Lethe: Secure Deletion by Addition

Eugene Chou
UC Santa Cruz
California, USA
euchou@ucsc.edu

Leo Conrad-Shah
UC Santa Cruz
California, USA
lconrads@ucsc.edu

Austen Barker
Sandia National Laboratories
California, USA
atbarke@sandia.gov

Andrew Quinn
UC Santa Cruz
California, USA
aquinn1@ucsc.edu

Ethan L. Miller
UC Santa Cruz / Pure Storage
California, USA
elm@ucsc.edu

Darrell D. E. Long
UC Santa Cruz
California, USA
darrell@ucsc.edu

Abstract

Modern data privacy regulations such as GDPR, CCPA, and CDPA stipulate that data pertaining to a user must be deleted without undue delay upon the user's request. Existing systems are not designed to comply with these regulations and can leave traces of deleted data for indeterminate periods of time, often as long as months.

We developed Lethe to address these problems by providing fine-grained secure deletion on *any* system and *any* storage medium, provided that Lethe has access to a fixed, small amount of securely-deletable storage. Lethe achieves this using *keyed hash forests* (KHFs), extensions of *keyed hash trees* (KHTs), structured to serve as efficient representations of encryption key hierarchies. By using a KHF as a regulator for data access, Lethe provides its secure deletion not by removing the KHF, but by *adding* a new KHF that only grants access to still-valid data. Access to the previous KHF is lost, and the data it regulated securely deleted, through the secure deletion of *the single key* that protected the previous KHF.

CCS Concepts: • **Security and privacy → Key management**; • **Information systems → Information storage systems**.

Keywords: Secure deletion, storage systems, key management, security, privacy

ACM Reference Format:
Eugene Chou, Leo Conrad-Shah, Austen Barker, Andrew Quinn, Ethan L. Miller, and Darrell D. E. Long. 2023. Lethe: Secure Deletion by Addition. In *3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '23), May 8, 2023, Rome, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3578353.3589541

CHEOPS '23, May 8, 2023, Rome, Italy

ACM ISBN 979-8-4007-0081-1/23/05.
https://doi.org/10.1145/3578353.3589541

1 Introduction

Today's storage systems are tasked with providing timely secure deletion. Timely secure delete requires ensuring that data is irrecoverable within short timescales that are outside the control of the system (ideally within a few seconds). The feature is increasingly important due to consumer protection laws (*e.g.*, the European Union's GDPR [5], California's CCPA [2], Virginia's CDPA [3], *etc.*), compliance with laws governing the protection of classified information [4], and ensuring proper data governance for customers [7].

Unfortunately, traditional data deletion (*e.g.*, `rm`, `unlink`) does not securely delete data, and existing secure delete techniques require impractical resource overhead or constraints. *Physical destruction* [15], in which a user drills, shreds, or melts a storage medium, requires destroying an entire drive for each deletion and leads to impractical resource overhead. *Overwrite erasure* [13, 15], which securely deletes data by overwriting it in-place, suffers from two issues: (1) it places impractical constraints on storage media because it only works on storage that supports in-place overwrites (thus precluding use of media such as flash [24] and write-once, read-many (WORM) media [10]) and (2) it places impractical resource overhead on storage devices because it repetitively overwrites data and thus degrades device durability.

Cryptographic erasure [15] eases some of the impracticality of physical destruction and overwrite erasure, but retains high resource overhead in either high storage or compute costs. Cryptographic erasure encrypts each chunk (anywhere from a block to an entire drive) of data in a storage system using a key and strong cipher; it supports secure deletion by performing overwrite erasure on the key. Since breaking the encryption is computationally infeasible (either through the key or the algorithm), cryptographic erasure ensures practical secure deletion given enough storage for its keys.

However, existing cryptographic erasure systems introduce impractical resource overheads due to the tradeoff between computation and storage in managing the encryption keys. On one extreme, a system could use a single key for all data blocks (coarse-grained). This approach requires little key storage, but imposes high compute overhead since it requires re-encrypting the entire drive for each deletion,

no matter the size. On the other extreme, a system could use a separate key for each data block (fine-grained). This approach eschews re-encryption, but imposes high storage overhead since it requires an amount of securely deletable key storage proportional to the amount of data on the drive.

