Project Almanac: A Time-Traveling Solid-State Drive

Xiaohao Wang, Yifan Yuan, You Zhou, Chance C. Coats, Jian Huang
Systems and Platform Research Group
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
jianh@illinois.edu

Abstract

Preserving the history of storage states is critical to ensuring system reliability and security. It facilitates system functions such as debugging, data recovery, and forensics. Existing software-based approaches like data journaling, logging, and backups not only introduce performance and storage cost, but also are vulnerable to malware attacks, as adversaries can obtain kernel privileges to terminate or destroy them.

In this paper, we present Project Almanac, which includes (1) a time-travel solid-state drive (SSD) named TimeSSD that retains a history of storage states in hardware for a window of time, and (2) a toolkit named TimeKits that provides storage-state query and rollback functions. TimeSSD tracks the history of storage states in the hardware device, without relying on explicit backups, by exploiting the property that the flash retains old copies of data when they are updated or deleted. We implement TimeSSD with a programmable SSD and develop TimeKits for several typical system applications. Experiments, with a variety of real-world case studies, demonstrate that TimeSSD can retain all the storage states for eight weeks, with negligible performance overhead, while providing the device-level time-travel property.

CCS Concepts

• **Computer systems organization → Secondary storage organization**; • **Software and its engineering → File systems management**; **Secondary storage**; • **Security and privacy → File system security**.

Keywords

solid-state drive, time travel, firmware-isolated logging

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303983

ACM Reference Format:
Xiaohao Wang, Yifan Yuan, You Zhou, Chance C. Coats, Jian Huang. 2019. Project Almanac: A Time-Traveling Solid-State Drive. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303983

1 Introduction

Retaining past storage states enables users to recover detailed information about updates in storage systems. This is useful for many purposes, such as debugging, data recovery and rollback, forensics, and error tracking [6, 11, 38, 39].

Prior approaches rely on a variety of software-based techniques, such as journaling, logging, checkpointing, and backups. These techniques have become essential components of a vast majority of storage systems. However, they suffer from serious issues that significantly affect system reliability and security. To be specific, software-based solutions are vulnerable to malware attacks that can terminate the data backup processes or destroy the data backup itself. For instance, encryption ransomware can obtain kernel privileges and destroy data backups in order to prevent victims from recovering their encrypted data [14]. Furthermore, these software-based techniques inevitably impose storage overhead and additional I/O traffic [35, 39, 47].

A more lightweight and secure approach is desired, that retains the storage states and their lineage, without relying on software-based techniques, and without incurring overhead. In this paper, we present a time-travel SSD (TimeSSD) that transparently retains storage states in flash-based storage devices, and maintains the lineage of states. TimeSSD enables developers to retrieve the storage state from a previous time, creating a time-travel property. We also present a toolkit called TimeKits which supports storage-state query and rollback, for developers to exploit the firmware-isolated time-travel property.

Flash-based SSD is an ideal candidate for TimeSSD for four reasons. First, since flash pages cannot be written without being erased, modern SSDs perform out-of-place updates to reduce the overhead generated by expensive erase operations. The out-of-place update routine inherently supports logging functionality since old versions of data are not immediately erased [1, 10, 13]. Second, the SSD has to run garbage collection (GC) to reclaim obsolete data for free space. With a slight modification to the GC algorithm, we can reclaim

older versions of storage state in time order. Therefore, we can maintain the lineage of storage states in an SSD by keeping track of older versions of data. Third, the SSD firmware is isolated from the operating system (OS) and upper-layer software [2, 14]. The isolated firmware has a much smaller Trusted Computing Base (TCB) than upper-level systems software, which is less vulnerable to malware attacks [5, 38]. It provides a last line of defense for protecting user data, even if the host OS is compromised. Lastly, SSDs have become prevalent in a vast majority of storage systems driven by their significantly increased performance and decreased cost [37, 41].

TimeSSD is a lightweight firmware-level solution to fulfill the time-travel property in the hardware device. It balances the tradeoff between the retention time of past storage states and the storage performance by estimating the GC overhead. TimeSSD can dynamically adjust retention duration according to application workload patterns, while still enforcing a guarantee of a lower bound on the retention duration (three days by default). To further retain past storage state for a longer time, TimeSSD compresses the storage state during idle I/O cycles. Our study with a variety of storage workloads, collected from both university and enterprise computers, demonstrates that TimeSSD can retain storage state for up to eight weeks, with minimal performance overhead.

To facilitate the retrieval of storage states, TimeSSD leverages the available hardware resources in commodity SSDs, such as the out-of-band (OOB) metadata of each flash page to store the reverse mapping from physical page in flash chip to logical page in the file system. It uses back-pointers to connect the physical flash pages that map to the same logical page, which enables TimeSSD to maintain the lineage of storage states in hardware. All OOB metadata operations are performed in SSD firmware, they are also firmware-isolated from upper-level software. To accelerate retrieving storage states, TimeSSD utilizes the internal parallelism of SSDs to parallelize queries among multiple flash chips.

Based on this time-travel property, we develop TimeKits, a tool that can answer storage state queries in both backward (e.g., what was the storage state a few hours ago?) and forward (e.g., how has a file been changed since some prior state?) manner. These basic functionalities enable features, such as protecting user data against encryption ransomware, recovering user files, retrieving update logs, and providing an evidence chain for forensics.

Project Almanac utilizes this firmware-level time-travel property to achieve the same goals as conventional software-based solutions, in a transparent and secure manner, with low performance overhead. Overall, we make the following contributions:

- We present a time-travel SSD, named TimeSSD, which retains past storage states and their lineage in hardware, in time order, without relying on software techniques.
- We present a toolkit, named TimeKits, to exploit the firmware-isolated time-travel property of TimeSSD. It supports rich storage-state queries and data rollback.
- We quantify the trade-off between retention duration of storage states versus storage performance. We propose an adaptive mechanism to reduce performance overhead for TimeSSD, while retaining past storage states to a bounded window of time.

We implement TimeSSD in a programmable SSD, and develop TimeKits for a few popular system functions, such as data recovery and log retrieval of file updates. We run TimeSSD against a set of storage benchmarks and traces collected from different computing platforms. Our experimental results demonstrate that TimeSSD retains the history of storage states for up to eight weeks. TimeSSD fulfills these functions with up to 12% performance overhead. We also apply TimeSSD to several real-world use cases, and use TimeKits to retrieve past storage states or conduct data rollback , The performance results also show that TimeKits can accomplish the storage-state queries and data rollback instantly.

The rest of the paper is organized as follows: § 2 explains the background and motivation of this work. The design and implementation of Project Almanac are detailed in § 3 and § 4 respectively. We present the evaluation in § 5. We discuss the related work in § 6 and conclude the paper in § 7.

2 Background and Motivation

In this section, we briefly introduce the technical background of SSDs, then motivate our Project Almanac with a few real-world case studies.

2.1 Technical Background on SSDs

Solid-state drives are widely used in various computing platforms as they provide orders-of-magnitude better performance than HDDs with a price that is approaching that of HDDs [31, 37, 41, 44]. Rapidly shrinking process technology has also allowed us to boost SSD performance and capacity, accelerating their adoption in commodity systems.

An SSD has two major components, as shown in Figure 1: a set of flash memory chips, and an SSD controller. Each SSD has multiple channels, where each channel can independently receive and process read and write commands to flash chips. Within each chip, there are multiple planes. Each plane is further divided into multiple flash blocks, each of which consists of multiple flash pages.

Due to the nature of flash memory, SSDs can read and write only at page granularity. Furthermore, a write can only occur to a free page. Once a free page is written, that page is no longer available for future writes until that page is first erased and its state reset to free. Unfortunately, the erase operation can be performed only at block (which has multiple pages) granularity, and it is time-consuming. Thus, a write operation is issued to a free page that has been erased

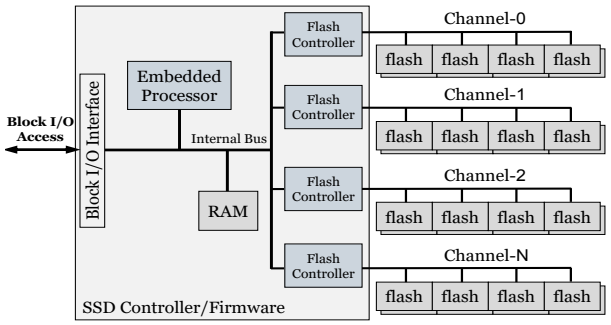


Figure 1. The architecture of an SSD.

in advance (i.e., out-of-place write), rather than issuing an expensive erase operation for every write. Because of out-of-place writes, an SSD employs indirections for maintaining the logical-to-physical address mapping. Out-of-place writes mean that each time a page is updated, the old version of the page is simply marked as invalid. And garbage collection (GC) will be performed later to reclaim it. The GC process runs once the number of free blocks is below a threshold in order to clean up invalid pages. The GC process will scan the SSD blocks, picking one candidate for cleaning. Then it will migrate any valid pages in the candidate block to a new free block, and when finished, it will erase the candidate block thereby freeing its pages for future writes. *An SSD inherently preserves past storage states, which can be exploited to implement the time-travel property within firmware.* We will discuss the details in § 3.

