ERASER: Your Data Won't Be Back

Kaan Onarlioglu
Akamai Technologies
Cambridge, MA
www.onarlioglu.com

William Robertson
Northeastern University
Boston, MA
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
Boston, MA
ek@ccs.neu.edu

***Abstract*—Secure deletion of data from non-volatile storage is a well-recognized problem. While numerous solutions have been proposed, advances in storage technologies have stymied efforts to solve the problem. For instance, SSDs make use of techniques such as wear leveling that involve replication of data; this is in direct opposition to efforts to securely delete sensitive data from storage.**

We present a technique to provide secure deletion guarantees at file granularity, independent of the characteristics of the underlying storage medium. The approach builds on prior seminal work on cryptographic erasure, encrypting every file on an insecure medium with a unique key that can later be discarded to cryptographically render the data irrecoverable. To make the approach scalable and, therefore, usable on commodity systems, keys are organized in an efficient tree structure where a single master key is confined to a secure store.

We describe an implementation of this scheme as a file-aware stackable block device, deployed as a standalone Linux kernel module that does not require modifications to the operating system. Our prototype demonstrates that secure deletion independent of the underlying storage medium can be achieved with comparable overhead to existing full disk encryption implementations.

1. Introduction

Secure deletion of data from non-volatile storage is a heavily-studied problem. Researchers and developers have proposed a plethora of techniques for securely erasing data from physical media, often employing methods such as overwriting files containing sensitive data in-place, encrypting data with temporary keys that are later discarded, or hardware features that scrub storage blocks. We refer readers to prior work [25] for an in-depth discussion of previous literature on this topic.

Despite these extensive efforts, advances in storage technologies and characteristics of modern hardware still pose significant difficulties to achieving irreversible data deletion. For instance, Solid State Drives (SSDs) often utilize hardware controllers inaccessible to the outside world. These controllers can redirect I/O operations performed on logical device blocks to arbitrary memory cells in order to implement wear leveling and minimize the effects of write amplification. As a result, many secure deletion methods that base their security on behavioral assumptions regarding older mechanical disk drives are rendered ineffective, because tracking and removing sensitive data in modern settings is often infeasible, and sometimes impossible.

In the face of these emerging challenges, recent research has adapted secure deletion technologies to new applications. For example, Reardon et al. [26] present an encrypting file system that guarantees secure erasure on *raw* flash memory used in smartphones. However, secure deletion remains a challenge on *blackbox* devices such as the aforementioned SSDs, which only allow access to their storage through opaque hardware controllers that translate I/O blocks in an unpredictable manner.

In this work, we present a technique that provides secure deletion guarantees at file granularity, independent of the characteristics of the underlying storage medium. Our approach is based on the general observations in previous work that secure deletion cannot be *guaranteed* on a blackbox storage medium with unknown behavior. Therefore, we instead bootstrap secure deletion using a minimal *master key vault* under the user's control, such as a Trusted Platform Module chip or a smartcard.

Our approach is an evolution of the cryptographic erasure technique proposed by Boneh and Lipton [8]. At a high level, we encrypt every file on an insecure medium with a unique key, which can later be discarded to cryptographically render a file's data irrecoverable. Note that while these keys would need to be persisted to keep the files accessible in the future, they cannot be stored on the same medium together with the files since that would then prevent us from securely deleting the *keys*.

To address this problem, we compress the keys into a single master key that is never persisted to insecure storage, but instead is evicted to the master key vault. To this end, we utilize a key store organized as an n-ary tree (i.e., a tree where each node has up to n children), where every node represents a unique encryption key. We term this key store a *file key tree* (FKT). Keys corresponding to leaf nodes each encrypt a single file stored on the blackbox medium, and in turn parent nodes encrypt their children nodes. This tree hierarchy compresses the master secret to a single encryption key, the root node, which is never persisted

to the blackbox storage but is instead easily evicted to the master key vault. In contrast, the rest of the tree nodes (i.e., encrypted keys) are stored together with the files on the insecure device.

In this model, securely deleting a file from an FKT of capacity $|F|$ involves decrypting $n \log_n |F|$ nodes, regenerating $\log_n |F|$ keys, and re-encrypting the $n \log_n |F|$ nodes with the new keys. During this process, the master key is also securely wiped from the vault and replaced with a fresh one. In this way, the previous path leading to the deleted file will be rendered irrecoverable.

