%20a%20type,gaining%20access%20to%20a%20system.
[8] New ZeroCleare Wiper Malware Used in Targeted Attacks. https://duo.com/decipher/new-zerocleare-wiper-malware-used-in-targeted-attacks, 2019.
[9] Iran May Deploy Wiper Malware in Response to U.S. Military Strike, Experts Warn. https://spectrum.ieee.org/tech-talk/telecom/security/iran-wiper-malware-cybersecurity-us-military-strike-news-experts-warning, 2020.
[10] Monitoring Snapshot Space Consumption with Pure Storage FlashArray. https://blog.purestorage.com/purely-technical/monitoring-snapshot-space-consumption-with-pure-storage-flasharray/, 2020.
[11] Trim/deallocation and garbage collection: The science of reclaiming ssd storage space. https://www.atpinc.com/blog/how-trim-ssd-works-to-free-storage-space, 2020.
[12] Alibaba Block Traces. https://github.com/alibaba/block-traces, 2022.
[13] 980 PRO PCIe 4.0 NVMe SSD 1TB. https://semiconductor.samsung.com/us/consumer-storage/internal-ssd/980pro/, 2023.
[14] CVE - CVE. https://cve.mitre.org/index.html, 2023.
[15] Data retention periods and their impact in analyzing user behavior. https://www.smartlook.com/blog/data-retention-period-user-behavior-analysis/#:~:text=Data%20retention%20and%20data%20retention%20periods,-In%20analytics%20tools&text=Typically%2C%20the%20minimum%20data%20retention,data%20retention%20timeframe%2C%20the%20better, 2023.
[16] Is Cloud Storage Data Loss Possible? According to Microsoft, Yes It Is. https://weareproactive.com/cloud-storage-data-loss-is-possible/, 2023.
[17] Largest SSDs and hard drives of 2023: the biggest internal, portable and external storage devices you can buy. https://www.techradar.com/best/large-hard-drives-and-ssds, 2023.
[18] Samsung 870 QVO SATA 2.5 SSD. https://semiconductor.samsung.com/consumer-storage/internal-ssd/870qvo/, 2023.
[19] Samsung 970 EVO Plus SSD Review: More Layers Brings More Performance. https://www.tomshardware.com/reviews/samsung-970-evo-plus-ssd,5608.html, 2023.
[20] Sorry Storage Makers But Falling SSD Prices Show No Signs Of Slowing Down This Summer. https://hothardware.com/news/falling-ssd-prices-show-no-signs-slowing-down, 2023.
[21] Supercapacitors have the power to save you from data loss. https://www.theregister.com/2014/09/24/storage_supercapacitors/, 2023.
[22] What is a USB security key, and how do you use it? https://www.tomsguide.com/news/usb-security-key, 2023.
[23] What Is QLC SSD? https://www.purestorage.com/knowledge/what-is-qlc-flash.html, 2023.

[24] Ssd security firmwares features tech brief. https://www.datasheetarchive.com/whats_new/1c1a884377ab1954f2efc54b614636ec.html, 2024.

[25] Jinwoo Ahn, Junghee Lee, Yungwoo Ko, Donghyun Min, Jiyun Park, Sungyong Park, and Youngjae Kim. Diskshield: A data tamper-resistant storage for intel sgx. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2020.

[26] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. https://www.blake2.net/blake2.pdf, 2013.

[27] SungHa Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and DaeHun Nyang. SSD-Insider: Internal Defense of Solid-State Drive against Ransomware with Perfect Data Recovery. In *38th International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[28] M. Bozyigit and M. Wasiq. User-level process checkpoint and restore for migration. *SIGOPS Operting Systems Review*, 2001.

[29] Christopher Burgess. Contractor hacks former employer, destroys and corrupts data. https://www.csoonline.com/article/564205/contractor-hacks-former-employer-destroys-and-corrupts-data.html, 2018.

[30] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*, 2007.

[31] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A Content-Aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[32] Ann Chervenak, Vivekanand Vellanki, and Zack Kurmas. Protecting file systems: A survey of backup techniques. 1998.

[33] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. Wayback: A user-level versioning file system for linux. In *USENIX Annual Technical Conference (USENIX ATC)*, 2004.

[34] CORVUS. Q4 Travelers' Cyber Threat Report: Ransomware Goes Full Scale. https://www.corvusinsurance.com/blog/q4-2024-travelers-cyber-threat-report/, 2025.

[35] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[36] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Annual Technical Conference (USENIX ATC)*, 2010.

[37] Peter J Denning. The locality principle. *Communications of the ACM*, 2005.

[38] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[39] Clare Duffy. Colonial Pipeline attack: A 'wake up call' about the threat of ransomware. https://www.cnn.com/2021/05/16/tech/colonial-ransomware-darkside-what-to-know/index.html, 2021.

[40] Trusted Computing Group. Trusted Platform Module (TPM) Summary. https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/, 2025.

