UNIVERSITY OF CALIFORNIA
SANTA CRUZ

LETHE: IT WON'T TAKE LONG TO FORGET

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Eugene Chou

June 2022

The Thesis of Eugene Chou
is approved:

Professor Darrell D. E. Long

Professor Ethan L. Miller

Professor Andrew R. Quinn

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Algorithms

Abstract

Lethe: It Won't Take Long To Forget

by

Eugene Chou

Modern general data privacy regulations in Europe (GDPR) stipulate that, at a user's request, data pertaining to them is deleted without undue delay. Existing storage systems are not equipped to provide secure deletion, leaving traces of deleted data for indeterminate periods of time, sometimes on the order of months. Current approaches to secure deletion, overwrite erasure and cryptographic erasure, are also unsatisfactory. Overwrite erasure requires numerous in-place overwrites that are difficult on flash media and negatively impact media lifetime. With cryptographic erasure, secure deletion of data is tied to secure deletion of the encryption key. This quickly becomes a key management problem since enabling fine-grained deletion requires that a key must be maintained for each data block that may be deleted. To address these problems, we propose Lethe, a new system that provides fine-grained secure deletion regardless of storage medium by utilizing keyed hash trees. With keyed hash trees, Lethe is able to drastically reduce the amount of key material that must be stored and forgotten while still providing the necessary amount of keys required for fine-grained secure deletion. The amount of key material that needs to be securely deleted in Lethe does not increase linearly with the amount of data that is to be securely deleted. With Lethe, the fine-grained secure deletion of any amount of data requires only a *single* key to be securely forgotten.

To my friends and family,

Thank you for constantly believing in me,

especially during the times when I don’t believe in myself.

Chapter 1

Introduction

Some of the most important changes in data privacy regulations in the past 20 years include the "EU General Data Protection Regulation (GDPR)" [1] and the "California Consumer Privacy Act (CCPA)." [2] Both the GDPR and CCPA stipulate the individuals have the "right to be forgotten" and "right to delete personal information," respectively. The GDPR explains in Article #17 that the "right to be forgotten" means that not only does a data subject have the right to request a data controller to erase any data that concerns them, but also that the data controller must perform the erasure without undue delay. Furthermore, the CCPA states that the response to a data erasure request must occur within 45 calendar days, with an extension to up to 90 days if the data subject is notified beforehand. In any case, what it means to "forget" or "delete" data is left vague—perhaps deliberately so—but the canonical interpretation is that the end result of either is that forgotten or truly deleted data is irrecoverable *computationally*, perhaps even physically. Thus, both the GDPR and CCPA require *timely secure deletion* of data. The issue with existing storage systems and data centers is that they do not provide the secure deletion of data in a timely manner [3]. As an example, we consider Google Cloud. According to its documentation [4], it may take up to *six months* for data to be securely deleted from their cloud hosting systems. The reason for this delay is in part due to the method that is employed to securely delete data: *overwrite*. With overwrite erasure, data is securely deleted once it has been overwritten in-place a sufficient number of times with alternating patterns of ones and zeros. This has two key issues, the first of which is that excessive overwrites degrade storage mediums at a higher rate, and the second of which is that in-place overwrites are difficult, even practically impossible [5] for flash-based storage.

A viable alternative to providing secure deletion is through *cryptographic erasure*. With cryptographic erasure, persisted data is always encrypted. To forget the persisted data, one only needs to guarantee the secure deletion of the encryption key, which is much easier to guarantee due to the small size of a key. With a sufficiently secure cipher, recovering the encrypted data without the key becomes a computationally infeasible task. Cryptographic erasure, however, does come with its own set of problems that must be addressed. The largest of these problems is the problem of *key management*. While it does reduce the amount of data required to be forgotten down to a single key, secure deletion through cryptographic erasure requires a separate key for every *block* of data that *might* be deleted in order to provide fine-grained secure deletion, the keys in which must be able to be securely deleted. While all the block encryption keys themselves can be protected behind a single wrapping key, we ideally would like to minimize the amount of key material that must be stored and forgotten while still providing fine-grained secure deletion.

To address the issues with overwrite erasure, the key management problem that accompanies cryptographic erasure, as well as shortcomings of previous related systems, we introduce *Lethe*, a new approach for providing fine-grained secure deletion in a file system that efficiently manages keys using *keyed hash trees* (KHTs) [6], and allows for an arbitrary amount of data to be forgotten at a moment's notice with just the secure deletion of a *single encryption key*. The next chapter, Chapter 2, will discuss prior secure delete systems. Chapter 3 will discuss requirements of secure deletion and what Lethe is required to provide. Chapter 4 will be dedicated to KHTs and their usage. Chapter 5 will be about *encryption root lists*, a new data structure introduced by Lethe to address the shortcomings of KHTs. This segues nicely into Chapter 6, which presents the complete design of Lethe in a bottom-up manner and leads into Chapter 7, which discusses the usage of epochs in Lethe's design to improve performance. Preliminary results of a userspace prototype of the presented design is shown in Chapter 8. Finally, we conclude with our closing remarks in Chapter 9.

Chapter 2

Background

Different approaches to secure deletion have been proposed, but all either incur significant performance penalties or are targeted for specific storage mediums, both of which make their use in systems affected by laws like GDPR unlikely. Early systems such as the one described by Bauer *et al.* [7] focused on providing secure deletion through overwrite erasure, specifically by repeatedly overwriting deleted data blocks. Gutmann *et al.* [8] later proposed an improved overwrite erasure technique which entails writing 35 alternating patterns of ones and zeros over deleted disk regions. While both approaches are simple and effective, neither are particularly efficient given the requirement for multiple data block overwrites. Furthermore, neither approach works with flash media [5].

With flash, data blocks must first be "erased" before they can be overwritten. This is due to the unique characteristics of flash cells, in which only ones can be flipped to zeros, but never from zeros to ones. To avoid the overhead of needing to erase blocks in-place (setting all cells to one) before writing, flash memory devices such as SSDs typically just write to a new block and handle logical block mappings with the flash translation layer (FTL). The main takeaway with flash is that overwrite erasure does not really work with flash media since physical overwrite is not guaranteed.

Cryptographic erasure is the popular, alternative method of providing secure deletion, first proposed by Boneh *et al.* [9] as a fast means to forget data on magnetic backup tapes. With cryptographic erasure, encrypted data is considered securely deleted when the corresponding encryption key is forgotten. This technique was used by Peterson *et al.* in a file system [10] that appended a small decrypting "stub" to the end of each encrypted data block. The overwrite of the stub would mean the secure deletion of the corresponding data block. This also means that the amount of metadata maintained must necessarily grow linearly as the amount of data grows, since each data block gets its own stub. Lee *et al.* later designed a system for flash that modifies YAFFS to encrypt and securely delete data by erasing the keys and metadata [11]. Both approaches, however, despite being more efficient than either Bauer or Gutmann's systems with regard to the amount of data that needs to be overwritten, are still unsatisfactory for use with flash due to the need for in-place overwrites.

Reardon *et al.* [12] attempted to address the in-place overwrite problem with two userspace techniques called purging and ballooning. Purging fills free space with junk data, ensuring all blocks are overwritten, while ballooning artificially constrains free space. Both purging and ballooning are expensive and have an adverse effect on flash endurance due to the excessive writes of random bits and forced garbage collection.

Other approaches to the in-place overwrite issue involve modifying the FTL itself [13, 14], or even implementing custom firmware [15]. The former approach was a system proposed by Kim *et al.* to provide secure deletion on SSDs and the latter approach was a system proposed by Chen *et al.* to provide secure deletion on shingled magnetic (SMR) drives. Though possible to implement, both approaches seem impractical due to the need for custom firmware.

It is clear that none of these prior systems provide a satisfactory solution for providing efficient, fine-grained secure deletion, and certainly not one that can be used on any storage medium, regardless of the characteristics of the medium. From this, we establish the main goal of Lethe: to provide a system that provides fine-grained secure deletion, minimizes the amount of persisted metadata, and works regardless of storage medium.