This paper presents Lethe, a portable system for timely, fine-grained secure deletion with low storage and computation overhead. Lethe encrypts keys recursively as a tree in a *hierarchical structure*. This design reduces the required amount of securely deletable storage down to a single root key (typically 128–256 bits). This storage requirement is supported by ubiquitously deployed systems such as secure enclaves (*e.g.*, Apple's Secure Enclave [1], Intel's Software Guard Extensions (SGX) [6], *etc.*) and smart cards (*e.g.*, YubiKey [8]). Delegating secure deletion of a small, constant amount of key material to a trusted component allows Lethe to be agnostic of storage media since there is no requirement for in-place overwrite of data; all data, including metadata, is written *append-only*. Thus, Lethe reframes secure deletion not as *removing the data that is no longer wanted*, but as *adding data that only provides access to what remains valid.*

Hierarchical key management also enables Lethe to minimize the computational overhead required for each deletion. Lethe introduces the *keyed hash forest* (KHFs), an extension of a *keyed hash tree* (KHT) [18], which serves as an efficient data structure from which to derive and revoke keys. Lethe structures KHFs hierarchically: *inode KHFs* protect data blocks of a file, a *master KHF* protects the inode KHFs, and a single *master key* protects the master KHF. The master KHF acts as a data regulator in that it only allows access to data covered by its subordinate inode KHFs. When Lethe securely deletes data, it creates a new master key, "rolls forward" the still-valid keys in its master KHF by re-encrypting them with the new master key, and erases the previous master key. Thus, Lethe's compute overhead is proportional to the size of the master KHF rather than to the amount of data in the system. Lethe could further reduce compute overhead by adding additional layers to its KHF hierarchy.

To evaluate Lethe, we prototyped its design by integrating it into the Zettabyte File System (ZFS) [9]. The added capability for secure deletion in our unoptimized prototype results in a 17.63% decrease in throughput compared to baseline ZFS, and a 15.5% decrease in throughput compared to ZFS with native encryption (which encrypts each block of data but cannot provide secure deletion).

2 Background

Prior secure delete systems have shortcomings that prevent them from providing the privacy mandated by legislation (GDPR, CCPA, CDPA, *etc.*), as we will describe. Namely, prior systems do not support media without in-place updates, including WORM media and flash devices that use an FTL.

Reardon *et al.* [22] proposed *ballooning* and *purging* in user space, to address the inability to perform in-place overwrites on flash memory. Ballooning artificially reduces free space by occupying it with junk data, and purging periodically fills up the free space with junk data to ensure the secure deletion of deleted data blocks. However, these techniques adversely affect the endurance of flash memory, perform poorly, and do not work on WORM media.

Decrypting *stubs* [20] provide per-block encryption keys. Erasing the block's decrypting stub securely deletes the block. This design requires in-place overwrite and cannot be used on devices without such support (e.g., flash and WORM media).

File header blocks [17] provide per-file encryption. Consequently, file header blocks provide secure delete for files but not for individual blocks. Like the decrypting stub design, this design also requires in-place overwrite and thus cannot be used on devices without such support.

DNEFS [21] encrypts each data block and stores all data block keys in a group called a Key Storage Area (KSA). DNEFS batches block-level secure deletion using purging epochs. When a block is deleted or overwritten, DNEFS marks the block's key to be deleted in the next epoch. Purging epochs run periodically to *roll forward* keys that remain valid in a KSA by copying them to a new KSA. DNEFS overwrites the previous KSA in-place to securely delete the keys that weren't rolled forward to the new KSA. This system thus requires in-place overwrites and cannot be used on devices without such support.

No prior system provides portable secure delete, since they cannot be used on WORM or flash media. Consequently, prior work is inadequate for fine-grained secure deletion requirements mandated by legislation.

3 Key Management

We now discuss key management, contrasting different encryption granularities for cryptographic erasure and the amount of key storage overhead they each produce. By exploring how keys are stored, we demonstrate how storing keys in a hierarchical manner reduces the amount of data that needs to be securely erased for secure deletion of an arbitrary amount of data to that of a single key.

A cryptographic-erasure based system typically applies encryption at one of the following granularities, listed in order of decreasing granularity: full-drive, per-file, and per-block. *Full-drive encryption* encrypts every data block with the same key. While securely deleting all the data blocks only requires securely deleting that one key, there is the caveat that even securely deleting a single byte of that data requires all the data blocks to be re-encrypted with a new key. *Per-file encryption* encrypts every data block within a single file with the same key. This approach suffers from the same re-encryption issue as full-drive encryption when partially

deleting a single file, but less pronounced due to how much smaller files are in comparison to entire drives. *Per-block encryption* encrypts every data block with its own key, thus avoiding the re-encryption penalty paid by full-drive and per-file encryption.