Since each block has limited lifetime, modern SSDs also have wear-leveling support to prolong SSD lifetime by evenly distributing write and erase to all blocks. To achieve wear-leveling, infrequently used block will be periodically moved so these blocks with low erase count can be used to reduce the stress of frequently erased blocks. The indirect mappings, wear-leveling and GC are all managed by the Flash Translation Layer (FTL) in SSD firmware.

To handle the FTL, SSD controllers are usually equipped with embedded processors and RAM. The embedded processors in SSD controllers help with issuing I/O requests, translating logical addresses to physical addresses, and handling garbage collection and wear-leveling. To exploit the massive parallelism of flash chips, the embedded processor may have multiple cores. For instance, the MEX SSD controller used in Samsung SSDs [33, 40] has a 3-core processor running at 400 MHz. The SSD controller has multiple channels and issues commands to perform I/O operations in different channels in parallel. *These hardware components available in modern SSDs provide the essential resources to meet the requirements (see details in § 3) of Project Almanac.*

2.2 Motivation Examples: Why TimeSSD

In this section, we will use several use cases to show the benefits of the hardware-isolated time-travel property provided by TimeSSD, and explain how it can be leveraged to provide better security for flash-based storage systems.

Storage Security: Durability of user data is a major design concerns in secure storage systems. To achieve this, storage systems have developed techniques such as journaling [29, 35] and data backups [7, 42] for decades. However, they still suffer from malware attacks today. If malware manages to gain administrator privileges, it can execute kernel-level attacks to destroy both local and remote data backups.

Take the WannaCry encryption ransomware for example. It quickly infected hundreds of thousands of computers across 150 countries in less than a week after it was launched in May 2017 [43]. Encryption ransomware like WannaCry destructively encrypts user files. To defend against this, prior work has proposed several detection approaches [19, 34, 46]. However, they kick in only after some data has already been encrypted. Furthermore, ransomware can use kernel privileges to destroy backups [14]. Therefore, software-based approaches do not offer sufficient protection.

Hence, it is natural to turn to a hardware-assisted approach. TimeSSD provides firmware-level time-travel property by retaining the past storage state in the storage device. Therefore, TimeSSD can recover user data, even after it has been encrypted or deleted by malware. The firmware-assisted approach adopted by TimeSSD has two advantages. First, it makes the time-travel property firmware-isolated to malware. Second, it has a much smaller TCB, compared to the OS kernel and other software-based approaches, and thus is less vulnerable to malware attacks.

Data Recovery in File Systems: An important functionality of file systems is to recover the system and user data upon system crashes and failures. To achieve this, software-based tools like data journaling have been proposed. However, journaling introduces a significant performance overhead, due to the extra write traffic [35].

TimeSSD transparently retains the lineage of the past storage states in SSDs. As TimeSSD retains the history of both metadata (e.g., *inode* in file systems) and data for a window of time, it allows developer to roll back a storage system to a previous state (at a specific time in the past) with minimal software involvement. Moreover, TimeSSD leverages the intrinsic out-of-place write mechanism in SSDs to retain past storage state, avoiding the redundant writes generated by the software-based journaling approach.

Storage Forensics: Digital storage forensics have been widely used in both criminal law and private investigation, such as criminal identification and the detection of unauthorized data modification [32]. Storage forensics need to reconstruct the original sequence of events that led to the incident. However, it is challenging to collect trusted evidence and to reconstruct the history of events. Due to insufficient evidence, incomplete recovery of events, or incomplete chronology of events, reconstruction of evidence may be incomplete or incorrect, which causes the failure of the investigation.

To make matters worse, existing storage forensics have little capability to mitigate an anti-forensics malware with OS kernel privilege.

TimeSSD provides trustworthy evidence for storage forensics softwares, even if malicious users try to destroy the evidence in the victim computer. In contrast to forensics software that executes with the OS, TimeSSD leverages the time-travel property in the hardware-isolated firmware, which facilitates the collection and reasoning of trusted evidence for storage forensics. Since TimeSSD is isolated from the OS, the evidence will be tamper-proof from malicious users.

3 Design of Project Almanac

In this section, we first discuss our threat model, then briefly introduce our goals. We then present an overview of the design followed by the details of how we achieve these goals.

3.1 Threat Model

As discussed in § 2.2, malicious users could try to elevate their privilege to run as administrators and disable or destroy the software-based data backup solutions. We do not assume the OS is trusted, but instead, we trust the flash-based SSD firmware. We believe this is a realistic threat model for two reasons. First, the SSD firmware is located within the storage controller, underneath the generic block interface in computer systems. It is hardware-isolated from higher-level malicious processes. Second, SSD firmware has a much smaller TCB compared to the OS kernel, making it typically less vulnerable to malware attacks. Once the firmware is flashed into the SSD controller, commodity SSDs will not allow firmware modifications, which guarantees the integrity of TimeSSD. In this work, we only consider the situation where data on persistent storage is overwritten or deleted.

Once abnormal events are recognized (e.g., malicious attacks are detected) or users want to recover data, they can use TimeKits to conduct the data recovery procedure. However, malicious programs can try to exploit the data recovery procedure to attack TimeSSD. For instance, a malicious user might roll back and erase all the data at a point, and then insert a large amount of junk data to trigger GC to erase the retained data copies. Project Almanac will defend against this attack. As TimeKits rolls back data by writing them back like new updates (see § 3.9), and TimeSSD retains the invalid data versions with a time window guarantee (three days by default, see § 3.4), TimeSSD still retains the recent data versions in the SSD. We will discuss more potential attacks in § 3.10.

3.2 Design Goals

Project Almanac consists of TimeSSD and TimeKits as shown in Figure 2. TimeSSD is a new SSD design that retains past storage states and their lineage over a window of time. TimeKits supports storage-state querying and data roll-back, to exploit the new firmware-isolated time-travel property provided by TimeSSD.

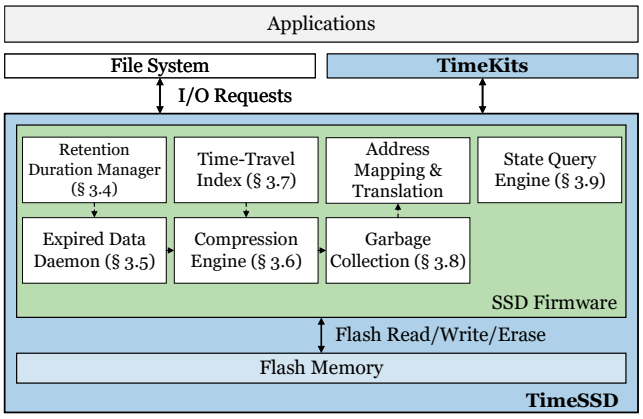


Figure 2. System overview of Project Almanac.

TimeSSD is designed with two goals. First, to provide a lightweight and firmware-isolated solution for retaining and retrieving past storage states. Second, to have comparable performance and lifetime to a regular SSD.

TimeKits should exploit the time-travel property to enable storage-state queries and file recovery.

3.3 TimeSSD Overview

TimeSSD leverages the intrinsic properties of flash SSDs (see § 2.1) to retain past storage state (or data versions) in the time order for future retrieval. Figure 2 provides an overview of the major components in TimeSSD. The *retention duration manager* maintains a *retention window* in a workload-adaptive manner (§3.4), whereas an *expired data daemon* is used to identify invalid data versions (i.e., old pages) beyond this time window (§3.5). Instead of reclaiming an invalid page immediately during GC, TimeSSD reclaims it after the retention window has passed. We use an invalid state marker to indicate when a page is deleted or updated but should still be retained. To be able to retain storage states for a long time, we compress retained data using the *compression engine* (§3.6). The *reverse index manager* maintains the reverse mapping of invalid data versions, to enable fast past state query (§3.7). The *garbage collector* is responsible both for cleaning expired data in order to recover free space, and for monitoring the GC overhead to provide feedback to the *retention duration manager* (§3.8). The *state query engine* can execute a variety of storage-state queries (§3.9) to exploit the time-travel property.

To facilitate our discussion, we first introduce the data structures used by traditional SSD firmware to support out-of-place updates and GC, as shown in Figure 3. The address mapping table (AMT) (1) translates the host logical page address (LPA) of an I/O request to a physical page address (PPA) in Flash. The AMT contains a mapping for each LPA

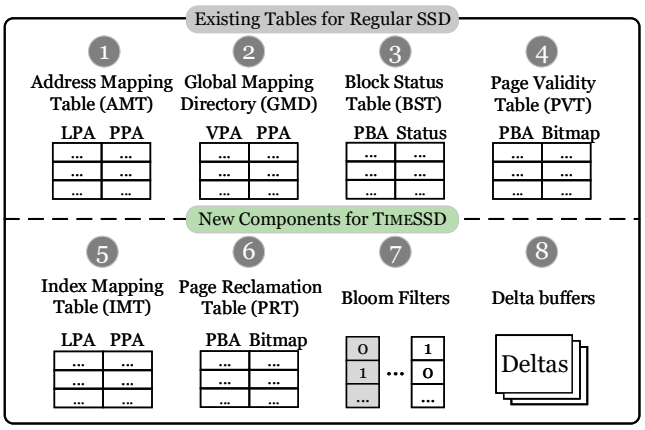


Figure 3. Data Structures in the RAM of TimeSSD. Tables are cached on demand if RAM resource is scarce. LPA: logical page address, PPA: physical page address, VPA: virtual page address, PBA: physical block address.

to its PPA on SSD. To accelerate this address translation, recently-accessed mappings are cached. The entire AMT is stored in Flash as a set of translation pages, whose locations are recorded in the global mapping directory (GMD) 2 [10]. The block status table (BST) 3 maintains status information at block granularity, and the page validity table (PVT) 4 maintains status information at page granularity. The BST tracks the number of invalid pages in each flash block, and the PVT indicates whether a flash page is valid or not.