Chapter 3

System Requirements

Here, we concretely establish requirements of the Lethe system. First, we establish how much data is required to be forgotten in order to provide secure deletion from an information-theoretic standpoint. This requirement is not strictly just for Lethe, and is in fact a requirement for any system that provides secure deletion. Then, we present a design for a system that provides *coarse-grained* secure deletion and contrast it against a design for a system that inefficiently provides *fine-grained* secure deletion. This provides the necessary exploration into the design space of secure delete systems and also provides a good opportunity to concretely establish what requirements Lethe is expected to fulfill.

Simply put, to forget encrypted data, we *must* securely delete *some* piece of information used to encrypt it. The size of this information must be large enough that such that it isn't vulnerable to a brute force attack. A single 128-bit key paired with a cryptographically secure cipher meets these requirements, making the smallest amount of information necessary to be securely deleted in order to provide secure deletion a single encryption key. If for some reason

128 bits is insufficient, this can easily be amended to match the key size of any secure cipher of choice.

3.1 Coarse-Grained Secure Deletion

We start by defining that a coarse-grained secure delete system provides encryption per-file, rather than per-data block. This means that each file gets its own encryption key and uses it to encrypt every data block that constitutes the file. Naturally, this means that any change to a data block in a file—even a single byte overwrite—requires the *entire file* to be re-encrypted.

To make matters worse, we must also keep in mind the amount of key material that must be re-encrypted whenever changes are applied to a file. Let us assume a relatively small file system with 4.000 00 TiB of space for files. Further assume that the system uses a block size of 4.000 00 KiB and that each file consists of exactly 4 data blocks. With these assumptions, 4.000 00 GiB of keys, each of which are 16.000 00 B in size, are needed in order to protect each possible file on the system.

Like any data, these keys must be stored as well, perhaps in a dedicated key file. With this setup, even changing a couple of megabytes across a set of files would potentially require the entire 4.000 00 GiB key file to be re-encrypted. Ideally, we would like it if a small change to a file did not require its complete re-encryption. An immediate solution that comes to mind is by moving away from coarse-grained encryption, and towards *fine-grained* encryption.

3.2 Fine-Grained Secure Deletion

For any storage encryption system to provide fine-grained secure deletion, it must necessarily encrypt each block of data with its own key, thereby providing encryption per-data block. This, of course, requires a lot more overhead for key storage than a system that provides coarse-grained secure deletion. The upside is that a fine-grained approach *reduces* the amount of re-encryption that occurs whenever a part of a file is modified. With a coarse-grained approach, the entire file is required to be re-encrypted. With a fine-grained approach, only the blocks that were modified need to be re-encrypted.

The reduction in re-encryption applies not just to data blocks, but also the keys themselves. Like in the previously presented coarse-grained system, the block encryption keys in a fine-grained system can be stored in a file. Whenever a key is changed, only the block in which the key is found needs to be re-encrypted, not the entire key file.

To compare the fine-grained approach with the previously presented coarse-grained approach, we assume the same 4.000 00 TiB system with 4.000 00 KiB blocks. Since encryption is now provided per-block, there is now 16.000 00 GiB worth of key data that must be stored, assuming 16.000 00 B keys. The problem with this naïve fine-grained secure deletion approach is that, while it does reduce the amount of re-encryption needed for per-file changes, it suffers from having a huge overhead for key storage.

3.3 Requirements for Lethe

Now that we've explored how prior systems provide secure deletion, as well as an exploration into the design of both coarse-grained and fine-grained secure deletion systems, we can establish requirements for Lethe.

1. Lethe must provide timely secure deletion.
2. Lethe must provide fine-grained secure deletion.
3. Lethe must provide the secure deletion of an arbitrary amount of data through the secure deletion of a single key.
4. Lethe should minimize the amount of key material that must be managed and stored.

Our proposed design of Lethe fulfills all these requirements. On demand, Lethe is able to securely delete an arbitrary amount of data through the secure deletion of just a single key. On top of this, Lethe also provides fine-grained secure deletion while efficiently managing and minimizing stored key material. What enables all of this is the use of *keyed hash trees*.

Chapter 4

Keyed Hash Trees

Keyed hash trees (KHTs) were originally presented by Li *et al.* for use in Horus [6], which provided a way to efficiently generate and manage data block encryption keys to provide fine-grained security in high performance computing (HPC) systems. Part of its efficiency is due to its nature of being a completely logical tree structure, eliminating the need to store pointers commonplace for typical linked structures. Since there are no pointers, the topology of a KHT must be defined in some other way. The simplest way to describe a KHT's topology is by defining its fanout at each level, where levels are indexed in a 0-based manner. This can naturally be expressed as a list of integers.

As an example, consider a fanout list $(2\ \ 3\ \ 2)$. This list describes a KHT where a level 1 (L1) node has 2 children, an L2 node has 3 children, and an L3 node has 2 children. Thus, any L1 node in this KHT topology is the ancestor of exactly 12 leaf nodes, where 12 is the product over all elements of the fanout list. Equivalently, we say that an L1 node *covers* 12 leaf nodes. It is important to note that the example fanout list describes a KHT's topology starting at level 1, and yet we stated earlier that levels in a KHT are indexed in a 0-based manner. This is precisely because a KHT has one true root at level 0 that has an *infinite fanout*, and thus covers an infinite number of L1 nodes. With this construction, it is possible to cover an infinite number of leaf nodes with just a single root node.

4.1 Defining A Node

Here we formally describe the key components of a KHT: nodes and their relationship with other nodes. A KHT node, sometimes referred to as a subroot if not the true root of a KHT itself, is a triple containing a value, level, and offset.

$$Node\ n = \langle value, level, offset\rangle$$

The level and offset of a node acts as a unique identifier, where the level indicates the level, or depth, in which the node can be found in, and where the offset indicates its position in the level itself. Consider some node n that exists at level 3 and is the 4th node in that level. The components of n would then be

$$n = \langle value, 3, 4\rangle$$

We will occasionally represent n in shorthand by just a tuple consisting of its level and offset, which is $\langle 3,4\rangle$.

The value component of the node triple is exactly an encryption key, since the original use of KHTs proposed by Horus [6] was to use leaf nodes as encryption keys for data blocks. This, of course, means that any suitable value of a leaf node can be anywhere between 16.000 00 B and 32.000 00 B in size, with 16.000 00 B being a lower bound. Since a KHT can cover an infinite number of leaf nodes from just a single root, it should be clear that a KHT can effectively encrypt an infinite number of data blocks. Next, we present the relationship between KHT nodes to show how and why KHTs are used for efficient key management.

4.2 Key Derivation

In a KHT, any child node must necessarily be computed, or *derived*, from its parent. Let node n be an immediate child of node m. Then,

$$n_{\text{value}} = H(m_{\text{value}} \,||\, n_{\text{level}} \,||\, n_{\text{offset}}),$$

where $||$ indicates concatenation. H is a cryptographically secure hash function such as SHA2 or SHA3. 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, meaning that it is trivial to compute the value of any descendant node given an ancestor node, but is computationally infeasible to compute the value of any ancestor node from a descendant node. This also ensures that it is computationally infeasible to compute the value of sibling nodes as well. Figure 4.1 is provided to show the structure of a KHT, how it covers an infinite number of leaf nodes, and the relationship between the connected nodes.