We implemented this technique as a *file-aware stackable block device*, which is deployed as a stand-alone Linux kernel module that does not require any modification to the operating system architecture. As the name implies, our implementation exposes a virtual block device on top of an existing physical device installed on the computer. Users can format this drive with any file system and interact with it as they would normally do with a physical disk. Our block level implementation is able to capture higher-level file system information to identify file blocks, while providing I/O performance significantly better than a file system-level solution.

The contributions of this paper are as follows.

- We present a secure deletion technique called ERASER that builds on the cryptographic erasure primitive. In contrast to previous work, ERASER guarantees performant irrecoverability of deleted files on blackbox storage media with unknown characteristics, such as modern SSDs equipped with opaque hardware controllers. We achieve this property by bootstrapping cryptographic erasure from a cheap, commodity external key vault such as a Trusted Platform Module chip or smartcard.
- We discuss the design choices involved in realizing our approach in practice, and present a prototype implementation of ERASER as a *file-aware stackable block device* for Linux.
- We demonstrate that ERASER can provide full disk encryption on top of secure file deletion, and exhibits comparable performance to dm-crypt, the standard Linux disk encryption solution.

Availability. Source code of ERASER is licensed under GPLv2 and GPLv3, and is publicly available on the primary author's website.

2. Background & Related Work

2.1. Related Work

Secure deletion approaches have been investigated at several different layers of abstraction. We refer the interested reader to a comprehensive classification of prior approaches [25], while in the following we summarize relevant related work.

Hardware Techniques. The lowest point at which secure deletion can be performed is at the physical layer. In the most direct interpretation, secure deletion can be performed through destruction of the physical medium through various means. Scenarios where these methods apply are out of scope for this paper.

Secure deletion can also be performed at the hardware controller. For magnetic media, SCSI and ATA controllers provide a Secure Erase command that overwrites every physical block. Some SSDs also provide such a command. However, this is a coarse-grained approach to secure deletion that is difficult to improve upon since, without knowledge of the file system, controllers cannot easily distinguish data to be preserved from data to be deleted. Furthermore, prior work has shown that hardware-level secure deletion is not always implemented correctly [31].

File System-based Solutions. The next layer of abstraction is at the file system. Here, secure deletion approaches can take advantage of file system semantics, but are potentially restricted by the device driver interface.

One class of techniques is aimed at devices for which the operating system can reliably perform in-place updates (e.g., magnetic disks). Many specific techniques have been proposed, including queuing freed blocks for explicit overwrite [7], [18], [19] as well as intercepting unlink and truncation events for user space scrubbing [19].

Other techniques are intended for devices such as raw flash memory, where there is asymmetry between the minimum sizes of read or write and erase operations (described below). A notable example is DNEFS [26], which modifies the file system to encrypt each data block with a unique key and co-locates keys in a dedicated storage area. Secure deletion is implemented by erasing the current key storage area and replacing it with a new version. During this replacement, keys corresponding to deleted data are not included in the new version.

However, a fundamental underlying assumption of these approaches – that the operating system can directly read or write physical blocks as in the case of magnetic hard drives or raw flash memory – is not valid for modern storage devices such as SSDs as we describe below.

User-level Tools. User space is the highest layer of abstraction from which secure deletion can be attempted. These approaches are restricted to the file system API exposed by the operating system to accomplish their task. One example of such an approach is Secure Erase [16], an application that simply invokes the Secure Erase command on a storage controller. However, as discussed above, this is not a reliable secure deletion mechanism.

User-level tools can also attempt to explicitly overwrite data to be securely deleted [12], a popular approach first proposed by Gutmann [15]. However, these approaches assume that overwriting a block using the interface provided by the operating system guarantees that all copies of that data on physical storage will be overwritten on the underlying physical medium.

A third user space secure deletion approach is to fill the free space of a file system [6], [14]. The motivation for this approach is to proactively overwrite remnants of potentially sensitive data on storage left in the free block

pool. However, this approach is also limited by the operating system actually providing the capability to overwrite *all* free blocks on storage, as well as the system's ability to expose all physical blocks to user space.