Since SSDs only support out-of-place update, any data update or delete will cause the old version to be invalidated. Because writes can only happen to erased blocks, we need the GC process to clean up the invalid pages to make room for new writes. For a regular SSD, the GC process will first use the BST to select a candidate block (e.g., that with the lowest number of valid pages), and then check the PVT to migrate valid pages in the victim block to a free block. During migration, the AMT will be updated accordingly. After the migration, the GC process will erase the candidate block and mark it as a free block.

To support time-travel properties, TimeSSD requires minimal firmware modifications. It adds four data structures, as shown on the bottom of Figure 3. As data versions are compressed, the index mapping table (IMT) 5 is used to maintain the mapping of compressed invalid data versions (§3.7), which is responsible for translating an LPA to a PPA for compressed data pages. The page reclamation table (PRT) 6 is a bitmap to indicate whether a flash page can be reclaimed during GC or not (§3.6). Bloom filters 7 are used to identify whether an invalid page has expired or not (§3.5). The Bloom filters are also kept in time order to help us maintain the temporal ordering across invalid data versions. Delta buffers 8 group version deltas at page granularity in order to write the delta page back to the flash device (§3.6). In addition, TimeSSD slightly extends the BST 3 to mark the blocks storing deltas, (§3.8) and extends the GMD 2 to locate the IMT 5 in the flash device.

To enable the time-travel properties of TimeSSD, we modify the GC procedure to clean the expired and invalid pages. For the invalid pages, instead of reclaiming the free space immediately during the garbage collection (GC) process, TimeSSD reclaims invalid pages after the retention window has passed. We use the term *invalid* to indicate when a page is deleted or updated but should still be retained. Once an invalid page has passed its retention window, it will become *expired* and can be reclaimed by GC. The *expired data daemon* is used during GC to decide whether a page has expired. In addition, GC has a *compression engine* to compress invalid pages in order to further save storage space.

3.4 Retention Duration Manager

As discussed, since SSDs make out-of-place updates, updating or deleting data will invalidate their previous flash pages, which we retain for a time window. Retaining invalid pages incurs both storage overhead and performance degradation . When the number of retained invalid pages increases, the percentage of free space will decrease and GC must be triggered more aggressively. For each GC operation, as the proportion of retained invalid page increases, more retained invalid pages would need to be migrated to new flash blocks. As a result, GC operations will take longer, and SSD performance will degrade since it cannot respond to I/O requests during these operations. On the other hand, retaining as much invalid data as possible helps the time-travel property in TimeSSD. Therefore, there is a trade-off between retention duration and storage performance.

To ensure performance, TimeSSD dynamically adjusts the retention duration, using GC overhead as a proxy for storage performance degradation. If the estimated GC overhead per write (see details in §3.8) exceeds a given threshold, we reclaim some of the oldest invalid data. This reduces retention duration but improves SSD performance. A larger threshold results in lower SSD performance, but a longer retention duration. To fit with various workload patterns, TimeSSD dynamically adjusts the retention duration, in a workload-adaptive manner (see the details in §3.5). For example, when workload writes become more intensive, the retention duration decreases accordingly.

However, TimeSSD will guarantee a lower bound for the retention duration in order to avoid malicious attacks. We chose a default minimum duration of three days, but this is configurable by SSD vendors. If the free storage space runs out and the retention duration has not reached three days, TimeSSD stops serving I/O requests, resulting in the failure of file system operations. The users or the administrator will quickly notice this abnormal behavior. Even in this extreme

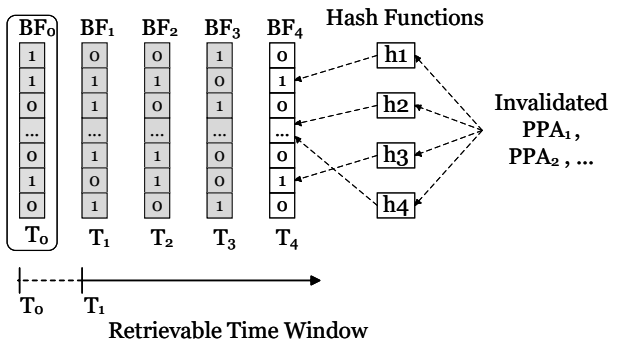


Figure 4. Retention window and Bloom Filters. Each Bloom Filter (BF) records a fixed number of Physical Page Addresses (PPAs) invalidated in a time segment. The retention window goes from the creation time of the oldest BF to the present. For example, after BF_0 was deleted, the time window is shortened from T_0 to T_1 and thus data pages invalidated during T_0 and T_1 become expired.

case, TimeSSD preserves the past storage state of the last few days, for further analysis.

3.5 Identifying Expired Data

Identifying expired data versions for space reclamation is essential for GC operation, but is challenging. In order to judge whether an invalid page is expired or not, a naive approach would be to maintain a table that records the invalidation time of flash pages. Assuming an SSD of 1TB capacity with 4KB pages and each timestamp occupies 4 bytes, this table will occupy 1GB, which cannot be fully cached in the RAM space of the SSD controller. Although it could be partially cached, like the address mapping table, cache misses would occur and introduce extra flash access overhead leading to reduced SSD performance.

To solve this problem in a space-efficient manner, TimeSSD employs Bloom Filters (BFs) (7) to record the invalidation time of flash pages, as shown in Figure 4. Whenever a data page is invalidated, its physical page address (PPA) is added to the active BF (i.e., the most recently created one). Once the number of PPAs in the current BF reaches a threshold, the active BF becomes inactive and a new active BF is created. Thus, each BF records a number of PPAs invalidated at approximately the same time, spanning from the BF's creation to its inactivation. Furthermore, we recycle BFs in the order of their creation. Thus, the retention window can be reduced simply by deleting the oldest BF. The retention window therefore starts from the creation time of the oldest BF to now.

By looking up the BFs, the garbage collector can identify whether an invalid page has become expired or not. If the PPA was found in one of the BFs, the corresponding page is retained. False positives could happen, in which case an expired page is retained because it was found in an active BF. This does not cause incorrect behavior. In contrast, there are no false negatives, *i.e.*, no non-expired pages will be recycled by mistake.

To reduce the number of BFs, TimeSSD exploits spatial locality. Flash pages in a block can only be written sequentially, and sequential writes are likely to result in sequentially invalidated pages. TimeSSD tracks invalidation at the granularity of a group, *i.e.*, N consecutive pages in a flash block, where N is user configurable (16 in our design). This way, each BF can accommodate more invalidated PPAs, and the number of BFs decreases. If any page in a group gets invalidated, the entire group is invalid in the BF. This will not cause incorrect behavior, because the GC algorithm will always verify that a page is expired before erasing it.

3.6 Delta Compression and Management

Content locality commonly exists between data versions mapped to the same logical page address (LPA) [45, 48]. For example, only a small portion of bits (typically 5% to 20%), called *delta*, are changed in a page update. TimeSSD exploits this property and uses delta compression to condense the obsolete data versions. Delta compression computes the difference between two versions of the same page and represents the invalid version with a compressed delta derived from the difference. Such a compressed delta is usually much smaller than a page, allowing us to save storage space for retained versions. When an obsolete version is selected for compression, the latest data version mapped to the same LPA is taken as a reference. Since obsolete versions are reclaimed in time order, a reference data version can never be reclaimed before all the corresponding deltas. Delta compression brings two advantages. First, retention duration is increased, because storage overhead is reduced. Second, the GC overhead is reduced, since fewer retained invalid pages need to be migrated during the GC procedure.

TimeSSD performs delta compression during GC operations and idle I/O cycles. During a GC operation, obsolete pages are compressed and migrated to new flash blocks. This could increase the GC overhead due to the extra compression operations. The increase in GC overhead could further degrade the SSD performance since no I/O operations can happen during GC.

To overcome this challenge, we exploit idle I/O cycles to perform background delta compression. As shown in prior studies, idle time between I/O requests commonly exists in real-world workloads [18, 22]. TimeSSD predicts the next idle time length ($t^i_{predict}$) based on the last interval of time between I/O requests ($t^{i-1}_{interval}$) with $t^i_{predict} = \alpha * t^{i-1}_{interval} + (1-\alpha) * t^{i-1}_{predict}$, where $t^{i-1}_{predict}$ refers to the time length of last prediction. We use the exponential smoothing method (with $\alpha = 0.5$) to predict idle time. Once the predicted idle time is longer than a threshold (10 milliseconds by default), TimeSSD

chooses a victim flash block, delta compresses the obsolete data pages, and marks the page that has been compressed or has expired as reclaimable in the page reclamation table (6) in Figure 3. Thus, future GC operations can directly reclaim these expired pages. When an I/O request arrives, TimeSSD immediately suspends background compression operations in order to remove the overheads of retaining obsolete data from the critical I/O path.