The KHT key derivation algorithm is provided in Appendix A. We note that the algorithm itself does not define a KHT's topology using a fanout list, but rather a list of descendants for simplicity. To demonstrate this, assume that KHT K is described using a $(2\ 3\ 2)$ fanout list. The corresponding descendant list for K would then be $(0\ 12\ 6\ 2\ 1)$. $K_0 = 0$ is just a placeholder to indicate an infinite number of descendants at L0, $K_1 = 12$ indicates that each L1

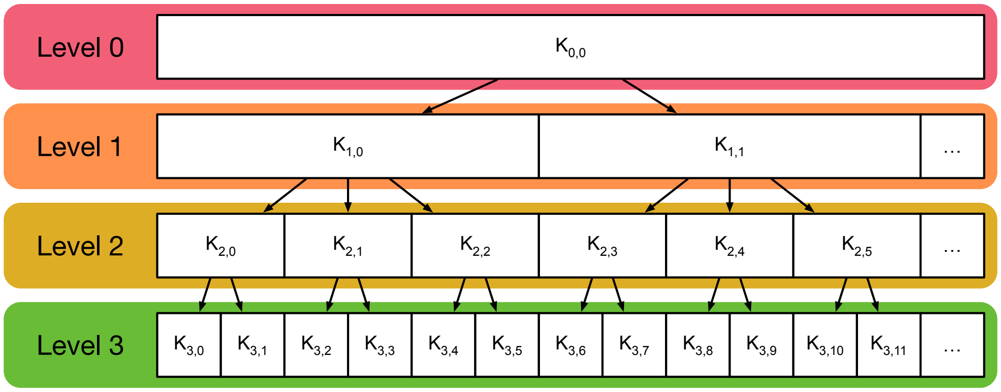

Figure 4.1: A $(2\;\;3\;\;2)$ keyed hash tree, as presented in Horus [6].

node has 12 descendants, $K_2 = 6$ indicates that each L2 node has 6 descendants, and so on and so forth. The 1 at the end of the descendant list is not strictly part of the topology itself, but is included to simplify the key derivation algorithm.

4.3 Key Revocation

An existing problem with Horus is the lack of a key revocation mechanism, with the original paper stating that the mechanism itself would be addressed in future work. The crux of the key revocation problem in a KHT is that it fragments a KHT into a *forest* of KHTs. This is due to the need to preserve the subroots needed to derive still-valid keys, but none of the subroots that can provide access to the revocated keys.

Consider Figure 4.2, which depicts the state of a KHT before and after modifying (overwriting) data blocks B_6, B_7, and B_8. This overwrite operation clearly shows the KHT fragmenting in order to both provide new subroots for the newly overwritten blocks, and also to prevent access to old blocks whose keys have been revocated. What is needed to address

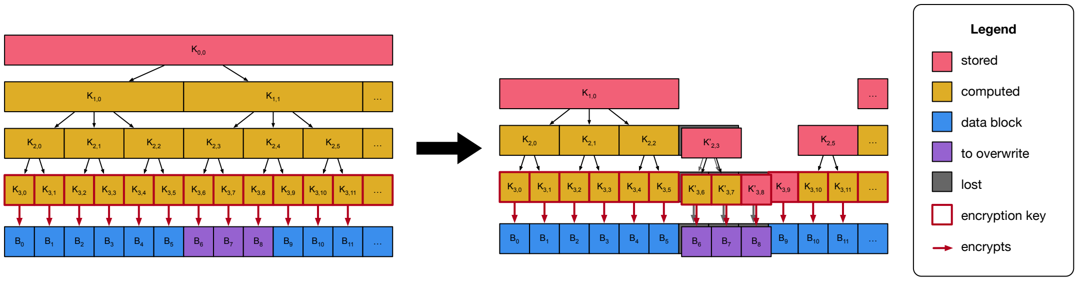

Figure 4.2: A KHT before and after modifying blocks.

the key revocation problem, then, is a container that describes a forest of KHTs. As a solution, Lethe introduces a new data structure, the *encryption root list*.

Chapter 5

Encryption Root Lists

Encryption root lists, or ERLs, are introduced by Lethe to describe a forest of KHTs from which encryption keys can be derived. An ERL is represented as a list of $\langle node, start, leaves\rangle$ tuples (items), where *node* is a $\langle value, level, offset\rangle$, as described in Section 4.1. The range of encryption keys that an ERL item covers is given by the half-open interval $[start, start + leaves)$.

Items in an ERL are sorted by their respective *start* components for efficient lookup. A lookup is required whenever an encryption key needs to be derived since each item covers only a certain number of keys. Without an established ordering over the items in an ERL, a lookup would be an $\mathrm{O}(n)$ operation, requiring a linear search to find the item that can be used to derive an encryption key. By establishing an ordering, a lookup becomes an $\mathrm{O}(\log n)$ operation by using binary search.

Having an ordering also simplifies perhaps the most important ERL operation: the *update* operation. This operation is necessary for key revocation since it updates an ERL with new subroots that provide new keys in place of the old, revoked keys, while retaining any

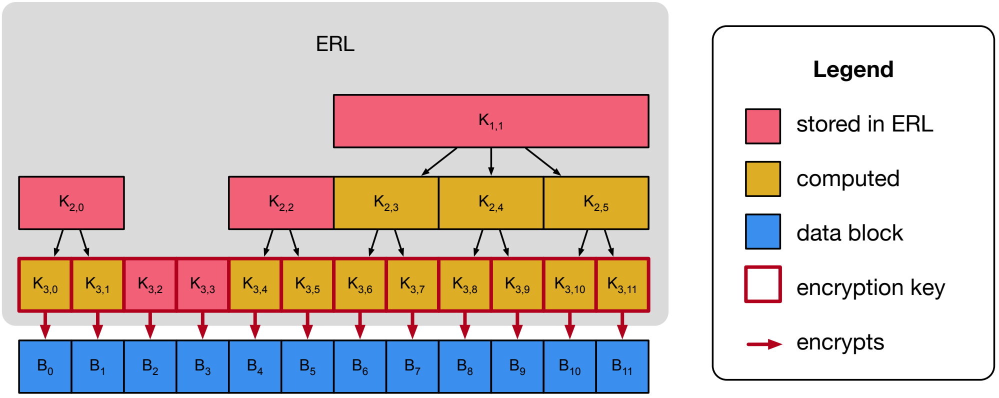

Figure 5.1: An ERL describing a fragmented $(2\ \ 3\ \ 2)$ KHT.

subroots still required for still-valid keys. Since the subroots from which the revoked keys are no longer in the updated ERL, the ERL itself cannot be used to derive the revoked keys. Figure 5.1 is included to precisely show what is and is not stored in an ERL at a high level, as well as how leaf nodes encrypt data blocks. This figure will be helpful going forward, when we present the roles that ERLs play in the overall design of Lethe.

Chapter 6

System Design

Here we present the design of Lethe in a bottom-up manner, starting from the metadata that is stored per-file, then the metadata that is stored system-wide. We then present how the read and write file system operations are handled, focusing especially on the write operation, since writes can further be categorized into either truncations, overwrites, and appends. We will describe, step-by-step, how each of these write operations are carried and out and change the state of the system. Showing the change in system state per-write sets up the motivation for the usage of epochs, as will be described in the next chapter.

6.1 Per-file Metadata

Each file in Lethe, along with its usual inode metadata, is given an ERL as well as a single KHT root, the *encrypting root*, from which to derive new encryption keys. The purpose of the encrypting root is to prevent unnecessary ERL fragmentation when writing a sequence of data blocks. Consider a file consisting of *N* blocks that is initially covered by a single KHT root, and assume that blocks 1 through N will be overwritten in sequence. Each of these blocks must be encrypted with its own key in order to satisfy the fine-grained secure deletion guarantee established by the design requirements of Lethe. Without a dedicated KHT root from which to derive new encryption keys, each one of the overwritten blocks is essentially covered by its own dedicated KHT root. This means that a full overwrite of the file will fragment its ERL from containing just a single KHT root to containing N KHT roots.

Clearly, we want to avoid this fragmentation. What we notice is that sequential blocks that are overwritten can be covered by a single common ancestor. This is where the encrypting root comes into play. Instead of immediately fragmenting an ERL, we can *delay* its fragmentation by using encryption keys derived from the encrypting root and later compute which subroots under the encrypting root need to be updated into the ERL. This delayed ERL fragmentation and update can be performed when an ERL is *persisted*.