Cryptographic Erasure. Along a different axis, numerous techniques utilize cryptographic erasure as a fundamental primitive. Put simply, these techniques reduce secure deletion of data to secure deletion of a key encrypting that data. Under computational hardness assumptions, encrypted data without the corresponding key is infeasible to recover. Prominent examples of this include Boneh's secure deletion approach for offline data such as tape archives [8], Lee's secure deletion approach for YAFFS [21], DNEFS [26], and TrueErase [9], [10]. Note that, these works are not compatible with flash translation layers implemented in opaque hardware controllers, excluding them from use on typical SSDs.

Another approach to cryptographic erasure was proposed by Tang et al. [30]. In CleanOS, sensitive data on mobile devices is encrypted and the corresponding key is evicted to the cloud. The fundamental assumption underlying this work is that the cloud is more trustworthy than the user's device, which is not always the case. In another work, Lacuna [11] leverages cryptographic erasure to implement ephemeral channels that eliminate privacy leaks into operating system drivers.

Yet another example of cryptographic erasure was proposed by Swanson et al. [29], this time at the controller level. Here, a cryptographic key is used to encrypt the entire physical device, and this key is stored within a dedicated memory also located on the device. Secure deletion is performed by replacing this key, resulting in a coarse-grained secure deletion of all data on storage.

Reardon et al. [27] also present a graph theoretic approach to analyzing and proving the security of any tree-like approach to secure deletion involving encryption and key wrapping. They provide an implementation of an instance of this class of approaches as a B-tree that can provide file-level deletion granularity, and exhibits the potential for good performance when combined with a suitable caching policy. This work is closely related to ours, and therefore, we defer a direct comparison between them to Section 6.

2.2. Flash Translation Layers

Flash memory is a common storage technology due to its low power consumption, density, and efficient random-access characteristics. In a significant departure from classical storage technologies such as magnetic hard disks, flash memory possesses an asymmetry between the sizes of read and write operations versus the size of erasure operations. In particular, data is read and written at *page* granularity (e.g., 4K), but is erased at an *erase block* granularity (typically 256K). Furthermore, flash memory cannot be written to unless the page, and its enclosing erase block, has first been erased. Since this operation incurs significant wear, *wear leveling* is performed wherein erasure operations are evenly distributed across flash erase blocks in order to maximize the device's service lifetime. This leads to the phenomenon of *write amplification*, where one logical I/O operation leads to multiple physical I/O operations.

For raw flash devices intended to be directly exposed to an operating system, wear leveling is expected to be performed by the device driver. However, devices such as solid-state drives (SSDs) do not expose this low-level interface. Instead, a *flash translation layer* (FTL) is interposed to provide a traditional sector-based interface to the operating system. For an SSD, the FTL is implemented within the hardware controller, and in such cases the operating system does not have direct access to physical flash pages, erase blocks, or visibility into the wear leveling process. In fact, in order to accommodate expected wear and account for failed erase blocks, modern SSDs are typically over-provisioned by 25%.

Since FTLs obscure physical flash erase blocks and wear leveling leads to write amplification that results in significant amounts of duplicated data, existing secure deletion techniques are incompatible with such devices.

2.3. Motivation

While prior secure deletion approaches work under certain circumstances, almost all unfortunately do not address common cases where the operating system cannot guarantee that physical blocks are not duplicated on storage, or that logical blocks map directly to physical blocks, as in the case of FTL-based devices such as SSDs. Approaches that remain, such as whole-device secure erase commands or cryptographic erasure [29] only operate at the coarsest granularity possible. Others require integration of secure deletion capabilities directly into the FTL layer (e.g., [17]), through the modification of hardware controllers.

Our work aims to fill this important gap for *arbitrary* storage devices, by satisfying the following design goals.

1) Secure deletion must not rely on the assumption that blocks are not duplicated without its knowledge.
2) Secure deletion must not rely on the assumption that logical block addresses map one-to-one to physical block addresses.
3) Secure deletion must operate at a useful level of granularity – in our case, at the file level.

2.4. Threat Model

The threat model we assume in this work is essentially a notion of forensic security. That is, while the system computes over sensitive data we assume that an adversary is not present on the system and cannot examine or tamper with this data. We also assume a trusted computing base (TCB) composed of a subset of the system's software that includes the kernel and a small set of high-privilege user space utilities. The TCB also includes a subset of the underlying firmware and hardware, in particular a secure storage area described later in the paper such as a Trusted Platform Module (TPM) chip or smartcard. However, storage controllers are considered to be untrusted, and no assumptions

are made as to the kind of physical medium used in the system. We assume that an adversary can later gain access to the system, up to and including physical access. Regardless, the secure deletion approach we describe in the remaining sections guarantees that attackers cannot recover data that was deleted during prior computation.