As a delta is usually much smaller than a page, we use delta buffers (8) to coalesce deltas to page granularity. When a buffer is full, or the remaining space is not large enough to accommodate a new delta, the delta buffer is written to a delta block. Each BF is associated with dedicated delta blocks for storing deltas corresponding to the pages that were invalid in a time segment. Note that a compressed data page might hit multiple BFs, due to false positives. TimeSSD checks the BFs (7) in reverse time order (*i.e.*, from the most recently created one to the oldest one). Once a hit occurs, the checking is stopped. This may delay the expiration of some data versions, but it avoids premature expiration.

TimeSSD stores deltas with different invalidation times separately for two reasons. The first is to maintain the reverse mapping for deltas which are mapped to the same LPA (see §3.7). Second is to improve the GC efficiency of cleaning expired data. When a BF is deleted to shorten the retention duration, the associated delta blocks contain all expired data versions and can therefore be erased immediately.

3.7 Time-travel Index

To achieve fast retrieval of previous data versions, the index manager of TimeSSD maintains a reverse index for each LPA. Each flash page has a reserved out-of-band (OOB) area for in-house metadata [10]. TimeSSD leverages this area to store: (1) the LPA mapped to this flash page, (2) the previous PPA mapped to this LPA, called a *back-pointer*, and (3) the write timestamp of this page. The back-pointer constructs the reverse mapping chain between different data versions for the same LPA. The write timestamp is used to identify different data versions of an LPA.

As discussed previously, some data versions are stored as deltas. Each delta contains the following metadata: (1) the LPA mapped to this delta, (2) the back-pointer, i.e., the PPA that contains the previous data version of this LPA, (3) the write timestamp of this data version, and (4) the write timestamp of the reference version (necessary for decompression). In a delta block, a page may contain multiple deltas. To retrieve these deltas, TimeSSD stores a header in each delta page. The header includes: (1) the number of deltas in this delta page, (2) the byte offset of each delta, and (3) the metadata of all the deltas.

We can build a reverse mapping chain with back-pointers between flash pages that store different data versions of an LPA. However, this index can be broken by GC operations. If GC happens to a page that is in the middle of the reverse

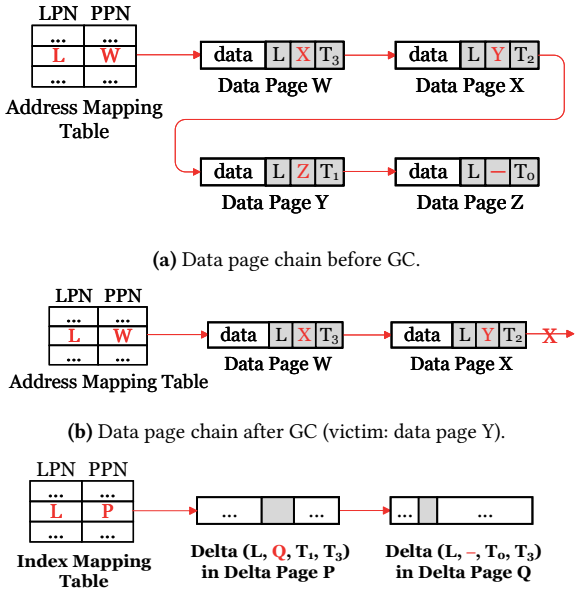


Figure 5. Reverse index of an LPA. (a) shows the reverse mapping chain before the data page chain is broken by a GC operation; (b) and (c) show the reverse mapping chain after data page Y is located and reclaimed. Each data page contains OOB metadata: LPA, back-pointer, and write timestamp. Each delta contains metadata: LPA, back-pointer, the write timestamp of itself, and the reference data version. '-' indicates a NULL pointer.

mapping chain, the page data will be migrated to a new flash page. The chain will be broken since the previous back-pointer will no longer point to the correct page.

To solve this problem, TimeSSD divides the reverse mapping chain of an LPA into two parts: the data page chain and the delta page chain, as shown in Figure 5. The data page chain contains uncompressed data versions that are newer than those (in compressed form) in the delta page chain. The PPA of the head page for each data page chain is recorded in the address mapping table (1), while the PPA of the head page for a delta page chain is recorded in the index mapping table (5).

The delta page chain is immune to GC operations, because reclaiming delta blocks involves no delta migration. When the data pages chain of an LPA is broken by GC, TimeSSD traverses the chain to find out all the unexpired versions up to the victim data page. For each page found, the OOB metadata will be used to verify that it has the correct LPA and a decreasing timestamp value. Then, the pages found during traversal plus the GC victim are compressed into deltas. When these deltas are written to delta pages, their back-pointers are set so that they are added to the head of the

Algorithm 1 GC procedure of TimeSSD

```
1:  Check the block status table (3)
2:  if there is an expired delta block then
3:      Erase this delta block
4:  else
5:      Select a victim data block
                                        ▷ the block with the most invalid pages
6:      Identify valid/invalid pages in the block page
                                        ▷ by checking the page validity table (4)
7:      for each valid page do
8:          Migrate this page to a free page
9:          Update the address mapping entry (1)
10:     for each invalid page do
11:         Check the page reclamation table (6)
12:         if this page is reclaimable then
13:             Discard this page
                                        ▷ this page has been compressed or expired
14:         else
15:             Checking this page in the bloom filters (7)
16:             if this page misses all the bloom filters then
17:                 Discard this page
                                        ▷ this page has been expired
18:             else
19:                 Read this page and its OOB metadata
20:                 Read the older and unexpired data versions
                                        ▷ through the data pages chain
21:                 Read the latest data version
22:                 Compress these old data versions
                                        ▷ the latest version as reference
23:                 Write back deltas to delta blocks with metadata
24:                 Update the head of the delta pages chain
                                        ▷ in the index mapping table (5)
25:                 Mark compressed data pages as reclaimable
                                        ▷ in the page reclamation table (6)
26:     Erase this data block
```

corresponding delta page chain. At the same time, the PPA of the head page for the delta page chain is updated in the index mapping table (5) for the target LPA. The data pages that were compressed are marked as reclaimable in the page reclamation table (6). As a result, the data page chain and the delta page chain compose a complete reverse mapping chain for an LPA.

3.8 Garbage Collection in TimeSSD

Since TimeSSD needs to retain invalid versions for each page, the GC algorithm shown in Algorithm 1 is different from traditional SSDs. Expired delta blocks are prioritized to be reclaimed, as no migration overhead is included. If no expired delta blocks are available, the data block that has the largest number of invalid pages is chosen (by checking the block status table (3)). A page is reclaimable if it is either an expired page whose PPA misses in all the Bloom Filters (7) or an invalid page that is marked as reclaimable in the page reclamation table (6). A page is not reclaimable if it is either a valid page or a retained page that needs to be compressed and then migrated during GC operations.

Besides performing GC operations, the garbage collector monitors and periodically estimates the GC overhead per user page write. The GC cost can be quantified by tracking the number of flash page reads (N_{read}), flash page writes (N_{write}), flash block erases (N_{erase}), and delta compressions (N_{delta}) in each period. A period refers to the time during which a fixed number of page writes (N_{fixed}) are issued.

$$\frac{N_{read} * C_{read} + N_{write} * C_{write} + N_{erase} * C_{erase} + N_{delta} * C_{delta}}{N_{fixed}} > TH * C_{write} \quad (1)$$

We assume the operation costs (in units of time) of a flash page read, a flash page write, a flash block erase, and a delta compression are C_{read}, C_{write}, C_{erase}, and C_{delta} respectively. We use the left side of Equation 1 as the estimation of the average GC overhead. This overhead measurement is then used to adjust the retention window (see §3.4). Specifically, once the average GC overhead exceeds a threshold (TH), 20% of the cost of a page write by default (the right side in Equation 1), retention window is shortened.

GC operations erase flash blocks with the largest number of invalid pages, so wear imbalance occurs when flash pages are not evenly updated. To balance the wear between flash blocks, wear leveling swaps cold data (i.e., rarely updated data) to old blocks (i.e., blocks that bear more erases) [12]. TimeSSD employs such a swapping technique for data blocks and handles the migration of retained invalid data pages in victim blocks like GC operations. Delta blocks are not considered, because we need to prevent the delta pages chain from being broken. The modification to the wear-leveling mechanism, has little impact on its effectiveness for two reasons. First, delta blocks are erased in the time order, so the wear imbalance between them is avoided. Second, the allocations of delta blocks and data blocks share the same free block pool, where wear balance can be achieved by the wear leveling of data blocks over time.

3.9 TimeKits: Storage State Query and Recovery

To enable the exploitation of the time-travel property provided by TimeSSD, TimeKits are developed for both state queries and recoveries. The basic functionalities/APIs can be classified into three categories, as shown in Table 1.