Since system failures are always a possibility, file ERLs must be persisted at some point to allow for remounts. The persisted ERL must have been updated to contain all the still-valid roots, as well as any roots descended from the file's encrypting root needed to cover any overwritten blocks. Persisted ERLs go into metadata files in a manner similar to metadata files in WAFL [16].

It is important to note that persisted ERLs cannot be stored in the clear, since doing so would lose any notion of secure deletion. Let block B be a block encrypted by a key derived from ERL E. B cannot be considered securely deleted unless its encrypting key is truly inaccessible. By extension, this means that E cannot be freely accessible and thus must be encrypted when persisted. If E's encryption key is forgotten, then any block whose encryption key can only be derived from E is also forgotten. Naturally, this introduces a new key management problem, as each ERL now requires its own encryption key. To address this, we use yet another ERL at the inode level: the *master ERL*.

6.2 System-Wide Metadata

Lethe maintains a single ERL at the inode-level, the *master ERL*, to manage and provide encryption keys for file ERLs. The master ERL operates no differently than a file ERL, other than that it provides encryption keys per inode, and not per data block. It is through an inode's number, its *inumber*, that a file and its ERL's encryption key is uniquely identified. Deriving the ERL encryption key for a file corresponding with inumber N simply means deriving the value of leaf N of the master ERL.

Also like a file ERL, the master ERL must also be persisted in an encrypted manner. The master ERL's encryption key is referred to as the *master key*. It is this single key that unlocks access to the whole file system. Without it, the master ERL cannot be accessed, which means that no file ERL can be accessed, and thus means that no data block can be accessed. To securely delete all data on the system, one only needs to forget the master key. Figure 6.1 is provided to show the organization of Lethe: how the master key protects the master ERL, how the master ERL protects each of the file ERLs, and how the file ERLs protect data blocks.

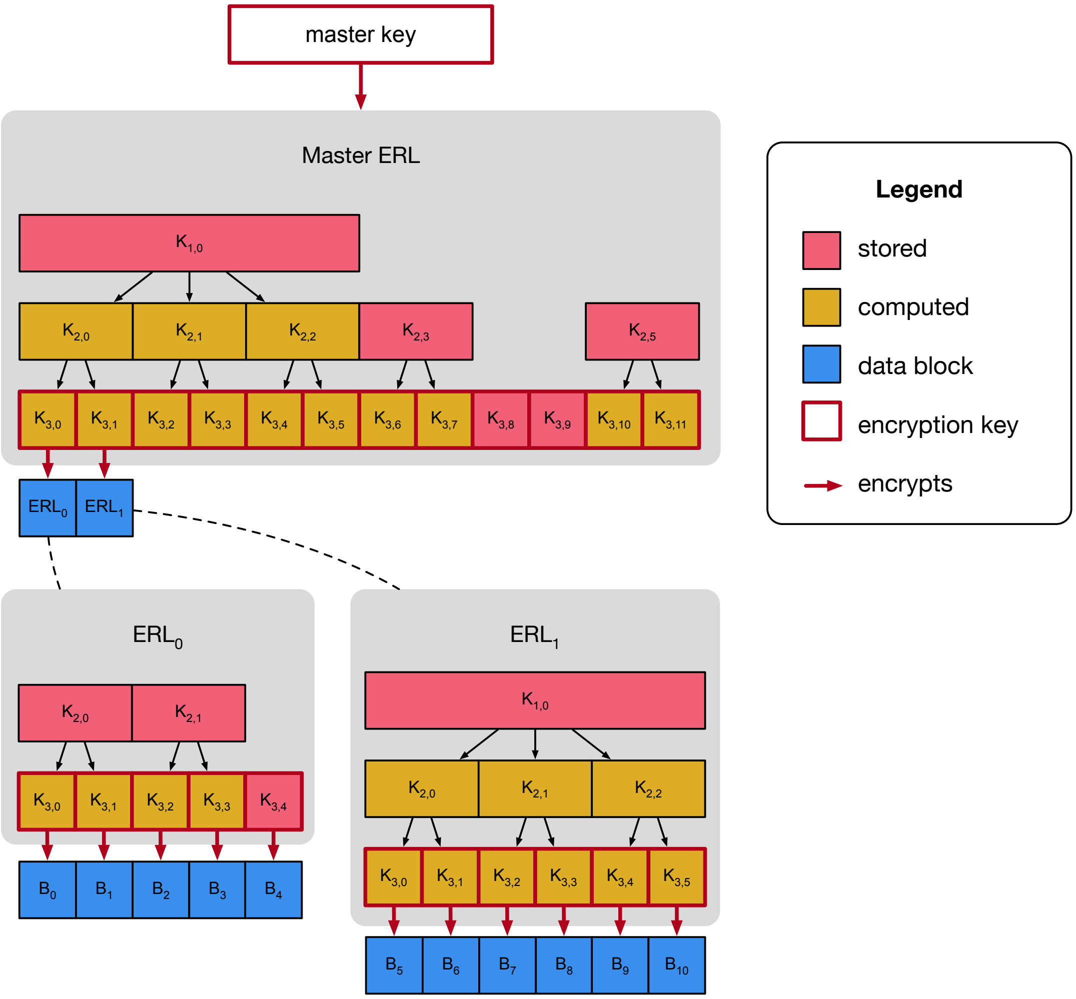
master key
Master ERL
$K_{1,0}$
$K_{2,0}$ $K_{2,1}$ $K_{2,2}$ $K_{2,3}$ $K_{2,5}$
$K_{3,0}$ $K_{3,1}$ $K_{3,2}$ $K_{3,3}$ $K_{3,4}$ $K_{3,5}$ $K_{3,6}$ $K_{3,7}$ $K_{3,8}$ $K_{3,9}$ $K_{3,10}$ $K_{3,11}$
ERL_0 ERL_1
ERL_0
$K_{2,0}$ $K_{2,1}$
$K_{3,0}$ $K_{3,1}$ $K_{3,2}$ $K_{3,3}$ $K_{3,4}$
B_0 B_1 B_2 B_3 B_4
ERL_1
$K_{1,0}$
$K_{2,0}$ $K_{2,1}$ $K_{2,2}$
$K_{3,0}$ $K_{3,1}$ $K_{3,2}$ $K_{3,3}$ $K_{3,4}$ $K_{3,5}$
B_5 B_6 B_7 B_8 B_9 B_{10}
Legend
stored
computed
data block
encryption key
encrypts

Figure 6.1: The Lethe system.

6.3 File System Operations

A presentation of a file system design would not be complete without discussion on how reads and writes are carried out. We cover reads first, then break up writes into the three aforementioned subcategories: truncations, overwrites, and appends. Each of the discussed file system operations will be accompanied by a step-by-step walkthough of the operation. The walkthrough is provided since we feel that it is valuable to cover how these operations, all of which can start at any arbitrary offset, are carried out despite using per-block encryption.

Before diving into the operations, we note that any of these operations, or file I/O requests, must fall into one of the following four cases:

1. The request starts on a block boundary and ends on a block boundary.
2. The request starts partway in a block and ends on a block boundary.
3. The request starts on a block boundary and ends partway in a block.
4. The request starts partway in a block and ends partway in a block.

The following sections will refer to these cases as case (1), case (2), case (3), and case (4).

6.3.1 Reads

Firstly, we must first get access to the ERL for the file that the I/O request is designated for. Note that the presented process for gaining access to a file ERL applies for all I/O request operations, not just reads. In order to read a file with inumber *N*, we derive leaf node *N* from the master ERL to unlock the file's ERL. After unlocking the ERL, we can move onto decrypting the data the comprises the read request.

Assume we have some file F made up of N blocks, where each block is of size S. In order to handle case (1), we simply need to derive the key for each data block included in the request. As an example, assume a request that reads $1 \leq K \leq N$ blocks starting from block P and ending at block Q. To perform the entire request, we simply need to derive the key for P and decrypt the block, then repeat this procedure for each block up to Q.