The issue with the finer-grained encryption approaches, per-file and per-block, is the problem of *key storage*. Consider a relatively small file system with 4 TiB of data, a block size of 4 KiB, and 16 B encryption keys. Providing a key for each possible data block results in 16 GiB of keys.

A naïve approach for storing these keys is to simply store them in together in blocks, as would occur if they were stored as a key file. Assume key K resides in block B. Securely deleting K and preserving the other keys in block B requires writing a new version of the block, B', that replaces K with a new key K' and keeps around the still-valid keys. K is securely deleted when B is securely deleted. Thus, any change to a block requires 4 KiB worth of keys to be securely deleted, and deleting all the data blocks in the system requires all 16 GiB of keys to be securely deleted. This is equivalent to the approach used for the KSAs introduced by Reardon *et al.* [21].

A different approach would be to employ per-file encryption, protecting the key file by encrypting all of its blocks with a single key. While this approach requires all the blocks of the key file, the *key blocks*, to be re-encrypted with another key in order to delete any key in the key file, the only key that needs to be securely deleted is the single key protecting all the key blocks.

The next obvious approach is to instead employ per-block encryption on the key blocks, which makes the recursive nature of this problem apparent. Encrypting each of the key blocks with their own key results in more keys, these keys in which are also stored in key blocks, each of which should be encrypted as well. This continues until there is a single key encrypting a block of indirect keys (at however many levels of indirection).

Thus, due to the *hierarchical structuring* of keys, the single top-level key regulates access to *all* the data that is encrypted by any key that is indirectly encrypted by it. Secure deletion of the top-level key is *sufficient—and necessary*—for the secure deletion of *any amount* of data covered by that top-level key.

Lethe uses *keyed hash trees* to efficiently represent this hierarchy of keys, reduce the amount of key storage required for fine-grained, per-block encryption, and to reduce the amount of effort necessary to roll forward still-valid keys.

4 Keyed Hash Trees

A *keyed hash tree* (KHT), originally presented by Li *et al.* [18], is a logical tree structure that allows for an effectively infinite number of block encryption keys to be generated from a single key, with the property that it is *computationally infeasible* to derive block keys from each other.

The topology of a KHT is defined by a list of integers describing the fanout (the number of child nodes per parent) at each level. Crucially, the fanout list describes fanouts starting from level 1 (L1) because the root at L0 has an arbitrary fanout, allowing the generation of an unlimited number of keys from a single root.

4.1 Composition

Like typical tree structures, KHTs are composed of nodes. Each node is defined as a triple:

$$\text{Node } n = \langle \mathit{value}, \mathit{level}, \mathit{offset} \rangle.$$

The level and offset of a node acts as a unique identifier, where the level indicates the level, or depth, within the KHT that the node can be found in and the offset indicates its position within the level itself.

The value component of the node triple is suitable for use as an encryption key, which allows a KHT to supply an infinite number of keys since it may cover an infinite number of leaf nodes.

4.2 Key Derivation

Aside from the root, each node in a KHT is *derived computationally*, from its parent. Critically, this means that only the root node of a KHT must be stored in order for the whole tree to be accessible. To get the value v' of a node given its level l, its offset o, and its parent's value, v, we simply compute:

$$v' = H(v \,||\, l \,||\, o),$$

where H is a cryptographically secure hash function and $||$ indicates concatenation. The usage of a cryptographically secure hash function ensures that the relationship from a parent node to a child node is a strict one-way relationship, making it trivial to recursively compute any descendant node given its ancestor, but computationally infeasible to compute an ancestor node from any of its descendants. It is computationally infeasible to compute the values of sibling nodes as well, since that would require computing the parent's value. Figure 1 shows the relationship between connected nodes in a KHT described by a $\langle 3, 2 \rangle$ fanout list.

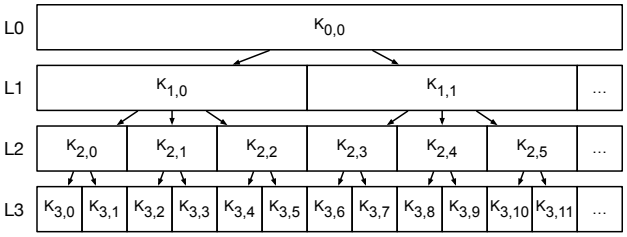

Figure 1. A KHT described by a $\langle 3, 2 \rangle$ fanout list.