3. System Design

3.1. Naïve Approach

A straightforward approach to secure file deletion using cryptographic erasure is to simply generate random encryption keys for each file. Any data written to storage would be first encrypted with its associated file key, and decrypted when read from storage. Securely deleting a file is then reduced to securely deleting the corresponding file key – i.e., cryptographic erasure where, under computational hardness assumptions, it should be infeasible for an attacker to recover the data without the key.

This approach, however, has a fatal flaw: there is no way to assure that file keys are securely deleted themselves. That is, file keys must be persisted to storage across system reboots or failures, and thus must themselves be encrypted with a *master key*. This is clearly a recursive problem. Therefore, the approach must rely, instead, upon a trusted element to serve as secure storage for the master key. We term this master key secure storage the *master key vault*, which must satisfy the following properties. The vault must *(i)* be large enough to store a master key, *(ii)* allow the system to perform encryption and decryption operations using the stored master key, and *(iii)* allow the system to update the stored master key with a new key.

This leads to a second problem: the simple two-level hierarchy described above implies that deleting a single file requires re-encrypting all file keys. To understand why, consider that on modern storage devices data might be persisted to multiple physical locations due to phenomena such as flash wear leveling, and that such processes are completely outside the control of an operating system. Therefore, in order to ensure that file data is irrecoverable, the master key must itself be rotated, and the old key securely deleted from the vault such that there is no computationally feasible way for an attacker to decrypt block data recovered from physical storage. Since the master key must be rotated, all file keys must be re-encrypted before being persisted to disk, leading to a phenomenon we term *encryption amplification*. This is an expensive operation and should be avoided for any practical system (see Section 4 for a concrete example).

3.2. File Key Trees

To address the above problems identified in the naïve approach, ERASER's design incorporates two key elements: *(i)* a master key vault, and *(ii)* a *file key tree* (FKT). The master key vault has the properties described above, which allows for master keys to be rotated with secure deletion of the old key. The FKT, on the other hand, avoids the problem of encryption amplification by bounding the number of keys that must be re-encrypted each time the master key is rotated.

An FKT is an n-ary tree – i.e., a tree where each node has up to n children – of height m. At the root is the master key, which is stored in the master key vault, is never released from the system TCB, and is never persisted to other storage in any form. Internal nodes of the tree correspond to randomly-generated encryption keys. Each node key encrypts the keys of its children. Leaves of the tree correspond to file encryption keys. An example of an n-ary FKT with $m = 2$ is shown in Figure 1.

FKT Space Complexity. To represent $|F|$ files, an FKT with at least $|F|$ leaves must be created. Therefore, the size of an FKT is bounded by

$$O\left(n^{\lfloor \log_n |F| \rfloor} + |F|\right).$$

This is simply the number of internal nodes required to represent $|F|$ leaves in an n-ary tree plus the leaves themselves. In practice, the root key will be evicted to the dedicated master key vault, while the remaining levels of the FKT will be persisted to disk.

FKT Operations & Time Complexity. Accessing a file encrypted using an FKT involves collecting a chain of encryption keys from the corresponding FKT leaf node to the master key and, performing a series of decryption operations to recover the file encryption key. Therefore, the number of decryption operations to obtain access to a file is bounded by

$$O\left(\lceil \log_n |F| \rceil\right).$$

Deleting a file, similarly to file access, first requires collecting a chain of encryption keys from the corresponding FKT leaf node to the master key. However, the next step of this process is to: *(i)* randomly generate new encryption keys for each node along the path to, and including, the master key node; and, *(ii)* re-encrypt the existing keys at direct children (i.e., non-recursively) for each node along the previously identified path in the FKT. Therefore, this operation's time complexity is bounded by

$$O\left(n\lceil \log_n |F| \rceil\right).$$

This process is explained in a concrete example in Figure 2 for $n = 2$, $|F| = 4$.