For case (2), assume a request that reads at byte $i \neq 0$ in block P and ends at block Q. To handle this, we read the entirety of P, decrypt it, and return $S - i$ decrypted bytes starting from i. If handling this non-block-aligned portion of case (2) doesn't complete the read, the rest of it can be handled by following the procedure for case (1), since the rest of the read must necessarily be block-aligned.

Handling case (3) is similar to case (2), but in reverse. First, we handle the read as we would for case (1), completing the part of the request that is block-aligned. After that, the read must start at the beginning of some block Q, but end on some byte $i < |Q|$ in Q. Here, we indicate the size of Q with $|Q|$ instead of S due to the possibility that Q may not be a full block. Regardless, to handle case (3) we simply read in the entirety of Q, decrypt it, and return the first i decrypted bytes.

Case (4) can be handled as a combination of cases (1), (2), and (3), though (2) must be handled slightly differently. Case (2) still handles the portion of the read that starts partway into a block, but now must also account for if the entire read is contained within a single block and doesn't end on a block boundary. In this event, we are trying to read k bytes starting from byte i in block Q, where $k+i < |Q|$. Handling this is straightforward, only requiring that we read the entirety of Q, decrypt it, and return k bytes starting from i. If the read spans multiple blocks, then case (1) handles all the full blocks in the middle, leaving the non-block-aligned part of the read to be handled with case (3).

6.3.2 Truncation

Truncating a file reduces its size by "removing" its data starting from the end. While to the user of a file system it appears that the truncated contents of a file are no longer present, the truncated contents still live on persistent storage, and can be accessed with raw block reads. This latter remark is true for not just truncations, but in general for any operation that seemingly removes data; the data still lives on persistent storage, but is simply inaccessible since the pointer to it has been lost.

To prevent the truncated data from being decrypted, the ERL of the truncated file must be updated to no longer include the roots from which the encryption keys for the truncated file contents can be derived. Note that the words "truncated file contents" are used instead of "truncated data blocks." This is precisely because the truncation of a file may not be block-aligned and thus leave part of a block untruncated.

If the truncation of a file is block-aligned, then there isn't anything to do other than update the file's ERL to only contain the subroots that cover the remaining blocks. If the truncation is not block-aligned, we need to re-encrypt the untruncated part of the block. This is exactly case (3). Assume that we are partially truncating block P to contain some number of bytes $i > 0$. First, we derive the key for P from the file ERL in order to decrypt P. Next, we re-encrypt and write the first i decrypted bytes in P. The new encryption key for these bytes is derived using the file's current encrypting root. The ERL is then updated to contain the subroot derived the encrypting root that covers the re-encrypted part of P.

6.3.3 Overwrites

The overwrite procedure mirrors the read procedure presented in Section 6.3.1 but perform writes instead of reads to modify existing data blocks. Handling case (1) is straightforward. Assume we have some file F made up of N blocks, each of size S, and a request that overwrites $1 \leq K \leq N$ blocks starting from block P and ending on block Q. To perform the request, we simply need to derive the key for each block starting from P using the file's current encrypting root, write the encrypted block, and repeat this procedure for each block up to Q.

For case (2), assume a request that overwrites at bytes $i \neq 0$ in block P and ends at block Q. To handle this, we first read and decrypt the entirety of P. We then take the first i decrypted bytes of P, then tack on $S - i$ bytes of the overwrite request to get to the block boundary. These $i + (S - i)$ bytes are then decrypted with the file's current encrypting root and written out to complete the overwrite of P. If handling this non-block-aligned portion of case (2) doesn't complete the overwrite, the rest of it can be handled by following the procedure for case (1), since the rest of the overwrite must necessarily be block-aligned.

It should follow logically that handling case (3) is similar to handling case (2). First, the block-aligned portion of the overwrite request is handled as described for case (1). After that, the overwrite must start at the beginning of some block Q, but end on some byte $i < |Q|$ in Q. This requires us to read to first the entirety of Q and decrypt it. The first i decrypted bytes are then replaced with the remainder of the overwrite request. To finish the request, the entirety of Q (now containing the “overwritten” bytes), is encrypted using the file’s current encrypting root, and written.

Case (4) can be handled as a combination of cases (1), (2), and (3), though (2) must be modified slightly to account for the event that an entire overwrite is contained within a single block. If an entire overwrite is contained within a single block, then there are k bytes starting from byte i in block Q, where Q is the block that contains the overwrite, and $k+i < |Q|$. To handle this, the entirety of Q is read in and decrypted. k of the decrypted bytes starting at byte i are replaced with the new bytes to overwrite. Then, the modified version of Q is encrypted with the current encrypting root and written out. If the overwrite spans multiple blocks, then case (1) handles all the full blocks in the middle, leaving the non-block-aligned part of the overwrite to be handled with case (3).

6.3.4 Append

There are two ways of handling appends, the main distinguishing factor being the level of the new roots added to the updated file ERL. The first way is to treat an append as its own, discrete operation, adding the highest roots possible from which keys for the newly appended blocks can be derived to the ERL. The second way is to treat an append exactly like an overwrite.

The first way is more efficient, since later appends can also take advantage of the same higher-level roots that were added to the ERL earlier, using them to provide encryption keys for newly appended blocks. The only concern with this approach is if a root is compromised, then any subsequently appended blocks that are encrypted with descendants keys from the compromised root are also compromised.

The second way, where appends are treated as overwrites, does not suffer from this problem. The downside to the second way is that it cannot make use of roots previously added to the ERL during any prior appends. The upside is that it eases the complexity of the write operation as a whole, since appends and overwrites are treated as the same operation.

What sets these two methods of handling appends apart is how the file ERL is updated, as well as the encryption keys for any appended blocks. It should be clear from the prior sections, though, that appends may not necessarily be block-aligned. An example of this would be appending to a file that has a size that isn't a multiple of the block size S, meaning that part of an append must go into an existing block. This could either be case (2) or (4).

To handle either case, assume we are appending k bytes to file F, which currently contains just $i < S$ bytes in block P. First, we read in the entirety of block P and decrypt it. Next, we tack on the maximum of k and $S - i$ bytes to the decrypted bytes. Using the maximum of these two quantities handles the case in which an append does not make it to a full block. These bytes are then encrypted using either the key derived from the current ERL, or the key derived from the current encrypting root, then written out. The former is done if appends are handled by reusing existing roots, and the latter is done if appends are handled by using new roots.

Any remaining bytes can either be handled in a fashion exactly like case (1) and case (3) for overwrites, presented in Section 6.3.3. The general procedure for both cases is exactly the same, except that no data needs to be read in and decrypted since all appended data at this point must necessarily go into data blocks that didn't exist before. It should be clear that the handling of case (1) here also applies to the event when an append starts on a block boundary and ends on a block boundary. With that, each of the four cases for the append have been presented, which concludes the step-by-step explanation for how Lethe handles each possible read and write operation.

While it hasn't yet been brought up, it should be immediately clear that any of the three write operations (truncate, overwrite, and append) all require the ERL to be updated due to the usage of new encryption keys. Once an ERL update occurs, it must be persisted at some point to tolerate system failures, as described in Section 6.1. This in turn requires the master ERL to be updated to provide the updated ERL a new encryption key, which also requires that the master ERL be re-encrypted and persisted with a new master key. It is evident that this cascade of ERL updates and persistings on every write operation does not bode well for the performance of Lethe. To address this problem, Lethe introduces the concept of an *epoch*.

Chapter 7

Epochs

As discussed in the previous chapter, any write operation results in a cascade of ERL updates and persistings. To avoid this, Lethe operates in *epochs*: intervals of time in which updates to ERLs are deferred and batched together. The end of an epoch is when ERLs are updated and persisted, similar to the execution of a group commit in a database system.

Each epoch is associated with an epoch key, which is exactly the master key that is used to encrypt the master ERL. Transitioning to a new epoch requres the generation of a new epoch key and the secure deletion of the previous epoch key. Data deleted during an epoch is only securely deleted when its corresponding epoch key is securely deleted. Thus, the secure deletion of data is provided at the end of every epoch.