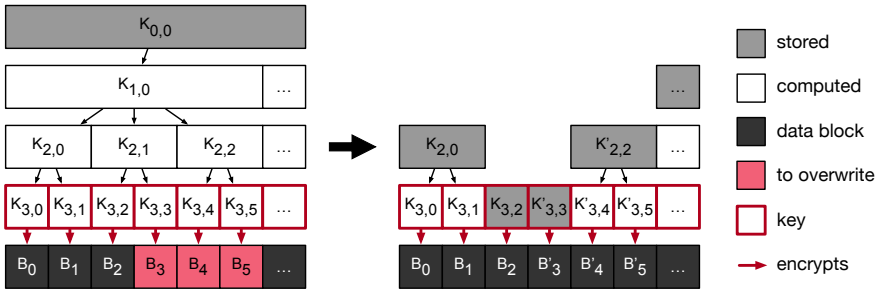

Figure 2. A KHT before and after key revocation.

4.3 Key Revocation

The primary difficulty with KHTs is *key revocation*, which was acknowledged by Li *et al.* [18] and stated to be future work. As seen with the KSAs used in DNEFS [21] and in the discussion on key storage in §3, deleting or modifying a block requires its key to be securely deleted—*revoked*—and the rest of the keys with which it was stored to be rolled forward. Revoking a single key in a KHT causes it to *fragment*, since the roots from which the revoked key can be derived must be rendered inaccessible. Consider Figure 2, which depicts the state of a KHT before and after revoking the keys for the to-be-overwritten data blocks B_3, B_4, and B_5. The new state of the KHT no longer includes the roots from which the revoked keys can be derived and includes new roots from which replacement keys can be derived. Thus, KHT key revocation fragments the KHT into a forest of KHTs. Lethe solves this using *keyed hash forests*.

4.4 Keyed Hash Forests

A *keyed hash forest* (KHF) describes a forest of KHTs from which encryption keys can be derived. A KHF is simply represented as lists of roots, where each root matches the definition of a node presented in §4.1.

There are two main operations associated with a KHF: *key derivation* and *update*. The key derivation operation, as its name implies, is used to derive a key using the roots stored in a KHF. Storing roots in a KHF in order of their offsets allows for efficient $O(\log n)$ search to find the root of the KHT that covers the desired leaf node.

The update operation is used to revoke keys from a KHF. It updates the roots in the KHF, replacing the roots covering the revoked keys with a new set of roots that provide replacement keys from which still-valid keys can be derived, but invalidated keys *cannot*.

KHFs naturally lend themselves to be used in a hierarchical manner, capable of both providing encryption keys for data blocks, as well as other KHFs. A single, top-level KHF thus effectively regulates access to all data covered by any subordinate KHFs protected by it. By extension, a single key encrypting this top-level KHF regulates access to the entire set of data that all of its subordinate KHFs cover.

Lethe provides secure deletion through addition, given a fixed, small amount of securely deletable storage used to store the key protecting the top-level KHF. Instead of overwriting a KHF, Lethe simply writes a new KHF that allows access to all data that should be kept. Any data that is not accessible through the new KHF is securely deleted when the key to the previous KHF is securely deleted.

5 Design

We now present the design of Lethe. We start with how Lethe uses *copy-on-write* (CoW) semantics not only to eliminate the need for in-place overwrites, but also to provide data consistency. We then describe the hierarchy of KHFs, assuming a traditional Unix file system that uses inodes.[1] Finally, we present the usage of *epochs*, time intervals over which KHFs updates are batched, and *consolidation*, an operation performed to address KHF fragmentation.

5.1 Copy-On-Write

All data, including metadata, is written in a copy-on-write (CoW) manner. This upholds Lethe's guarantee of secure deletion through addition, and provides portability. Since Lethe only writes new data in an append-only fashion, it is clear that *Lethe's operational security is not dependent on in-place overwrites.* Similarly, because data is explicitly assumed to never be overwritten, Lethe can be considered truly portable since it works agnostic to the storage medium it uses.

The usage of CoW enables data consistency. Once a new copy of data is written, pointers are atomically modified to point to the new copy instead of the old copy. This is the same approach that ZFS [9] uses for its data consistency.