Address-based state queries. Given a single LPA or a range of LPAs, TimeSSD can utilize the reverse index to quickly retrieve any invalid data version within the retention window for each LPA. TimeKits provides three APIs for such state queries. `AddrQuery` can get the first data version(s) written since some time ago for a single LPA or a range of LPAs. For each LPA, we traverse the reverse index and retrieve each data version, one by one, beginning from the latest version. Each data version's writing time is checked, and the retrieval stops when a version's writing time reaches the target time, or the tail of the index is reached. On contrast, `AddrQueryRange` or `AddrQueryAll` is used to get all the data versions within a given time period or the entire retention window, respectively. Address-based state queries are useful to retrieve the invalid versions of a given file, whose LPAs can be obtained from the file-system metadata.

Table 1. The API for storage-state query in TimeKits.

API	Explanation
`AddrQuery(addr, cnt, t)`	Get the first data version(s) written since some time ago (specificed by t) for a single LPA ($cnt = 1$) or a range of LPAs ($cnt > 1$) started at *addr*.
`AddrQueryRange (addr, cnt, t1, t2)`	Get all the data versions written in a time period (between $t1$ and $t2$) for a single LPA ($cnt = 1$) or a range of LPAs ($cnt > 1$) started at *addr*.
`AddrQueryAll(addr, cnt)`	Get all the retained data versions for a single LPA ($cnt = 1$) or a range of LPAs ($cnt > 1$) started at *addr*.
`TimeQuery(t)`	Get all the LPAs that has been updated since some time ago (specified by t) and their writing timestamps.
`TimeQueryRange(t1, t2)`	Get all the LPAs that has been updated within a time period (between $t1$ and $t2$) and their writing timestamps.
`TimeQueryAll()`	Get all the LPAs that has been updated within the entire retention window and their writing timestamps.
`RollBack(addr, cnt, t)`	Roll back to the first data version(s) written some time ago (specified by t) for a single LPA ($cnt = 1$) or a range of LPAs ($cnt > 1$) started at *addr*.
`RollBackAll(t)`	Roll back to the first data versions written some time ago (specified by t) for all the valid LPAs.

Time-based state queries. TimeKits supports three kinds of time-based state queries, which return the write history of LPAs over time. `TimeQuery` can get the all the LPAs that have been updated since some time, and their writing timestamps (an updated LPA may have multiple writing timestamps). `TimeQueryRange` or `TimeQueryAll` perform such queries within a given time period, or the entire retention window, respectively. These queries are similar to address-based state queries except that all the valid LPAs are checked. To improve query performance, TimeKits leverages the parallelism of SSDs to maximize read throughput. Time-based state queries are useful for storage forensics.

State rollbacks. Besides state queries, TimeKits can roll back storage states to undo changes. `RollBack` reverts an LPA or a range of LPAs to a previous version. For each LPA, this is achieved by (1) retrieving a specified previous version through `AddrQuery`, (2) writing back this data version like an update to the LPA, and (3) invalidating the latest data version and modifying the address mapping entry. This way a state rollback is just a regular write with a previous data version. Future accesses will return the previous version, while all previous versions still remains recoverable. This API enables a lightweight rollback for a given LPA. `RollBackALL` rolls back all the valid LPAs to a previous version, is also supported. However, rolling back a large amount of LPAs is aggressive and causes a large volume of writes. This could significantly shorten the retention duration, or even cause failures due to the violation of the 3-day retaining guarantee for invalid data.

These state query APIs provide flexibility, and enable developers to efficiently and securely defend against encryption ransomware, recover data, and perform storage forensics. Previous file systems, such as Ext4 and XFS, write metadata changes to a journal, and rely on the journal to recover the LPAs. If the LPAs have been overwritten or the relevant journal records have been erased for space reclamation, these tools will fail to recover the target file. Our file recovery tools exploit the time-travel property of TimeSSD. The tool can, either retrieve erased journal records and invalid versions via address-based state queries, obtain the LPAs from the file system superblock and inode table, or restore the storage device to a previous state, via the state rollback functions discussed above.

3.10 Discussion

Project Almanac not only inspires us to rethink how we should build more efficient and secure storage systems, but also provides suggestions for SSD vendors how future flash-based SSDs can be architected. Consider the increasing density and decreasing cost of flash memory (*e.g.*, the pricing of today's SSDs is about $0.20/GB). This allows us to add more flash chips into future devices to retain more storage states for a longer period of time.

Retaining past storage states can prevent the secure deletion of sensitive data from the flash medium [21]. However, it is feasible to protect sensitive data from leakage in TimeSSD. We can use a user-specified encryption key to encrypt invalid data. This data can still be recovered by users, but can not be retrieved by others without the encryption key.

A malicious attacker may initiate attacks dedicated to TimeSSD. For instance, attackers might intensively write and delete junk files. However, it is noted that a deleted file is not physically deleted in the SSD until it is garbage collected. Therefore, the SSD will quickly become full, and TimeSSD will stop accepting I/O requests. This can be easily observed by end users.

Alternatively, an attacker might slowly write junk data. In this case, TimeSSD retention duration will increase accordingly since the workload is less write intensive. As shown in Figure 8, the retention duration can increase to up to 56 days. Many ransomware variants aim to lock up user data and collect ransom quickly to prevent from being caught. TimeSSD will significantly increases the risk of getting caught and can thus thwart malicious attackers from attacking our system.

As discussed in § 3.1, malicious users would also exploit the data recovery procedure of TimeKits to destroy the retained invalid data. Since TimeKits recovers user data by writing the invalid versions back to the SSD, the recent data versions could still be retained in the SSD. Moreover, TimeSSD will retain the invalid data versions within a time-window guarantee, even though malicious users keep writing junk data to the SSD. Therefore, we are still be able to

Table 2. The Workloads Used in Our Evaluation.

Name	Description
MSR [25]	The storage traces collected from enterprise servers.
FIU [9]	The storage traces collected from computers at FIU.
IOzone [16]	A benchmark for generating various file operations.
PostMark [17]	A file system benchmark that emuates mail servers.
OLTP [36]	An open-source database engine Shore-MT.

recover the recent storage states. To defend this attack, another simple approach is that users can remove the SSD and plug it into another trusted computer for data recovery.

4 TimeSSD Implementation

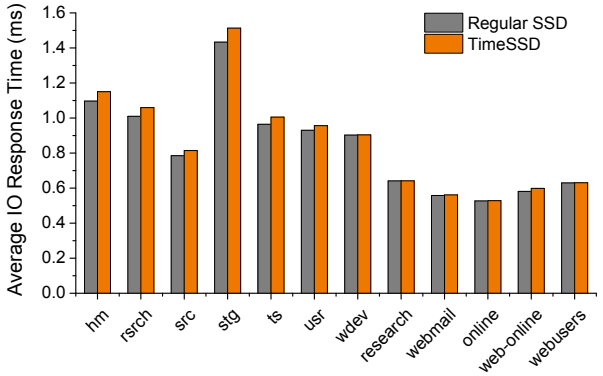
We implement TimeSSD on a Cosmos+ OpenSSD FPGA development board [27] running the NVM Express (NVMe) protocol. This board has a programmable ARM Cortex-A9 Dual-core and 1GB of DRAM. The SSD has a 1TB capacity and an additional 15% of the capacity as over-provisioning space. Each flash page is 4KB with 12 bytes of OOB metadata, and each block has 256 pages. Besides basic I/O commands to issue read and write requests, we define new NVMe commands to wrap the TimeKits API (shown in Table 1). TimeKits is developed atop the host NVMe driver which issues NVMe commands to the firmware running on the OpenSSD board. Inside TimeSSD, we slightly modify the NVMe command interpreter and add a state query engine into the SSD firmware for state query execution. We reserve 64MB of memory capacity in the firmware for the Bloom Filters and for the delta compression buffer. Implementing these functions adds 10,537 lines of code to the OpenSSD stack. TimeSSD adopts the page-level address translation [10]. We implemented delta compression with the LZF algorithm [23] because of its high speed.

5 Evaluation

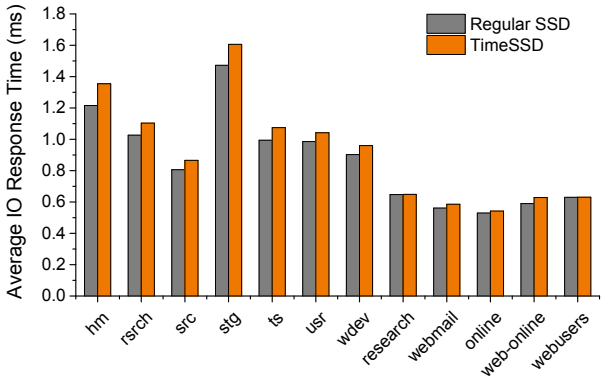
In our evaluation, we demonstrate that (1) TimeSSD has minimal negative impact on storage performance and SSD lifetime for data-intensive applications (§ 5.2). (2) It performs better than software-based approaches such as journaling and log-structured file systems, while providing firmware-isolated time-travel features (§ 5.3). (3) TimeKits performs fast storage-state queries for different systems functions (§ 5.4). (4) Two case studies to show that TimeSSD can restore user data to defend against encryption ransomware, and can provide data reverting for applications with low performance overhead (§ 5.5).