A problem that arises due to the usage of epochs is deciding the *duration* of an epoch. Using a longer epoch makes Lethe more performant since ERLs aren't re-encrypted and persisted as often. Using a shorter epoch provides secure deletion of data on a more frequent basis, since each epoch key is forgotten more frequently. We note that the duration of an epoch need not be temporal. By this, we mean that one could ostensibly transition from one epoch to the next after accumulating a sufficient number of write operations. Since the need for performance or for timely secure deletion is entirely subjective, the epoch duration is set as a tunable parameter in Lethe. It is also important to note that an epoch can be forced to pass if immediate secure deletion is required.

7.1 Epoch Key Storage

A question that immediately arises following the presentation of Lethe's design is where the epoch key should be stored. Up until now, we have made no assumptions about where data is written, or even if in-place overwrites are possible. This is part of the novelty of Lethe: that it works agnostic to any unique characteristics of different storage mediums. That said, the per-epoch key that Lethe needs to maintain must be stored somewhere that can truly be securely deleted.

One possible option is to use a small mechanical disk and erase epoch keys through overwrite erasure techniques such as the Gutmann method [8]. Given the small size of a key, the performance impact of such an approach should be neglible.

Another option, perhaps the best, is to store epoch keys on a small amount of storage contained within a piece of dedicated secure hardware such as a *trusted platform module* (TPM), or an *enclave*. Two examples of the latter include the enclave coupled with Intel's Software Guard Extensions (SGX) and Apple's Secure Enclave, which provides secure non-volatile storage in which keys can be securely stored.

7.2 Data Consistency

It was mentioned in Section 6.1 that ERLs are periodically persisted in case of system remounts or crashes. In Chapter 7, it was made clear that ERLs are persisted at the end of each epoch. As with any correct file system, Lethe must maintain *data consistency* and handle writes atomically; either they occur or do not occur.

It would be problematic if a write operation was interrupted, and even more problematic if the write happened to be overwriting existing data in-place, since the existing data would then be effectively corrupted. Not only could this write corruption occur for data blocks, but also when persisitng ERLs. To solve this issue, Lethe uses the copy-on-write technique, as done by ZFS [17]. In a copy-on-write file system, in-place overwrites do not occur. Whenever a file block is to be "overwritten," the block is instead allocated and written to a new block on disk. The file's corresponding inode is then atomically updated to point to the new data block instead. If the system crashes partway through, the data block pointer is never updated and data remains in a consistent, non-corrupted state.

With Lethe, any writes that occur during an epoch are always done in a copy-on-write manner, leaving the original blocks alone. At the end of an epoch, any updated ERLs are re-encrypted and written, also in a copy-on-write manner, leaving the original ERLs alone. The master ERL is also updated, re-encrypted, and written in a copy-on-write manner, leaving the original master ERL alone. Then, as a transaction, all data block and ERL pointers are updated to reflect their new locations on disk. To implement this atomic pointer update transaction, a log similar to the intent log found in ZFS [17] is maintained. Figure 7.1 depicts the state of a Lethe

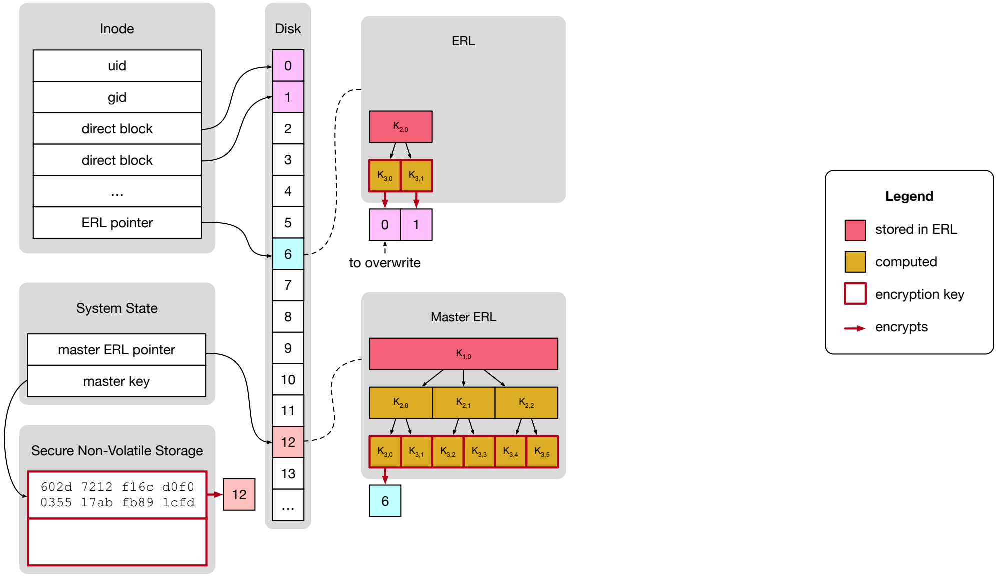

Figure 7.1: Lethe's maintained state before overwriting block 0.

system before an epoch concludes and Figure 7.2 depicts the state after the epoch concludes. The purpose of these figures are to showcase how Lethe leverages copy-on-write to maintain data consistency in the midst of system failures.

7.3 Consolidation

Notably, ERLs will become fragmented as writes occur over time, the worst case scenario being a file ERL that contains as many roots as it does data blocks. For a file, this is not the biggest concern since most files are written sequentially [18] and can thus have multiple blocks be covered by a single root. Where the fragmentation becomes an issue is with the master ERL. The leaves of a master ERL provide the encryption keys for the per-file ERLs. Unless the inumbers of files modified during an epoch are sequential, the master ERL will generally

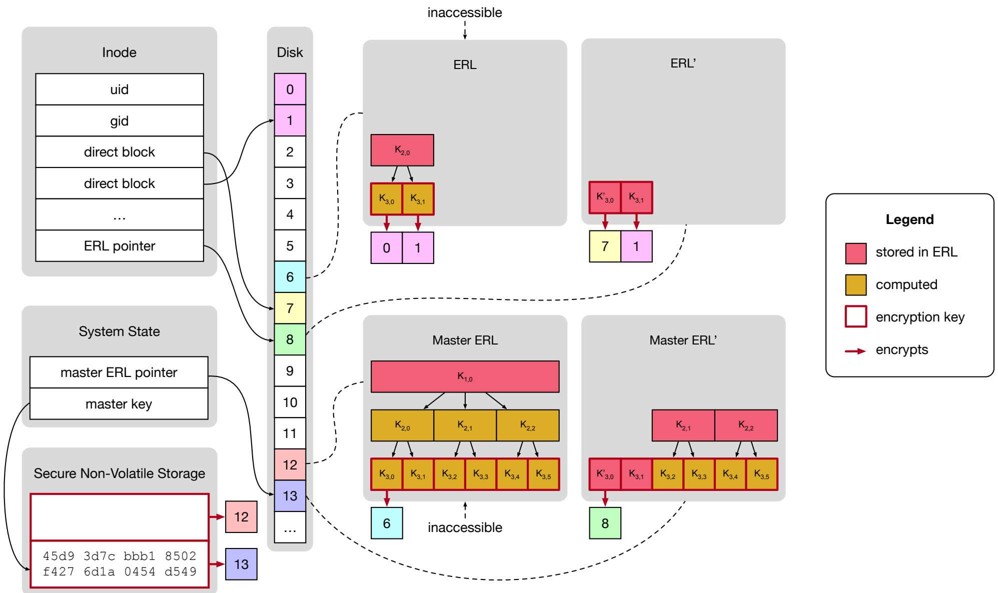

Figure 7.2: Lethe's maintained state after overwriting block 0.

fragment to the point of containing a single root per file. This is especially problematic when factoring in the fact that the master ERL must be re-encrypted and persisted after every epoch.