5.2 KHF Hierarchy

Lethe maintains a KHF for each inode, which we refer to as an *inode KHF*. Each inode KHF is responsible for managing the encryption keys for the blocks that its corresponding

[1] Although the presentation of Lethe's design assumes a traditional Unix file system structure, we note that Lethe can be easily modified to allow for secure deletion on any system.

inode points to. Persisting an inode KHF requires it to be encrypted.

To manage the keys for inode KHFs, Lethe maintains a *master KHF*. Each inode KHF's key is identified within the master KHF by its i-number, and a *master key* is used to encrypt the master KHF when persisting it. The master KHF, and by extension the master key, regulates access to all data in the system. Securely deleting the master key securely deletes all data covered by the master KHF that has not been rolled forward to being covered by a new master key and KHF. Figure 3 shows an example of the described KHF hierarchy.

The hierarchy established by inode KHFs and the master KHF is no different from the tree structure established by directories in a traditional file system. While only two levels of KHFs are shown in Figure 3, it is possible to stack KHFs as many times as is desired.

5.3 Master Key Storage

The master key must be stored on a device that provides for secure deletions that occur once per epoch. The amount of storage on the device can be quite small, since it only needs to store one or two keys. The best option for this is a secure enclave such as the one coupled with Intel's Software Guard Extensions [6] or Apple's Secure Enclave [1]. A more accessible option would be a smart card designed specifically for this purpose, such as a YubiKey [8].

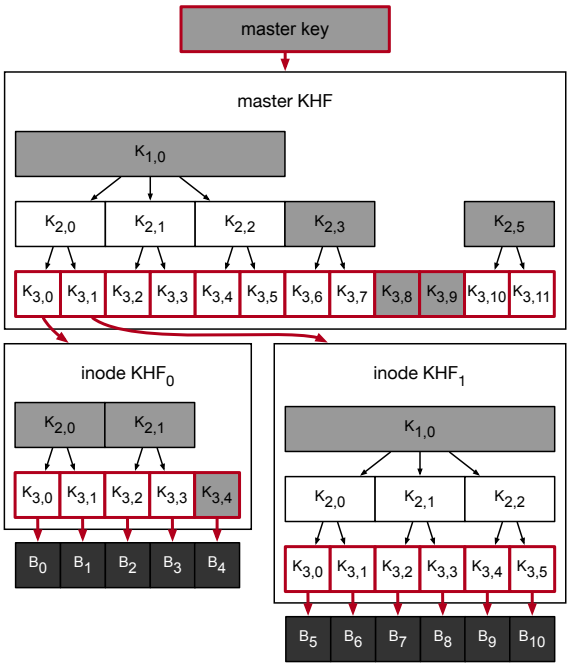

Figure 3. The hierarchy of KHFs in Lethe.

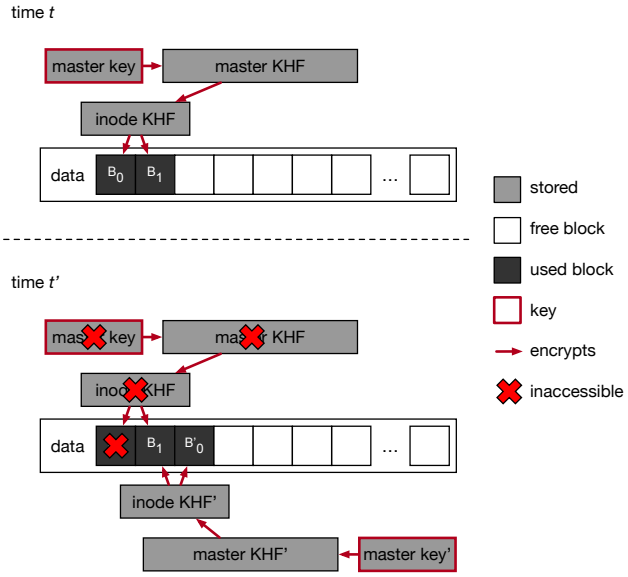

Figure 4. State before and after "overwriting" a block.

5.4 Epochs

Epochs are time intervals over which updates to KHFs are deferred and batched together, much like the approach of purging epochs used by DNEFS [21], and similar to the group commit technique employed by ZFS [9] and WAFL [14].