[41] Jiayang Guo, Yiming Hu, Bo Mao, and Suzhen Wu. Parallelism and garbage collection aware i/o scheduler with improved ssd performance. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[42] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[43] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[44] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *European Conference on Computer Systems (EuroSys)*, 2017.

[45] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K. Qureshi. Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware. In *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[46] Ping Huang, Ke Zhou, Hua Wang, and Chun Hua Li. Bvssd: Build built-in versioning flash-based solid state drives. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR)*, 2012.

[47] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. Improving the reliability of next generation SSDs using WOM-v codes. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022.

[48] Song Jiang, Lei Zhang, XinHao Yuan, Hao Hu, and Yu Chen. S-ftl: An efficient address translation for flash memory by exploiting spatial locality. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, 2011.

[49] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha. LRU-WSR: integration of LRU and writes sequence reordering for flash memory. In *IEEE Transactions on Consumer Electronics*, 2008.

[50] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. Viyojit: Decoupling battery and dram capacities for battery-backed dram. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[51] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *25th USENIX Security Symposium (USENIX Security)*, 2016.

[52] Amin Kharraz and Engin Kirda. Redemption: Real-Time Protection Against Ransomware at End-Hosts. In *International Symposium on Research in Attacks, Intrusions, and Defenses, (RAID)*, 2017.

[53] Hong Seok Kim, Eyee Hyun Nam, Ji Hyuck Yun, Sheayun Lee, and Sang Lyul Min. P-bms: A bad block management scheme in parallelized flash memory storage devices. *ACM Transactions on Embedded Computing Systems*, 2017.

[54] Hyojun Kim and Seongjun Ahn. Bplru: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[55] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Improving SSD reliability with RAID via Elastic Striping and Anywhere Parity. In *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[56] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. In *8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[57] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. PayBreak: Defense Against Cryptographic Ransomware. In *15th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[58] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[59] Guy Laden, Paula Ta-Shma, Eitan Yaffe, and Michael Factor. Architectures for controller based CDP. In *5th USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[60] Sungjin Lee and Jihong Kim. Improving performance and capacity of flash storage devices by exploiting heterogeneity of mlc flash memory. *IEEE Transactions on Computers*, 2014.

[61] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[62] Qiao Li, Min Ye, Yufei Cui, Liang Shi, Xiaoqiang Li, Tei-Wei Kuo, and Chun Jason Xue. Shaving retries with sentinels for fast read over high-density 3d flash. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[63] Haodong Lin, Jun Li, Zhibing Sha, Zhigang Cai, Yuanquan Shi, Balazs Gerofi, and Jianwei Liao. Adaptive management with request granularity for dram cache inside nand-based ssds. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[64] Haodong Lin, Zhibing Sha, Jun Li, Zhigang Cai, Balazs Gerofi, Yuanquan Shi, and Jianwei Liao. Dram cache management with request granularity for nand-based ssds. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP)*, 2023.

[65] Weihua Liu, Fei Wu, Meng Zhang, Chengmo Yang, Zhonghai Lu, Jiguang Wan, and Changsheng Xie. Deps: Exploiting a dynamic error prechecking scheme to improve the read performance of ssd. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[66] Ningfang Mi, Alma Riska, Qi Zhang, Evgenia Smirni, and Erik Riedel. Efficient management of idleness in storage systems. *ACM Transactions on Storage (TOS)*, 2009.

[67] Microsoft. Microsoft Digital Defense Report 2022. https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE5bUvv?culture=en-us&country=us, 2022.

[68] C. B. Morrey and D. Grunwald. Peabody: the time travelling disk. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2003.

[69] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *3rd USENIX Conference on File and Storage Technologies (FAST)*, 2004.

[70] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[71] Nate Nelson. Iran APTs Tag Team Espionage, Wiper Attacks Against Israel & Albania. https://www.darkreading.com/threat-intelligence/iran-apts-tag-team-espionage-wiper-attacks-against-israel-and-albania, 2024.

[72] Security News. A new variant from Chaos Ransomware family surfaces. https://www.sonicwall.com/blog/a-new-variant-from-chaos-ransomware-family-surfaces, 2023.

[73] Nomios. What is SamSam ransomware? https://www.nomios.com/resources/what-is-samsam-ransomware/, 2024.

[74] Jonghyeok Park, Soyee Choi, Gihwan Oh, Soojun Im, Moon-Wook Oh, and Sang-Won Lee. Flashalloc: Dedicating flash blocks by objects. *Proceedings of the VLDB*

Endowment, 2023.

[75] Allan Parker and Glen Lam. Partial page programming of multi level flash, 2004. Patent No. US6836432B1, Filed Feb. 11th., 2002, Issued Dec. 28th., 2004.

[76] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transaction on Storage (TOS)*, 2005.

[77] Portnox. A Closer Look at NotPetya. https://www.portnox.com/cybersecurity-101/notpetya-attack/, 2017.