4. Implementation

We implemented the general secure deletion approach presented in the previous section in a tool called ERASER, which operates at the block I/O layer of Linux but provides secure deletion guarantees at file granularity. ERASER does not require modifications to Linux, and can be distributed and deployed as a stand-alone kernel module that works on any Linux distribution, with any file system. Our prototype could easily be extended from user space to support various types of secure external storage such as TPM chips,

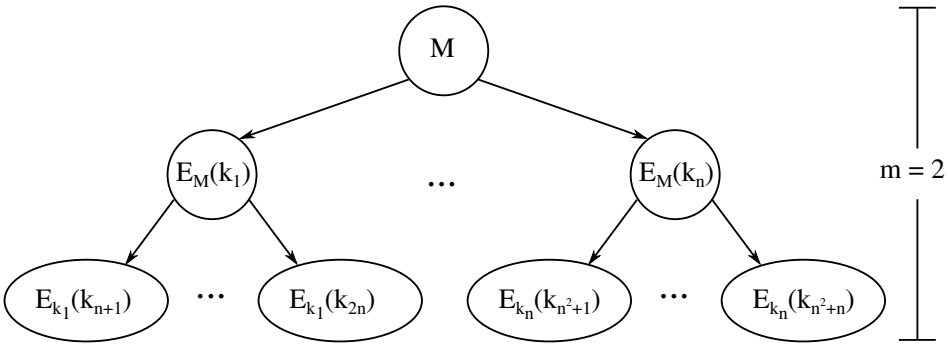

Figure 1. Structure of an n-ary FKT with $m = 2$. The root node is represented by a master key M stored in a secure master key vault. Each internal node contains a key encrypted by the parent key. Leaf nodes correspond to file encryption keys.