Lethe accumulates the updates that are made to KHFs throughout an epoch and applies them at the end of an epoch; the updated inode KHFs are encrypted with new keys derived from the updated master KHF and persisted, and the updated master KHF is then encrypted with a new master key and persisted. Securely deleting the previous master key, or *epoch key*, ensures the secure deletion of any data deleted during the previous epoch. Thus, Lethe provides secure deletion guarantees at an epoch granularity.

By increasing epoch duration, we gain performance by avoiding the cascade of KHF updates and persists that occur on any data modifying operation. Figure 4 shows the occurrence of this cascade of KHF updates and persists when "overwriting" just a single block. Time t in Figure 4 shows the state of Lethe before overwriting block B_0. Since Lethe utilizes CoW, overwriting block B_0 requires a new block, block B'_0, to be written. This requires a new version of the KHF covering block B_0 to be written with its contents updated to replace the key for block B_0, thereby removing access to block B_0. In turn, a new version of the master KHF is then written with its contents updated to replace the key for the updated KHF. Finally, a new master key for the updated master KHF is generated. Securely deleting the previous master key takes us to the state shown at time t' in Figure 4, in which the "overwritten" block B_0 is inaccessible and considered securely deleted. Handling this cascade of KHF updates

and persists for each write is inefficient, which provides the motivation for the usage of epochs, since they allow the updates to be batched and persisted together.

Epoch duration can either be based on time, amount of data written, or simply manually triggered. Importantly, epoch duration trades off performance for the frequency of secure deletion. In general, the longer the epoch, the more performant the system. The shorter the epoch, the more rapid the guarantee of secure deletion.

5.5 Consolidation

KHFs become fragmented as writes occur over time, with the degenerate case for a KHF being that it contains a root for each key that it provides. This is typically not a concern for an inode KHF since files tend to be written sequentially [23] and will have multiple blocks covered by a common root, but is a concern for the master KHF.

The leaves of a master KHF provide the encryption keys for the inode KHFs. Unless inodes are modified sequentially by i-number during an epoch, the master KHF will generally fragment into the degenerate case. The size of the master KHF as it fragments becomes unwieldy given that the master KHF must be re-encrypted and persisted after every epoch.

Consolidation addresses the issue of fragmentation. Consolidation is a KHF compaction operation that incurs the penalty of re-encrypting data in exchange for a reduction in KHF size. For an inode KHF, consolidation requires re-encrypting a range of data blocks using a common root that will then replace the roots covering the re-encrypted data blocks. For the master KHF, consolidation requires re-encrypting a range of inode KHFs and updating the master KHF with a single root that covers those re-encrypted inode KHFs. Although paying the cost of consolidation to improve performance may be occasionally necessary, it is important to note that delaying consolidation indefinitely has no impact on the *security* of Lethe.

6 Evaluation

Our goals for the evaluation of Lethe were:

1. To measure the performance of Lethe integrated into a widely-used system using real-world workloads.
2. To measure the performance overhead incurred in providing secure deletion with Lethe.

6.1 Implementation

For goal (1), we integrated Lethe into the Zettabyte File System (ZFS) [9]. The choice of ZFS over other file systems such as Ext4 and BtrFS was motivated by its structure, its mechanisms for ensuring data consistency, and its native encryption feature.

Structure. ZFS operates on *objects*, logical groupings of data blocks such as files and directories, which are then logically grouped into *object sets* (e.g. a file system). We added a KHF per-object to provide keys for an object's data blocks, and added an master KHF to provide the per-object KHF keys for a single file system.

Mechanisms for Data Consistency. ZFS uses copy-on-write and atomic transactions to ensure data consistency, both of which are desired by Lethe. ZFS handles any write operation by treating it as a transaction. Transactions are further grouped into transaction groups, which are batched together and committed to disk periodically. This mechanism of batching transactions together and committing them together aligns perfectly with Lethe's use of epochs. Thus, not only does ZFS provide copy-on-write semantics, it also happens to already include a periodically-run procedure to flush data to disk that can be extended to update KHFs and flush them to disk as well.

Native Encryption. Native encryption in ZFS provides a unique encryption key for each data block in a zpool using HKDFs [16], allowing for encrypted data sets. We note that, despite having per-block keys, *ZFS native encryption does not provide fine-grained secure deletion.* This is due to the fact that the public HKDF parameters used for deriving block keys are stored in unencrypted block pointers on disk, and erasing the parameters is not possible due to ZFS' usage of CoW. Furthermore, it is untenable to swap out the main HKDF keying material, as it would require every other block in the system to be re-encrypted using keys derived from the new keying material.