To combat this fragmentation, we introduce the idea of *consolidation*. Consolidation acts as an ERL compaction operation, in which the penalty of re-encrypting data is taken in order to reduce the size of an ERL. For a file ERL, this means re-encrypting a range of data blocks using a common root that will then replace the roots covering the re-encrypted data blocks. Similarly, for the master ERL, this means re-encrypting the per-file ERLs for a range of inodes, and updating the ERL with a single root that covers those ERLs. The cost to a large-scale consolidation operation is non-trivial due to all the re-encryptions that must occur, but may sometimes be necessary to improve performance in the long run. That said, the consolidation operation can be delayed indefinitely with no impact to security.

Chapter 8

Evaluation

Here we present the preliminary results of a Lethe prototype written in **C++** that uses FUSE, a software interface that allows for userspace file systems. The prototype is written as a pass-through file system, which makes Lethe's role effectively an additional computation layer, handling encryption and decryption while data persistence is handled by the host file system. The goal for these preliminary results is not to demonstrate the raw throughput of Lethe, but instead compare its relative performance against a pass-through file system without Lethe to provide secure deletion. This is due to the fact that FUSE is not especially fast, which does not make it a good platform to test raw performance on.

The benchmarks are carried out using `bonnie++`, a benchmarking suite for testing file system performance, and Pilot [19], a statistics-driven benchmarking framework that helps users produce reliable benchmark results. The performance indices from `bonnie++` that we are concerned with are the throughputs of writes, rewrites, and reads. The `bonnie++` write benchmark involves creating a new file, writing it to it a block at a time. The rewrite benchmark takes the created file, reads it a block at a time, dirties the block, and writes the block again after using `lseek()` to reset the file pointer to the position that the block should be written to. The read benchmark simply takes a file and reads it block by block.

We set up Pilot to repeat the `bonnie++` benchmark and analyze each of the aforementioned performance indices until a 95% confidence level is reached. `bonnie++` was configured to artificially constrain system RAM down to 4.000 00 GB in order to produce 8.000 00 GB files as part of the benchmark. This is due to `bonnie++`'s requirement that the size of the files created during the benchmark be exactly twice the amount of RAM. All tests were run on an machine equipped with an AMD EPYC 7452 32-core processor clocked at 3.000 00 GHz with 256.000 00 GB of RAM. We note that only the single-threaded performance of our prototype was benchmarked since multi-threaded operation has not yet been implemented.

Figure 8.1 compares the performance of two pass-through file systems, one using Lethe, and the other without. The one without Lethe is referred to as Styx, in reference to another one of the rivers of the Greek Underworld, and does not feature any encryption whatsoever. Thus, the goal of the evaluation is to see how exactly adding on Lethe impacts file system performance. The KHT topology used for all ERLs, including the master ERL, in this evaluation is defined with a $(8 \;\; 64 \;\; 32 \;\; 2)$ fanout list. The idea behind this fanout list is to have L1 nodes cover a large number of leaves, but have a smaller fanout at the lower levels to reduce the number of subroots produced upon KHT fragmentation. Experimenting with different KHT topologies remains future work. Epochs were implemented as a counter of write operations and passed whenever 2^{20} operations were counted. What we see is that Lethe performs worse than Styx for writes, rewrites, and reads, which lines up with our expectations. The reason for the performance decrease is due to Lethe's need to fragment ERLs, derive encryption keys, and encrypt data blocks.

We note that the performance of Lethe could be improved for the write benchmark if the more efficient append method discussed in Section 6.3.4 was implemented instead of the easer-to-implement method. We also mention that having epochs pass every 2^{20} means that epochs happen extremely infrequently, and thus the penalty of needing update and persist ERLs is not really paid. Figure 8.2 is included to demonstrate the impact of having a drastically shorter epoch, where only 2^{7} write operations are needed to trigger an epoch.

With shorter epochs, Lethe's write and rewrite throughput takes a massive hit while read throughput stays about the same as when epochs were passed every 2^{20} writes. This makes complete sense since only performing reads does not require any ERL updates, which means that no epoch penalty is taken.

The conclusion of the evaluation is not immediately clear. Ultimately, the question is whether or not the throughput penalty incurred in order to provide secure deletion is worth it. When using a decently long epoch like with Figure 8.1, Lethe consistently reports throughputs about 70% that of Styx's reported throughputs. Depending on the application, this sort of throughput may be acceptable, especially when considering the fact that performance can be improved further by implementing the more performant append method. Of course, there are also applications in which any sort of slow down is unacceptable, but arguably, providing timely secure deletion that doesn't depend on excessive overwrites or specific storage mediums certainly seems worth the penalty. Especially in the face of maintaining either GDPR or CCPA compliancy.

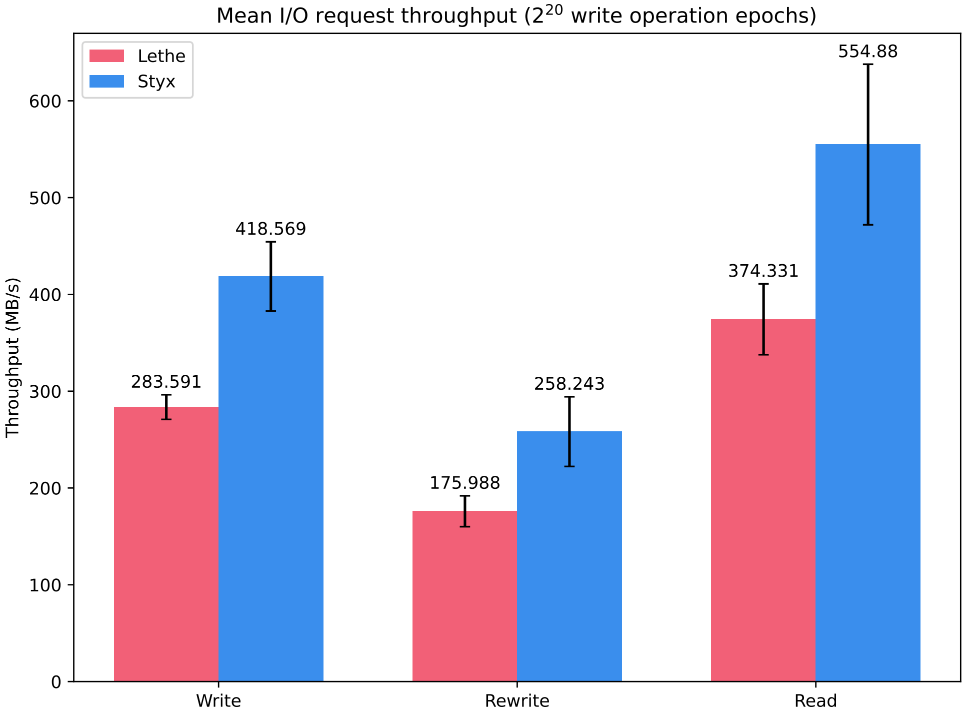

Figure 8.1: The performance of Lethe vs. Styx when passing epochs every 2^{20} write operations.