5.1 Experimental Setup

To evaluate the benefits of the firmware-isolated time-travel properties, we compare TimeSSD with a regular SSD. We use a variety of real-world storage traces, file system benchmarks, and data-intensive workloads, as shown in Table 2: (1) A set of week-long storage traces collected on enterprise servers running different applications, such as a media server, a research project management system, and a print server at Microsoft Research Cambridge [25]. (2) A set of storage traces for twenty days, collected on department computers at FIU [9]. (3) The IOZone benchmark, which generates a variety of file operations [16]. (4) The PostMark benchmark, which generates a workload approximating a mail server. (5) An open-source database engine, Shore-MT, running with a variety of transaction benchmarks, such as TPCC, TPCB, and TATP. The machine connected to the programmable SSD has a 24-core Intel Haswell based Xeon CPU, running at 2.6 GHz, with 32GB of DRAM. Before each experiment, we run variants of the file system benchmarks to warm up the SSD and to ensure GC operations are triggered.



(a) 50% capacity usage

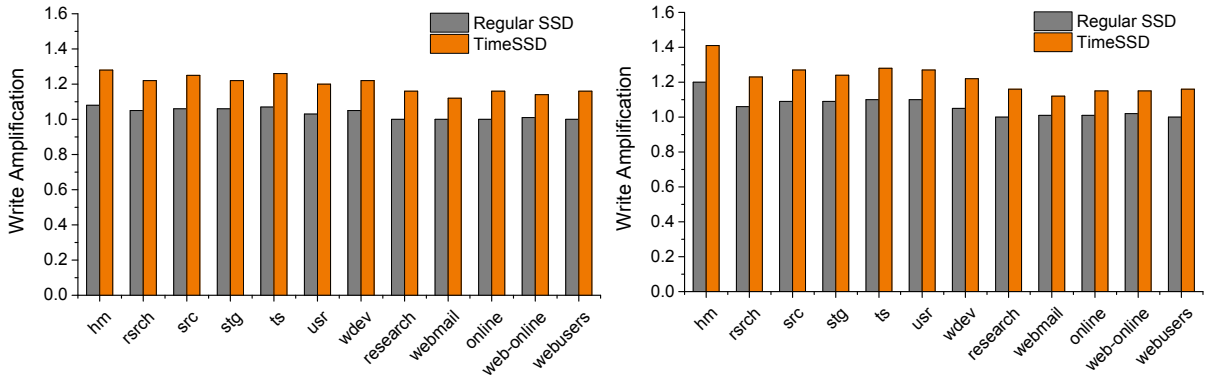


(b) 80% capacity usage

Figure 6. Average I/O request response time of running real-world storage traces with TimeSSD vs. regular SSD.

5.2 Comparison with Conventional SSD

We first evaluate the impact of TimeSSD on storage performance, device lifetime, and data retention duration, with MSR storage traces. In order to evaluate the device-level time-travel properties for a long period of time, we prolong each MSR trace to one month by duplicating it ten times. In each duplication, we mutate the trace by shifting the logical addresses of the I/O requests by a random offset. As reported in [48], the delta compression ratio values follow a Gaussian



(a) 50% capacity usage **(b)** 80% capacity usage

Figure 7. Write amplification of running real-world storage traces with TimeSSD vs. regular SSD.

distribution and the average ratio ranges from 0.05 to 0.25 for real-world applications. Since the MSR and FIU traces do not contain real data content, we use 0.2 as the default compression ratio for the experiments in this section. To understand the impact of storage capacity usage, we vary the utilization of the device from 50% to 80% of the total storage capacity of the SSD.

5.2.1 Impact on Storage Performance

Figure 6 shows that, compared to a conventional SSD, TimeSSD increases the I/O response time by an average of 2.5% under 50% storage capacity usage, and 5.8% under 80% space usage. As shown in Figure 6, the experimental results demonstrate that TimeSSD introduces minimal performance overhead for the MSR traces, while retaining the invalid data of the recent three days in the SSD. Since TimeSSD preserves more invalid data versions, it could migrate and compress more pages during the GC process, leading to a higher GC overhead. As we can see in Figure 6, the performance degradation at 80% capacity usage is larger than at 50%. This is because TimeSSD guarantees a lower bound of retention duration for invalid data. With a higher capacity usage, the GC will be triggered more frequently, which slightly decreases the storage performance of the SSD.

5.2.2 Impact on SSD Lifetime

As flash devices have limited lifetime (or endurance), it is necessary to evaluate the impact of TimeSSD on this aspect. We use write amplification, the ratio of flash write traffic to user write traffic, as the metric to evaluate SSD lifetime. A larger write amplification factor indicates a shorter lifetime. As shown in Figure 7, TimeSSD increases write amplification by an average of 10.1% under 50% capacity usage and 15.3% under 80% usage. This increase is mainly due to the migration of retained invalid pages during GC operations. However, compared to the software-based approaches, TimeSSD significantly reduces write amplification (see details in § 5.3).

5.2.3 Impact on Retention Duration

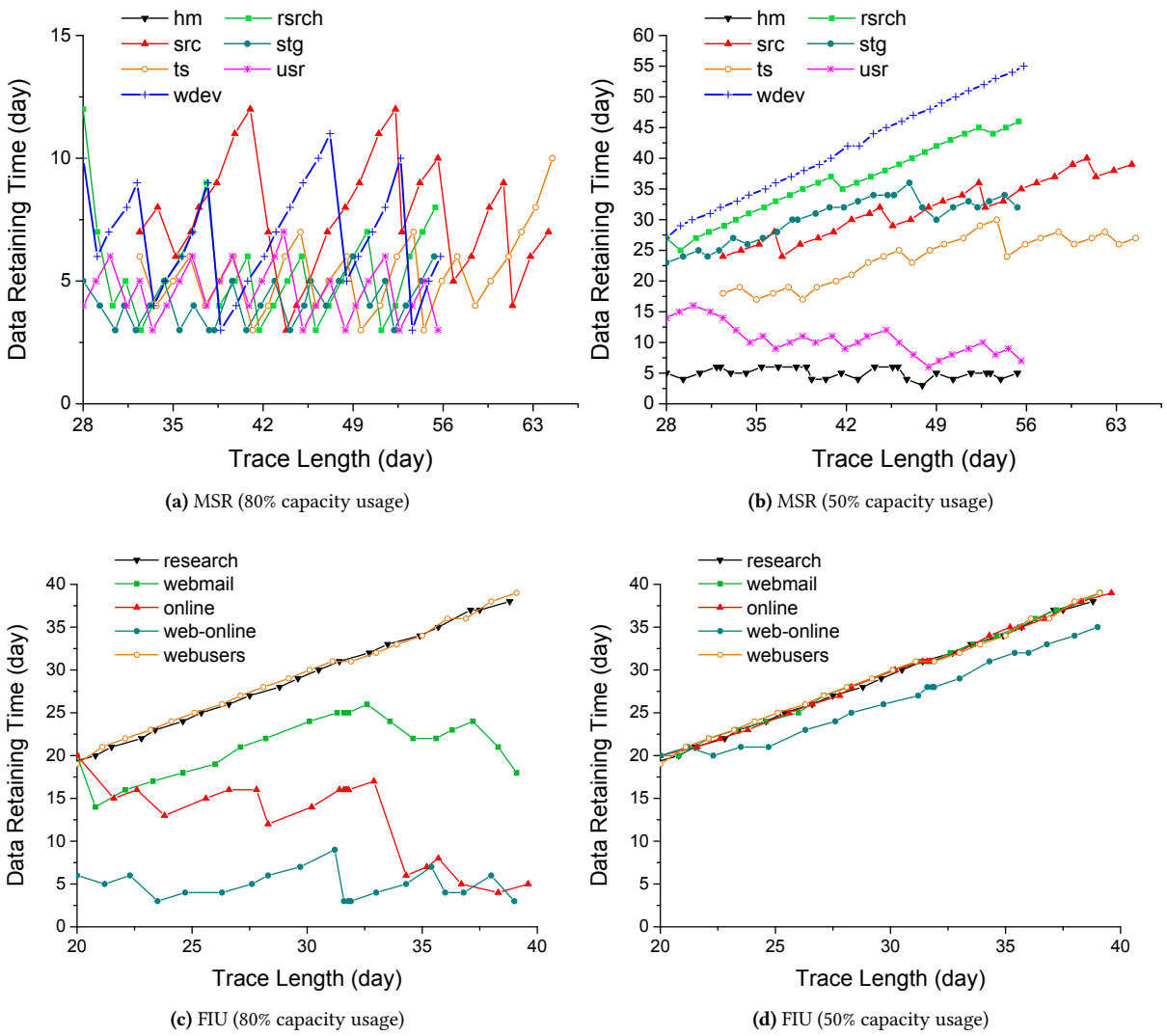
The retention duration of TimeSSD is determined by both device usage and workload write patterns. A high usage or a write-intensive workload degrades retention duration. Figure 8 shows retention duration under different workloads and capacity utilization. As we can see, the retention duration ranges from 3 days to 56 days. When the capacity usage decreases from 80% to 50%, retention duration is significantly improved because more storage space is available.

5.3 Comparison with Software-Based Approaches

We compare TimeSSD with the typical software-based solutions for tracking the history of storage states, such as journaling in the Ext4 file system and logging in the log-structured file system F2FS [20]. We ran file system benchmarks IOZone and PostMark that read/write real data content from/to files with 4KB granularity, and also an open-source database system Shore-MT [36] under the OLTP benchmarks TPCC, TPCB, and TATP. For our comparison, we run standard Ext4 and F2FS above a standard SSD (the OpenSSD board without the time-travel property), and we run Ext4 with journaling disabled for TimeSSD.