ZFS delegates all I/O operations to its ZIO layer, which requests encryption keys for each data block it acts on. Integrating Lethe into the ZIO layer was straightforward: we only modified it to request for a Lethe-managed encryption key for each data block instead of its native encryption-provided key.

We note that, at the time of writing, the prototype of ZFS does not yet handle partial block truncations. A partial block truncation occurs when a file truncation operation leaves behind part of a block, and as such ensuring secure deletion of the truncated block requires that the remaining, untruncated bytes be rolled forward via re-encryption with a new key. Forcing ZFS to issue this additional write, though possible, is complex and remains future work. We note that this does not critically threaten the validity of our evaluation since files tend to be written to sequentially [23] and do not typically encounter partial block truncations; there wouldn't be a big performance hit even if this issue was handled by rewriting partial blocks.

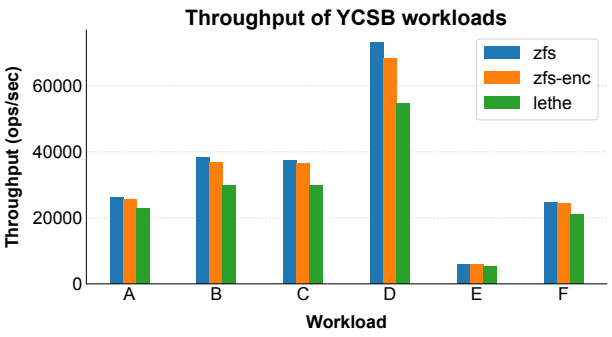

Figure 5. Result of YCSB benchmark with 5 second epochs.

6.2 Methodology and Experimental Setup

For goal (2), we compared ZFS integrated with Lethe against baseline ZFS and ZFS with native encryption enabled. Comparing the throughput of these three file systems allows us to see the relative performance overhead incurred through the addition of native encryption and secure deletion, ultimately making it easier to evaluate whether or not secure deletion is performant enough for adoption.

Our experiments were conducted on a 10^{th} generation Intel NUC equipped with an Intel CPU i5-10210U (1.60 GHz, 4 cores) and 32 GB of memory. A Samsung 970 EVO 500 GB SSD was used as the storage device. Lethe was configured to statically use a $\langle 16, 32, 8 \rangle$ fanout list for each constructed KHF. The duration of an epoch was set as 5 seconds—exactly the default duration between commits of ZFS [9] transactions. It remains future work to experiment with different epoch durations and epoch durations, measuring key storage overhead, and optimizing our implementation.

6.3 Results

We ran RocksDB [12] over each file system and measured the throughput of each using the YCSB [11] cloud serving benchmark on workloads A–F, each with 1 million operations, 1 million 1 kB records, using the Zipfian distribution. Each of the workload benchmarks were conducted using Pilot [19], a statistics-driven benchmarking framework. Figure 5 shows the result of the experimentation.

We see that the relative throughput of each file system across all the workloads is consistent. On average, going from baseline ZFS to natively encrypted ZFS yields a 2.6% decrease in throughput, going from baseline ZFS to Lethe-integrated ZFS yields a 17.63% decrease in throughput, and going from natively encrypted ZFS to Lethe-integrated ZFS yields a 15.5% decrease in throughput. Enabling native encryption on top of baseline ZFS understandably introduces a decrease in performance due to the need to generate encryption keys, encrypt data blocks, and store generated keys. Enabling secure deletion on top of baseline ZFS understandably introduces even more of a decrease in performance due to the increased cost of deriving keys, running the per-epoch KHF updating procedure, and storing KHFs.

Although the prototype of Lethe sees this degradation in performance compared to ZFS with native encryption, we expect to see less of a difference in performance as optimizations are made in the future. In any case, it is clear that the performance with secure deletion enabled on ZFS is comparable to baseline ZFS and ZFS with native encryption despite the relatively short epoch duration.