[78] Zhiwei Qin, Yi Wang, Duo Liu, Zili Shao, and Yong Guan. Mnftl: An efficient flash translation layer for mlc nand flash memory storage systems. In *Proceedings of the 48th Design Automation Conference (DAC)*, 2011.

[79] Benjamin Reidys, Peng Liu, and Jian Huang. Rssd: Defend against ransomware with hardware-isolated network-storage codesign and post-attack analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[80] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 1992.

[81] Samsung. Power loss protection (PLP): Protect your data against sudden power loss. https://download.semiconductor.samsung.com/resources/others/Samsung_SSD_845DC_05_Power_loss_protection_PLP.pdf, 2014.

[82] Samsung. To be? or Not to be? Hold up capacitors in 2.5" MIL SSDs. https://www.storagesearch.com/zero-to-three-hold-up-times-in-mil-ssds.html#:~:text=Power%20hold%20up%20time%20in%200%20to,in%20the%20event%20of%20sudden%20power%20loss., 2015.

[83] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin R. B. Butler. Cryptolock (and drop it): Stopping ransomware attacks on user data. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[84] Kai Shen, Stan Park, and Men Zhu. Journaling of journal is (almost) free. In *12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[85] Ryan Soliven and Hitomi Kimura. Ransomware Actor Abuses Genshin Impact Anti-Cheat Driver to Kill Antivirus. https://www.trendmicro.com/en_us/research/22/h/ransomware-actor-abuses-genshin-impact-anti-cheat-driver-to-kill-antivirus.html, 2022.

[86] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised system. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[87] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a flash with iosnap. In *9th European Conference on Computer Systems (EuroSys 2014)*, 2014.

[88] Hui Sun, Shangshang Dai, Jianzhong Huang, and Xiao Qin. Co-active: A workload-aware collaborative cache management scheme for nvme ssds. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[89] Hui Sun, Shangshang Dai, Jianzhong Huang, Yinliang Yue, and Xiao Qin. Dac: A dynamic active and collaborative cache management scheme for solid state disks. *Journal of Systems Architecture*, 2023.

[90] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A framework for enabling realistic studies of modern Multi-Queue SSD devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

[91] Arash Tavakkol, Pooyan Mehrvarzy, Mohammad Arjomand, and Hamid Sarbazi-Azad. Performance evaluation of dynamic page allocation strategies in ssds. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 2016.

[92] Chainalysis Team. 35% Year-over-Year Decrease in Ransomware Payments, Less than Half of Recorded Incidents Resulted in Victim Payments. https://www.chainalysis.com/blog/crypto-crime-ransomware-victim-extortion-2025/, 2025.

[93] L. Toka, M. Dell'Amico, and P. Michiardi. Online data backup: A peer-assisted approach. In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, 2010.

[94] Peiying Wang, Shijie Jia, Bo Chen, Luning Xia, and Peng Liu. MimosaFTL: Adding Secure and Practical Ransomware Defense Strategy to Flash Translation Layer. In *10th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2019.

[95] S. Wang, Z. Lin, S. Wu, H. Jiang, J. Zhang, and B. Mao. Learnedftl: A learning-based page-level ftl for reducing double reads in flash-based ssds. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.

[96] Xiaohao Wang, Yifan Yuan, You Zhou, Chance C. Coats, and Jian Huang. Project almanac: A time-traveling solid-state drive. In *14th European Conference on Computer Systems (EuroSys)*, 2019.

[97] Jake Wires and Michael J. Feeley. Secure file system versioning at the block level. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2007.

[98] Suzhen Wu, Jindong Zhou, Weidong Zhu, Hong Jiang, Zhijie Huang, Zhirong Shen, and Bo Mao. Ead: a collision-free and high performance deduplication scheme for flash storage systems. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020.

[99] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 2016.

[100] Weijun Xiao, Jin Ren, and Qing Yang. A case for continuous data protection at block level in disk array storages. *IEEE Transactions on Parallel and Distributed Systems*, 2009.

[101] Jeseong Yeon, Minseong Jeong, Sungjin Lee, and Eunji Lee. RFLUSH: Rethink the flush. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

[102] Tal Zamir. How to Detect and Prevent Dropper Malware Attacks. https://perception-point.io/guides/malware/understanding-dropper-malware-types-examples-detection-and-prevention/, 2024.

[103] Chijin Zhou, Lihua Guo, Yiwei Hou, Zhenya Ma, Quan Zhang, Mingzhe Wang, Zhe Liu, and Yu Jiang. Limits of i/o based ransomware detection: An imitation based attack. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[104] Weidong Zhu and Kevin R. B. Butler. Nasa: Nvm-assisted secure deletion for flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[105] Weidong Zhu, Grant Hernandez, Washington Garcia, Dave (Jing) Tian, Sara Rampazzi, and Kevin R. B. Butler. Minding the semantic gap for effective storage-based ransomware defense. In *Proceedings of the 38th International Conference on Massive Storage Systems and Technology (MSST)*, 2024.