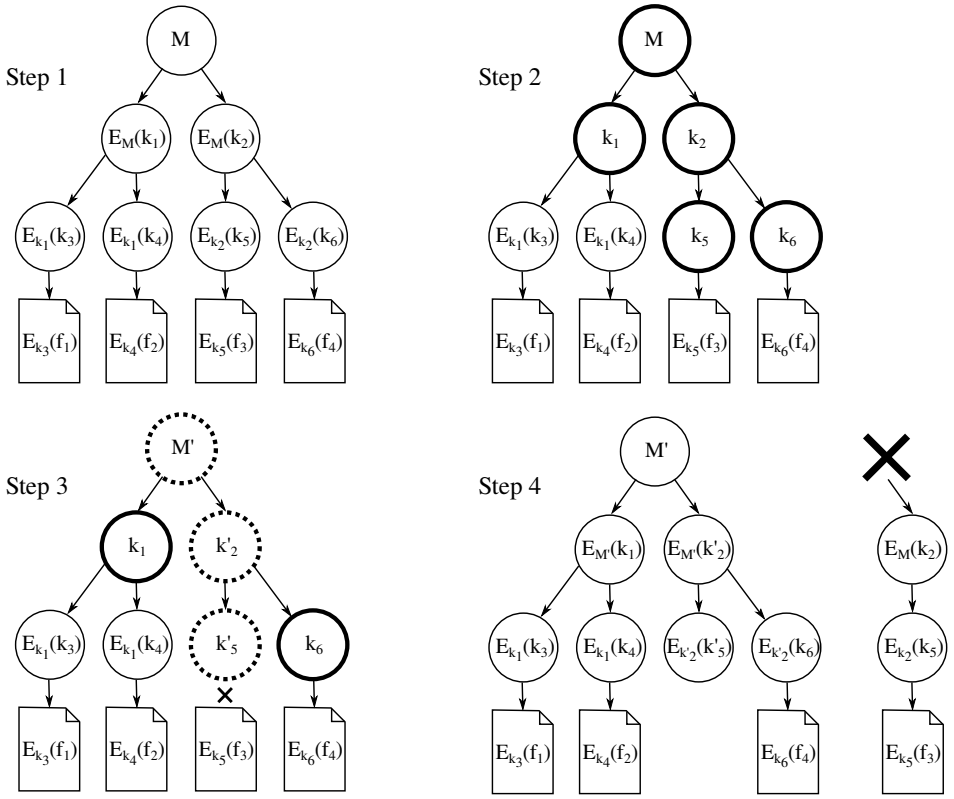

Figure 2. Secure deletion using an FKT with $n = 2$, $|F| = 4$. **Step 1:** The initial state of the tree contains encryption keys for four files. **Step 2:** When the user decides to delete file 3, a traversal of the FKT from the corresponding leaf node for file 3 to M is performed. Starting below the master node, each node's key is decrypted using the parent's key. Additionally, all other *direct* children of the current node are decrypted. Decrypted nodes are shown here in bold. **Step 3:** Keys along the *direct path* from file 3's leaf node to the master key node are randomly regenerated. These nodes have a dotted outline. The old master key M is securely deleted from the vault, and a new master key M' is stored. **Step 4:** Keys at direct children of nodes on the path from Step 3 are re-encrypted to obtain the new FKT, which is persisted to disk. Nodes from the pruned branch as it existed at Step 1 might remain on insecure storage, but since M has been erased it is computationally infeasible for an attacker to decrypt data along that path.

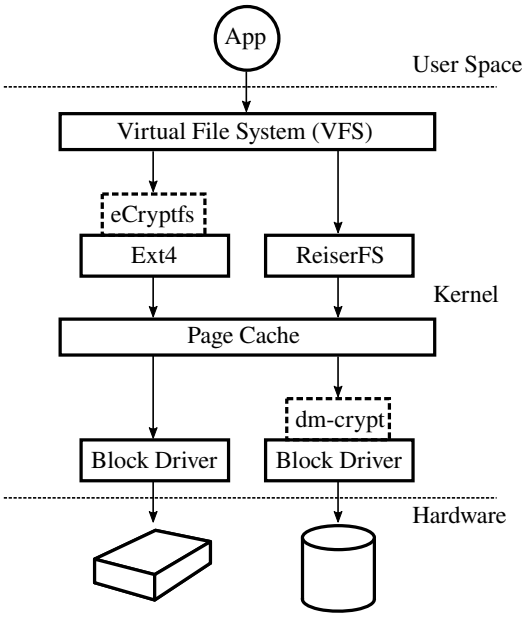

Figure 3. An overview of Linux I/O layers.

smartcards, or tokens with secure storage capabilities. In our implementation, we specifically utilize commodity TPM chips that are present on many modern motherboards.

Here, we first discuss various implementation alternatives to realize our approach, and explain our decision to choose the block I/O layer for our prototype. We then present technical details and optimization considerations to build a performant and practical implementation. We refer readers to ERASER's source code for other low-level implementation details that we omit for brevity.

4.1. Alternatives & Our Philosophy

As discussed, there is a myriad of tools and techniques that implement secure deletion capabilities at various layers of a computer system. To assess the advantages and drawbacks of each of these options, we first briefly describe the processing of a file access request through the various Linux I/O subsystems.

In Linux, a file access initiated by a user space application reaches the kernel through one of the I/O-related system calls, such as `read`, or `write`, which then passes this request to the *Virtual File System (VFS)*. The VFS is a unifying abstraction layer for file system operations, while the actual I/O operations defined by its API are implemented by concrete file system instances such as `ext4`. File blocks processed by a file system go through the *Page Cache*, where they are cached to improve I/O performance on repeated access requests. Finally, I/O requests are passed to the block I/O layer, and are served by specific device drivers that control the storage hardware. This entire procedure is illustrated in Figure 3.

Unsatisfactory Solutions. While it is relatively easy to develop user space solutions instead of trying to understand operating system internals, the I/O-related system calls offer minimal control over how data blocks are processed at lower layers, limiting the effectiveness of such solutions. Likewise, in this work, we refrain from directly modifying concrete file system implementations or device drivers. While implementing our approach at those layers is possible, choosing a specific instance to adapt to our needs would limit the impact of our system. Or else, modifying and maintaining every single file system or driver available in Linux would be a high-effort and bug-prone affair.

Despite the issues mentioned above, the file system layer is still a natural place to enforce secure deletion of files. By definition, the file system is already aware of all the data blocks corresponding to any given file, and also has full control over file metadata, all of which significantly ease development. One solution that alleviates the issues tied to working with a specific file system, while also leveraging the advantages of the file system layer, is utilizing a **stackable file system**. These special file systems reside on top of another file system and transparently interpose on the passing I/O requests, presenting a viable option to implement secure deletion. For instance, eCryptfs [3] is a stackable encrypting file system distributed with Linux, and could easily be adapted to our approach. Unfortunately, stackable file systems often come with a significant performance overhead. For example, recent research [24] and benchmarks [20] show that eCryptfs performs considerably worse than block-layer encryption. Since building a practical system is one of our goals, this is not an acceptable solution.