7 Conclusion

Lethe is the first system that provides fine-grained secure deletion on any storage medium, including those that make in-place overwrites difficult or even impossible, such as flash and WORM media. We first explored prior work in this design space and shown existing issues with prior systems that make their adoption for use in response to legislation like GDPR, CCPA, and CDPA unlikely. From there, we provided the notion of needing a hierarchy of keys for fine-grained secure deletion, which Lethe efficiently represents using KHFs. We then presented the design of Lethe, demonstrating how it provides secure deletion through addition of KHFs, and how it provides timely, fine-grained secure deletion of any selective amount of data through the secure deletion of just a single key. This further emphasizes the efficiency of Lethe, since it is impossible to achieve secure deletion by securely deleting any less than a single key; any less would be insecure. Our evaluation of a Lethe integrated into a real-world system also yields promising results despite lack of tuning and optimizations, demonstrating that Lethe can be seriously considered as a portable solution for timely guarantee of secure deletion.

Acknowledgments

This project was supported by the National Science Foundation under Grant No. CNS-2106259 and Grant No. CNS-1814347, Meta, and the industrial sponsors of the Center for Research in Storage Systems (CRSS) at UC Santa Cruz. We are thankful for the insightful comments and suggestions provided by faculty and students in the CRSS. We in particular would like to thank Peter Alvaro for the advice and feedback that made this work much stronger. We would also like to thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] Apple secure enclave. https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web. [Online; accessed 2, February 2023].
[2] California consumer privacy act. https://oag.ca.gov/privacy/ccpa. [Online; Accessed 24, May 2022].
[3] Consumer data protection act. https://law.lis.virginia.gov/vacodefull/title59.1/chapter53/. [Online; Accessed 17, Feb 2023].
[4] Executive order 12356–national security information. https://www.archives.gov/federal-register/codification/executive-order/12356.html. [Online; Accessed: 17, Feb, 2023].

[5] General data protection regulation. https://gdpr.eu. [Online; Accessed 24, May 2022].

[6] Intel software guard extensions. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. [Online; accessed 2, February 2023].

[7] What is data governance? https://cloud.google.com/learn/what-is-data-governance. [Online; Accessed: 17, Feb, 2023].

[8] Yubikey. https://www.yubico.com/products/. [Online; accessed 2, February 2023].

[9] Matt Ahrens, Jeff Bonwick, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. San Francisco, CA, March 2003. USENIX Association.

[10] Patrick Anderson, Richard Black, James Clegg, Chris Dainty, Raluca Diaconu, Austin Donnelly, Rokas Drevinskas, Alexander L Gaunt, Andreas Georgiou, Ariel Gomez Diaz, Peter G Kazansky, David Lara, Sergey Legtchenko, Sebastian Nowozin, Aaron Ogus, Douglas Phillips, Antony Rowstron, Masaaki Sakakura, Ioan Stefanovici, Benn Thomsen, Lei Wang, Hugh Williams, and Mengyang Yang. Glass: A new media for a new era? July 2018.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Facebook RocksDB Team. RocksDB: A Persistent Key-value Store for Fast Storage Environments. http://rocksdb.org. [Online; accessed 12, December 2022].

[13] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, page 8, USA, 1996. USENIX Association.

[14] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. 10 2000.

[15] Richard Kissel, Andrew Regenscheid, Matthew Scholl, and Kevin Stine. *Guidelines for Media Sanitization.* Number NIST SP 800-88r1. Dec 2014.

[16] Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. Cryptology ePrint Archive, Paper 2010/264, 2010. https://eprint.iacr.org/2010/264.

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

[18] Yan Li, Nakul Sanjay Dhotre, Yasuhiro Ohara, Thomas M. Kroeger, Ethan Miller, and Darrell D. E. Long. Horus: Fine-grained encryption-based security for large-scale storage. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 147–160, San Jose, CA, 2013. USENIX.

[19] Yan Li, Yash Gupta, Ethan Miller, and Darrell Long. Pilot: A framework that understands how to do performance benchmarks the right way. pages 169–178, 09 2016.

[20] Zachary N. J. Peterson, Randal Burns, Joe Herring, Adam Stubblefield, and Aviel D. Rubin. Secure deletion for a versioning file system. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, page 11, USA, 2005. USENIX Association.

[21] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 333–348, Bellevue, WA, August 2012. USENIX Association.

[22] Joel Reardon, Claudio Marforio, Srdjan Capkun, and David Basin. User-level secure deletion on log-structured file systems. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 63–64. Association for Computing Machinery, 2012.

[23] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15. Association for Computing Machinery, 1991.

[24] Michael Wei, Laura M. Grupp, Frederick E. Spada, and Steven Swanson. Reliably erasing data from flash-based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, page 8. USENIX Association, 2011.