We first run the IOZone benchmark to generate various storage operations as shown in Figure 9 (a): sequential/ramdom read and write. As IOZone generates random values for the file content, the delta compression ratio of TimeSSD is almost zero. TimeSSD provides similar performance to Ext4 and F2FS for read operations. For sequential writes, the performance of TimeSSD is similar to Ext4 and F2FS, because there are no invalid pages retained in the SSD. For random writes, TimeSSD improves the storage performance by 3.3× compared to Ext4, since it avoids the additional write traffic introduced by journaling. TimeSSD performs slightly better than F2FS as it avoids the overhead of managing logs in software.

For the PostMark and OLTP benchmarks, which generate real application data, TimeSSD performs 1.5–2.2× and



(a) MSR (80% capacity usage)

(b) MSR (50% capacity usage)

(c) FIU (80% capacity usage)

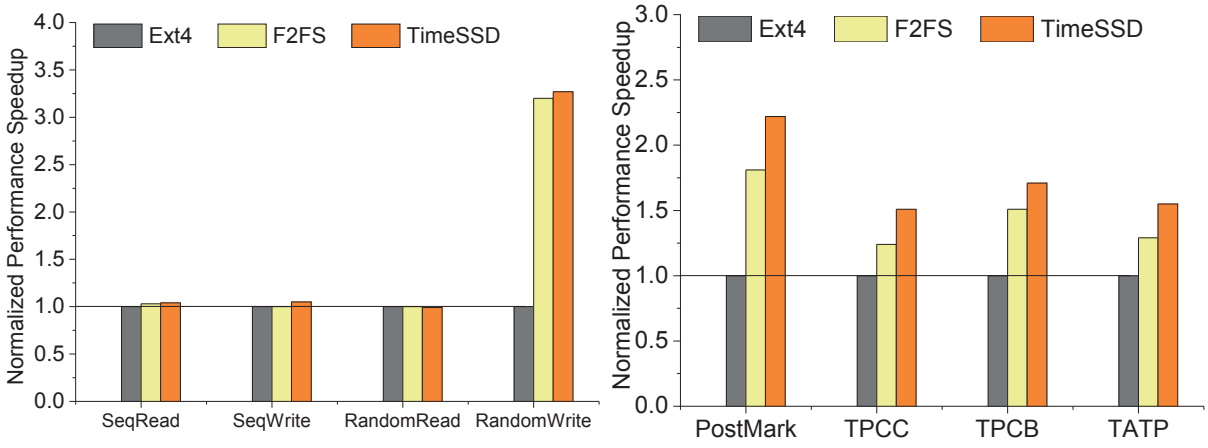
(d) FIU (50% capacity usage)

Figure 8. The data retention duration of TimeSSD under different workloads and capacity usages. For the applications running on university computers, the invalid data can be retained for up to 40 days; for the applications running on company servers, the invalid data can be retained for up to 56 days.

1.1–1.2× better than Ext4 and F2FS respectively (see Figure 9 (b)). Specifically, for the PostMark benchmark, TimeSSD outperforms Ext4 by 2.2×. For the OLTP benchmarks, executed with 16 threads, TimeSSD performs 1.5× (6.3K TPS), 1.7× (31.1K TPS), and 1.6× (122.3K TPS) better than Ext4 for TPCC, TPCB, and TATP respectively. Similar to the results reported by Lee. at al [20], F2FS performs 1.2–1.8× better than Ext4 by reducing the redundant writes of data journaling. As TimeSSD exploits the inherent logging feature in hardware, it reduces the write amplification by 3.4× and 1.3× on average compared to Ext4 and F2FS respectively. Moreover, TimeSSD compresses the data in SSD (with a compression ratio of 0.12–0.23) for more free space, which incurs less frequent GC operations and further improves the storage performance.

5.4 Performance of Storage-State Query

We now evaluate the effectiveness of querying storage states with TimeKits API. In each experiment, we first run the storage workloads (see § 5.2) to warm up the SSD, then execute three functions in the following order: `TimeQuery`, querying the storage states one day ago; `AddrQueryAll`, retrieving all retained data versions of a randomly selected LPA since one day ago; and `RollBack`, rolling back the selected LPA to a previous version. We show their performance in Table 3. Since a `TimeQuery` operation needs to scan all the valid LPAs to identify recently updated ones, it consumes



(a) IOZone **(b)** PostMark and OLTP

Figure 9. Performance speedup of running file system benchmarks and real OLTP workloads with TimeSSD vs. software-based approaches like Ext4 and F2FS.

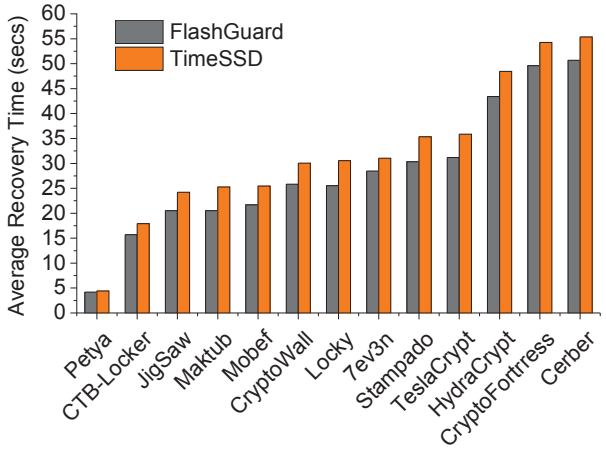


Figure 10. Recovering the data that have been encrypted by a variety of ransomware families.

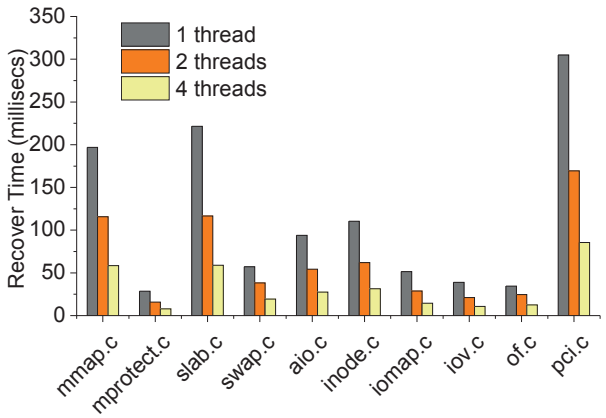


Figure 11. Reversing OS files to previous versions.

about 734 seconds (12 minutes). In contrast, as performing an `AddrQueryAll` operation requires only a few flash page reads and delta decompression operations, it takes a few milliseconds. These experiments demonstrate that the basic storage-state queries can retrieve invalid data versions or conduct data rollback quickly.

5.5 Case Studies

In the following, we use two real-world use cases to show that developers can leverage TimeKits to fulfill interesting system functions.

5.5.1 Recovering from Ransomware Attack

We compare our solution with another SSD named FlashGuard [14] which was developed specifically for defending against encryption ranomware. Unlike TimeSSD, FlashGuard only retains the invalid data (potential victim data) that was encrypted by ransomware. We gathered 13 encryption ransomware samples from VirusTotal and ran them on TimeSSD. Once the ransomware pops up a ransom screen to indicate the computer is attacked, we start our procedure to restore the encrypted user data. As we can see in Figure 10, TimeSSD can restore the data encrypted by ransomware in less than one minute. This is because most encryption ransomware will finish its attack quickly (75 minutes as indicated in [14], and only a limited volume of data updates are made to the SSD. Compared to FlashGuard, TimeSSD introduces 14.1% more performance overhead on average, due to the data decompression. This could be further optimized with hardware acceleration. We wish to implement the hardware acceleration as future work.

Table 3. Execution time of querying storage states.

	hm	rsrch	src	stg	ts	usr	wdev	research	webmail	online	web-online	webusers
`TimeQuery` (*seconds*)	764	740	734	758	734	746	728	710	722	716	728	722
`AddrQueryAll` (*milliseconds*)	5.5	1.1	1.1	1.3	1.1	0.8	0.7	0.3	6.6	0.3	5.5	0.6
`RollBack` (*milliseconds*)	6.3	2.1	2	2.3	2.1	1.7	1.7	1.4	7.6	1.2	6.3	1.5

5.5.2 Reversing File Changes

To further evaluate the time-travel property of TimeSSD, we develop a data-recovery function (similar to the revert function in GitHub) based on the APIs provided by TimeKits. We download the Linux kernel version 4.16.7. and replay its 1,000 most recent commits in 10 minutes (we commit 100 patches per minute) to the files in the code base. After that, we roll back some of the files to the version of one minute prior. We manually verify the content of the files and confirm that TimeSSD reverses the file contents to its correct version. Figure 11 shows that when we use more threads to do the data rollback, the data recovery time is dramatically reduced, as TimeSSD can leverage the internal parallelism of SSDs to enable multi-threaded data recovery.