Another seemingly viable alternative is to implement our system inside the **page cache**. However, examining the kernel internals reveals that page cache functions that manipulate file blocks (e.g., `pageread` and `pagewrite`) are actually provided by file systems. Furthermore, Linux gives applications the ability to perform *direct I/O* that bypass the page cache. As a result, the page cache is also not a suitable layer for our purposes.

Our Solution. In light of these considerations, we decided to implement ERASER at the block device level in a **stackable block device driver** (i.e., at the same layer as dm-crypt in Figure 3). Similar to how stackable file systems operate, stackable drivers intercept block I/O requests before they reach the underlying drivers, and allow us to manipulate them as necessary. The main advantage of a block-layer approach is its performance (e.g., compare dm-crypt's performance to eCryptfs [20]).

At a first glance, it is not clear how file-level information could be gathered at the block layer, or how physical sectors on a device could be matched to logical file blocks. ERASER closes this gap between the file system and block device layers by leveraging the property that Linux represents every file system object by a common data structure provided by the VFS, the `inode`, regardless of the file system implementation. In this way, we can avail ourselves of the performance benefits of operating on low-level device blocks, while still retaining a high-level understanding of the file system. Moreover, ERASER works under any Linux-native file system and is compatible with any physical block device.

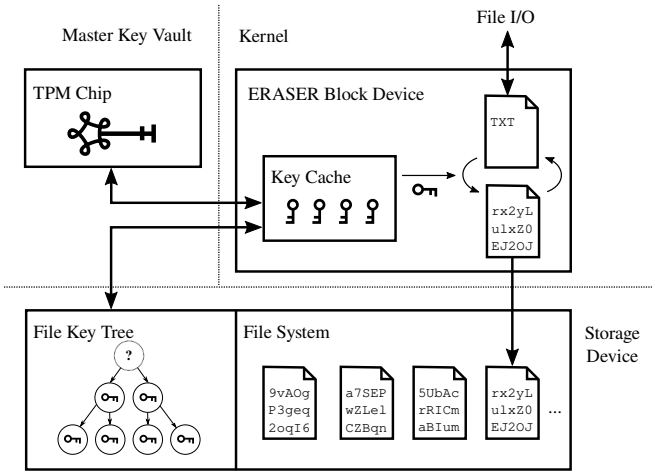

Figure 4. An overview of ERASER. Our prototype implementation utilizes a TPM chip as its master key vault.

4.2. System Overview

We implemented ERASER using *device-mapper* [2] in a stackable block device driver. Device-mapper is a standard Linux kernel framework that allows users to create stackable drivers, and is used in technologies such as dm-crypt, LVM, software RAID, and Docker. It maps existing physical block devices onto new ones, and exposes these virtual devices to user space via new device nodes, often found under `/dev/mapper/*`. Users can then interact with these device nodes in the usual way by formatting them with a file system of their choice.

A high-level view of ERASER is illustrated in Figure 4. ERASER organizes file encryption keys in an FKT as described, and stores them in a reserved section of the underlying storage device. The master key, however, is never persisted to this device; instead, it is confined to an external secure store. Specifically, in our implementation, we store it in the NVRAM area of a TPM chip installed on the machine. ERASER then intercepts all block I/O operations in flight, identifies which files those I/O blocks belong to, and retrieves the appropriate keys to encrypt or decrypt the file contents on the fly. When a file is deleted, its associated key is discarded. Finally, the newly generated keys are written to the key store, and a fresh master key is synced to the TPM chip, overwriting the obsolete key.

4.3. I/O Manipulation

The kernel represents and tracks in-flight block I/O operations with a data structure called a `bio`. Through the device-mapper framework, each `bio` destined for an underlying physical device is first handed to ERASER, where we can freely manipulate them before passing them on to the next device driver in the stack.

Identifying Files. The first task ERASER needs to perform is detecting whether a `bio` corresponds to a file system operation. Thanks to the way Linux handles pages of a file during I/O and the VFS layer which necessitates that every file system object have an `inode` associated with it, this task is possible without modifying the upper kernel layers to propagate this information downwards.