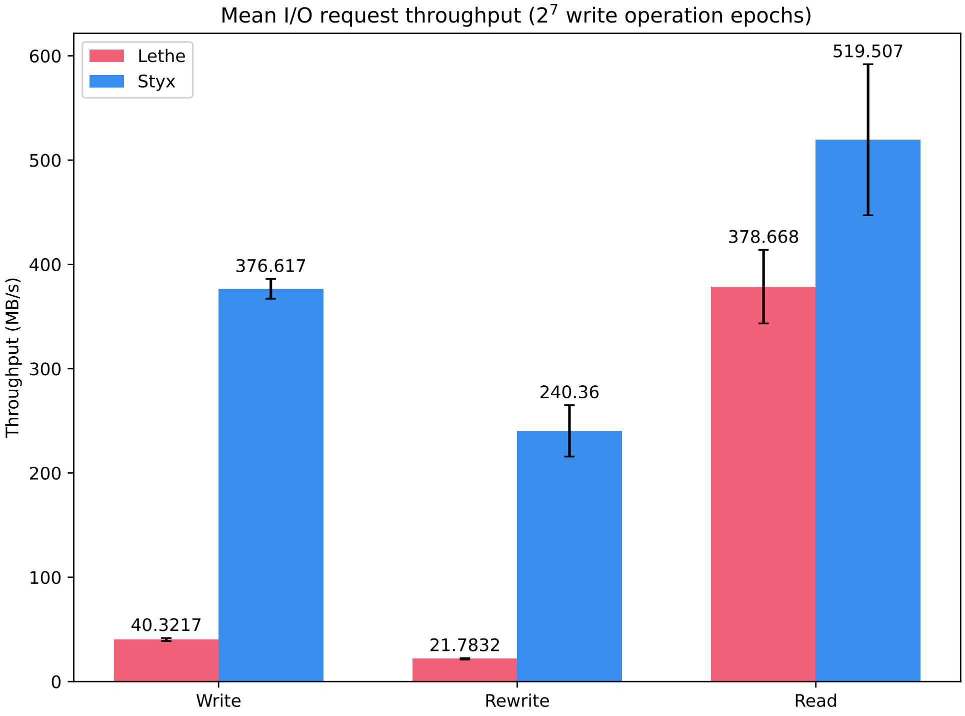

Figure 8.2: The performance of Lethe vs. Styx when passing epochs every 2^7 write operations.

Chapter 9

Conclusion and Future Work

We have presented Lethe, a new system that rethinks how fine-grained secure deletion is achieved. We have explored prior work in this design space, as well as described requirements for any secure delete storage system. From there we established the requirements of Lethe: timely secure deletion, fine-grained secure deletion, and tying secure deletion of an arbitrary amount of data to the secure deletion of a single key. Best of all, Lethe does so while minimizing the amount of key material that must be managed by using keyed hash trees for efficient key management. All of these points are demonstrably shown in the presentation of Lethe's design.

There are still a whole host of things that can be explored with Lethe itself. First and foremost, continuing to improve the read and write throughput is a must. An interesting experiment related to improving performance is experimenting with how different KHT topologies impacts system performance, or how tuning epoch duration impacts system performance. Another interesting experiment lies in ERL consolidation, and perhaps trying to establish heuristics for detecting opportune times to perform ERL consolidation. While both proposed experiments are interesting, the ultimate goal of Lethe is to integrate it into a production file system, the one in mind being ZFS [17].

Other questions that are worth considering down the line are concerned with Lethe's design. While it's clear that the design of Lethe works, one must wonder whether or not it is an optimal design and whether or not it could be further improved. For instance, is it possible to remedy the ERL fragmentation problem? Recent discussions suggest that punctured keys, as presented by Dan Boneh *et al.* [20], could possibly be used to reduce fragmentation, but the tradeoff with regard to system performance is unclear. Shamir's secret sharing [21] also seems worthy of consideration for possibly reducing as well via the use of polynomials, but this, like all other points of mention, remains a future academic excursion.

Bibliography

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

[2] California consumer privacy act. https://oag.ca.gov/privacy/ccpa. [Online; accessed 24, May 2022].

[3] Aashaka Shah, Vinay Banakar, Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Analyzing the impact of GDPR on storage systems. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, July 2019.

[4] Google Cloud Documentation Maintainers. Google Cloud Documentation: Data Deletion on Google Cloud. https://cloud.google.com/docs/security/deletion. [Online; accessed 28, March 2022].

[5] 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 Stroage Technologies*, FAST'11, page 8. USENIX Association, 2011.

[6] Yan Li, Nakul Sanjay Dhotre, Yasuhiro Ohara, Thomas M. Kroeger, Ethan L. Miller, and Darrell D. E. Long. Horus: Fine-grained encryption-based security for large-scale storage. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, February 2013.

[7] Steven Bauer and Nissanka B. Priyantha. Secure data deletion for Linux file systems. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[8] 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.

[9] Dan Boneh and Richard J. Lipton. A revocable backup system. In *Proceedings of the 6th USENIX Security Symposium*. USENIX Association, 1996.

[10] 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 USENIX Conference on File and Storage Technologies (FAST '05)*, 2005.

[11] 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.

[12] 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.

[13] Myungsuk Kim, Jisung Park, Genhee Cho, Yoona Kim, Lois Orosa, Onur Mutlu, and Jihong Kim. Evanesco: Architectural support for efficient data sanitization in modern flash-based storage systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1311–1326. Association for Computing Machinery, 2020.

[14] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. NFPS: Adding undetectable secure deletion to flash translation layer. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 305–315, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Shuo-Han Chen, Ming-Chang Yang, Yuan-Hao Chang, and Chun-Feng Wu. Enabling file-oriented fast secure deletion on shingled magnetic recording drives. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19. Association for Computing Machinery, 2019.

[16] Dave Hitz, James Lau, and Michael Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, January 1994.

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

[18] 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.

[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] Dan Boneh, Sam Kim, and Hart Montgomery. Private puncturable prfs from standard lattice assumptions. Cryptology ePrint Archive, Report 2017/100, 2017. `https://ia.cr/2017/100`.

[21] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, nov 1979.

Appendix A

Algorithms

Algorithm 1 Computing the offset of the first leaf a node covers

Input: KHT k, level l, offset o

Output: Offset of first leaf covered by the node at $\langle l, o \rangle$

```
procedure START-LEAF($k, l, o$)
    if $l = 0$ then
        return 0                                // Root node covers all leaves.
    end if
    return $o \times k_l$
end procedure
```

Algorithm 2 Computing offset within a level for a target leaf

Input: KHT k, level l, leaf c

Output: Offset of node on l that covers c

```
procedure LEVEL-OFFSET($k, l, c$)
    if $l = 0$ then
        return 0                                // The only node at L0 is the root.
    end if
    return $\lfloor \frac{o}{k_l} \rfloor$
end procedure
```

Algorithm 3 Deriving the value of a descendant node *n*

Input: KHT *k*, level *l*, offset *o*, ancestor node *m*

Output: Value *v* of *n* at $\langle l, o \rangle$

```
procedure NODE-VALUE($k, l, o, m$)
    $v \leftarrow m_{\text{value}}$
    $s \leftarrow$ START-LEAF($k, l, o$)
    for $i \leftarrow m_{\text{level}} + 1, l$ do
        $t \leftarrow$ LEVEL-OFFSET($k, i, s$)
        $v \leftarrow H(v || i || t)$
    end for
    return $v$
end procedure
```

Algorithm 4 Computing coverage for a specified range of leaves

Input: KHT k, start leaf offset o, leaves n

Output: Set of subroots s

```
procedure COVERAGE($k, o, n$)
    $s \leftarrow \{\}$                                        // Empty set of subroots.
    $c \leftarrow o$                                           // Tracks current leaf to cover.
    $e \leftarrow o + n$                                       // First leaf that is not covered.
    for $i \leftarrow |k| - 1, 2$ do                           // Add all subroots up until a full L1 node.
        while $(c \bmod k_{i-1} \neq 0) \wedge (c + k_i \leq e)$ do
            $p \leftarrow$ LEVEL-OFFSET($k, i, c$)
            $s \leftarrow s \cup \{\langle i, p \rangle\}$            // Add node at level $i$, offset $p$.
            $c \leftarrow c + k_i$
        end while
    end for
    while $c + k_1 \leq e$ do                                  // Add all full L1 nodes.
        $p \leftarrow$ LEVEL-OFFSET($k, 1, c$)
        $s \leftarrow s \cup \{\langle 1, p \rangle\}$
        $c \leftarrow c + k_1$
    end while
    for $i \leftarrow 2, |k| - 1$ do                           // Add any remaining subroots.
        while $c + k_i \leq e$ do
            $p \leftarrow$ LEVEL-OFFSET($k, i, c$)
            $s \leftarrow s \cup \{\langle i, p \rangle\}$
            $c \leftarrow c + k_i$
        end while
    end for
    return $s$
end procedure
```