6 Related Work

Retaining Storage State. Retaining storage state is useful for data recovery, auditing, and storage forensics. It has traditionally been done through software-based approaches, such as versioning [24, 26, 28, 49], snapshotting [8], and data backup [3]. However, these software-based approaches are vulnerable to malware attacks. They can be disabled or terminated by malicious users with kernel privileges. In contrast, TimeSSD presents a firmware-isolated solution using which the past storage state can be retained, even if the host OS is compromised. Devecsery et al. proposed an eidetic computer system [6] that uses hard disk drives to preserve the entire state of a computer system. Our Project Almanac focuses on retaining the storage states and their lineage in the hardware device transparently. TimeSSD can be used as the storage for the eidetic system.

Securing Storage Systems. As discussed, malicious users can acquire kernel privileges to tamper with, delete, and encrypt user data. To defend against these attacks, several secure storage systems are proposed. Huang et al. [14] proposed FlashGuard to defend against encryption ransomware to keep old versions of victim data. However, it does not retain all the past storage states. Strunk et al. [38] proposed a self-securing storage system, called S4, that uses log-structured and journal-based metadata to preserve old data versions for post-intrusion analysis and data recovery. S4 was implemented as a network-attached hard disk drive with an object-based interface. BVSSD [15] developed block-level versioning in SSDs, however, it cannot understand the relationships between blocks and thus cannot guarantee data consistency. With file system support, the consistency issue can be addressed, but these software-based solutions are vulnerable to malware attacks. TimeSSD achieves the same security goals, by presenting a firmware-isolated solution with minimal performance overhead.

System Software and Hardware Co-design. Modern SSDs have increased computing and memory resources, making it feasible to implement software functionalities inside the hardware device. Prior work has exploited the co-design of systems software and hardware for various purposes, such as journaling support [4], transaction support [30], and snapshotting [39]. They demonstrate that it is both practical and reasonable to exploit the SSD hardware for better performance. However, none of them took storage security as a serious concern in their design. In this work, TimeSSD exploits the intrinsic properties of flash to implement time-travel properties in hardware. It presents a firmware-isolated design, which can enhance the storage security significantly.

7 Conclusion

We present Project Almanac which can transparently retain past storage states and their lineage in flash-based storage devices with a low performance overhead. We develop its core idea into a time-travel SSD named TimeSSD that provides a firmware-isolated solution without any modifications to system software and applications. We also implement a toolkit called TimeKits to exploit the time-travel property of TimeSSD and to bring the benefits of a variety of system features, such as storage-state query and data roll-back, to end users and administrators. Our evaluation demonstrates that TimeSSD can retain past storage states for about eight weeks with minimal performance overhead.

Acknowledgments

We would like to thank our shepherd Marc Shapiro as well as the anonymous reviewers for their insightful feedback and comments. This work was supported in part by NSF grant CNS-1850317. Jian Huang is supported by the NetApp Faculty Fellowship Award.

References

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-Mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within A Unified Memory-Storage Hierarchy. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Providence, RI.

[2] SungHa Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and Dae-Hun Nyang. 2018. SSD-Insider: Internal Defense of Solid-State Drive against Ransomware with Perfect Data Recovery. In *Proceedings of 2018*

IEEE 38th International Conference on Distributed Computing Systems (ICDCS'18). Vienna, Austria.

[3] Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. 1998. Protecting file systems: A survey of backup techniques. In *Proceedings of Joint NASA and IEEE Mass Storage Conference*.

[4] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. 2009. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *ACM Transaction on Storage* 4, 4 (Feb. 2009), 14:1–14:22.

[5] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of 25th USENIX Security Symposium (USENIX Security'16)*. Austin, TX.

[6] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14) (OSDI'14)*. Broomfield, CO.

[7] Dropbox. 2019. https://www.dropbox.com/?landing=dbv2.

[8] G. Duzy. 2005. Match snaps to apps. In *Storage, Special Issue on managing the information that drives the enterprise*.

[9] FIU Traces. 2010. http://iotta.snia.org/traces/390.

[10] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. 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 System (ASPLOS'09)*. Washington, DC.

[11] Ryan Harris. 2006. Arriving at an Anti-forensics Consensus: Examining How to Define and Control the Anti-forensics Problem. In *Proceedings of the Digital Forensic Research Conference (DFRWS'06)*. Lafayette, IN.

[12] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. Santa Clara, CA.

[13] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. 2015. Unified Address Translation for Memory-mapped SSDs with FlashMap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. Portland, OR.

[14] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K. Qureshi. 2017. FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'17) (CCS'17)*. Dallas, TX.

[15] Ping Huang, Ke Zhou, Hua Wang, and Chun Hua Li. 2012. BVSSD: Build Built-in Versioning Flash-Based Solid State Drives. In *Proceedings of 5th Annual International Systems and Storage Conference (SYSTOR'12)*. Haifa, Israel.

[16] IOzone Lab. 2016. http://www.iozone.org/.

[17] Jeffrey Katcher. 1997. PostMark: A New File System Benchmark. *Technical Report* (1997).

[18] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. 2008. Characterization of storage workload traces from production Windows Servers. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC'08)*. 119–128.

[19] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. 2016. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX.

[20] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST'15)*. Santa Clara, CA.

[21] Jaeheung Lee, Sangho Yi, Junyoung Heo, Hyungbae Park, Sung Y. Shin, and Yookun Cho. 2010. An Efficient Secure Deletion Scheme for Flash File Systems. *Journal of Information Science and Engineering* 26, 1 (2010).

[22] Sungjin Lee and Jihong Kim. 2014. Improving Performance and Capacity of Flash Storage Devices by Exploiting Heterogeneity of MLC Flash Memory. *IEEE Trans. Comput.* 63, 10 (2014), 2445–2458.

[23] LibLZF. 2008. http://oldhome.schmorp.de/marc/liblzf.html.

[24] C. B. Morrey and D. Grunwald. 2003. Peabody: the time travelling disk. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'03)*. 241–253.

[25] MSR Cambridge Traces. 2008. http://iotta.snia.org/traces/388.

[26] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. 2004. A Versatile and User-oriented Versioning File System. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04) (FAST'04)*. San Francisco, CA.

[27] Open-Source Solid-State Drive Project for Research and Education. 2017. http://www.openssd.io/.

[28] Zachary Peterson and Randal Burns. 2005. Ext3Cow: A Time-shifting File System for Regulatory Compliance. *ACM Transaction on Storage* 1, 2 (May 2005), 190–212.

[29] Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2005. Analysis and Evolution of Journaling File Systems.. In *USENIX Annual Technical Conference (USENIX ATC'05)*. Anaheim, CA.

[30] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. 2008. Transactional Flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08) (OSDI'08)*. San Diego, CA.

[31] Price Trends of SSDs and HDDs. 2018. https://pcpartpicker.com/trends/price/internal-hard-drive/.

[32] Sriram Raghavan. 2013. Digital forensic research: current state of the art. *CSI Transactions on ICT* (2013).

[33] Samsung. 2013. Samsung SSD 840 EVO Data Sheet. *White Paper* (2013).

[34] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. 2016. Cryptolock (and drop it): stopping ransomware attacks on user data. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE, 303–312.

[35] Kai Shen, Stan Park, and Men Zhu. 2014. Journaling of Journal Is (Almost) Free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. Santa Clara, CA.

[36] Shore-MT. 2014. https://sites.google.com/view/shore-mt/.

[37] SSD prices plummet again, Close in on HDDs. 2016. http://www.pcworld.com/article/3040591/storage/ssd-prices-plummet-again-close-in-on-hdds.html.

[38] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. 2000. Self-securing Storage: Protecting Data in Compromised System. In *Proceedings of the 4th USENIX Conference on Symposium on Operating System Design & Implementation (OSDI'00) (OSDI'00)*. San Diego, CA.

[39] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Snapshots in a flash with ioSnap. In *Proceedings of the European Conference on Computer Systems (EuroSys'14)*. Amsterdam, Netherlands.

[40] Ken Takeuchi. 2008. Solid-state Drive and Memory System Innovation. *Lecture* (2008).

[41] The best cheap SSD deals in May 2018. 2018. https://www.techradar.com/news/cheap-ssd-deals.

[42] Michael Virable, Stefan Savage, and Geoffrey M. Voelker. 2012. BlueSky: A Cloud-Backed File System for the Enterprise. In *Proc. 10th USENIX conference on File and Storage Technologies (FAST'12)*. San Jose, CA.

[43] WannaCry Ransomware Attack. 2017. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[44] Michael Yung Chung Wei, Laura M Grupp, Frederick E Spada, and Steven Swanson. 2011. Reliably Erasing Data from Flash-Based Solid State Drives.. In *Procceedings of 9th USENIX Conference on File and Storage Technologies (FAST'11)*. San Jose, CA.

[45] Guanying Wu and Xubin He. 2012. Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. Bern, Switzerland.

[46] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *Proc. 38th IEEE Symposium on Security and Privacy (Oakland'17)*. San Jose, CA.

[47] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*. Broomfield, CO.

[48] Qing Yang and Jin Ren. 2011. I-CASH: Intelligently Coupled Array of SSD and HDD. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*.

[49] Qing Yang, Weijun Xiao, and Jin Ren. 2006. TRAP-Array: A Disk Array Architecture Providing Timely Recovery to Any Point-in-time. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)*. 289–301.