Since there is a one-to-one mapping between inodes and files, our implementation uses inode numbers to uniquely identify files and find their corresponding encryption keys. Whenever ERASER receives a `bio`, it iterates over all memory pages (i.e., data buffers in volatile memory) it points to. Linux provides another related object per file, called an `address_space`, that describes the mapping between physical blocks on a disk, pages in memory, and the inode owning these. By walking through this structure, ERASER can match every page in a `bio` to a specific inode, check whether the inode at hand corresponds to a file, and subsequently identify the correct encryption key based on the inode number. Otherwise, if the pages are found to have no corresponding inode or an inode that represents a file system object other than a file, that `bio` is simply remapped and sent to the actual underlying device without further processing.

Writing Files. Once a `bio` corresponding to a file write is identified, ERASER needs to retrieve the appropriate key, encrypt the contents, and perform the write to the underlying device. However, simply iterating over the memory pages pointed to by a `bio` and encrypting them in-place is not sufficient. This is because the same memory pages representing the write buffers are often also present in the page cache. Thus, directly encrypting them would result in ciphertext being served to user space from the cache with future I/O requests, without our driver having an opportunity to decrypt them. Even if we attempt to intercept cache hits, it would be sub-optimal to decrypt the same contents with every read.

To address this issue, ERASER makes a clone of the original `bio` and all of its pages, and instead encrypts the copied pages. Next, the cloned `bio` is asynchronously submitted to the underlying device, while the original I/O request is being stalled. Once ERASER receives notification of a completed disk write through a callback, it marks the original `bio` as completed as well, which automatically signals the upper layers of a successful disk write. In this way, the cached data remains untouched, and subsequent file reads that result in cache hits do not require repeatedly decrypting the same data buffers.

Reading Files. Handling of `bio` objects that represent file reads is similar, with one difference. When ERASER first intercepts the `bio`, the pages it points to are empty, ready to be filled with data read from the disk. Therefore, ERASER first needs to initiate the actual disk read, and decrypt the data only once the operation is complete.

This is achieved by, once again, cloning the `bio`, and submitting this clone for I/O to the underlying device while the original is stalled. However, this time, it is not necessary to allocate separate memory pages for the clone; instead, the clone points to the original memory pages. Once we receive notification of a completed read, ERASER retrieves the appropriate key, decrypts the contents in place, and signals completion of the original `bio`.

Cryptographic Operations. ERASER uses AES-256 in CBC mode to encrypt file blocks. Every file is given a unique initialization vector (IV), stored together with the keys in the FKT. Since encryption is performed on a page-by-page basis and pages of a file could be read or written in any order, a page IV is derived from the file's unique IV and the file offset of the processed page. Finally, all random data used by ERASER to regenerate encryption keys and IV after a file deletion is generated using AES-256 in CTR mode. This random stream is seeded by a key from the kernel's cryptographically secure random byte pool, and the cipher is reseeded after every 1MB of data output.

4.4. Intercepting File Deletions

When an ERASER partition is created, its FKT is initialized with randomly generated keys for all inodes[1], and the system is ready for use. Consequently, our system does not need to track file creation events. Instead, we only monitor *file deletions*, discard the appropriate keys in the FKT as discussed in Section 3, and immediately generate new keys for freed inodes, to be used by the next file that is assigned the same inode number.

While this simplifies our implementation, intercepting file deletion events from the block layer is still not a trivial task. In particular, because a file deletion often only involves changes to file system metadata, and no I/O operations are performed on file blocks, the block I/O layer remains oblivious to file deletions. We address this challenge with the help of another Linux kernel framework, *Kernel Probes (kprobes)* [4], which allows users to hook into the kernel address space. We utilize this capability to trap execution at the entry point of the `vfs_unlink` function, a choke point inside the VFS for all deletion operations. Next, in our hook, we access the original function's arguments from the CPU registers, retrieve a pointer to the deleted inode, and check whether it represents a file object residing on an ERASER partition. Once it is confirmed that a file on a relevant device is being deleted, we then trigger the secure deletion process and generate fresh keys for the freed inode.

4.5. Key Storage & Management

While ERASER's key organization is based on the high-level FKT design presented in Section 3, we also employ optimizations specific to our implementation.

Due to the large size of an FKT, the majority of the keys are stored on the disk at any given time and are only accessed when required for a file access or secure deletion. Because of this, the parameter n of the FKT should be chosen to optimize disk I/O performance. In our implementation, every node of the tree contains a 256-bit encryption key and 128-bit IV, for a total size of 48 bytes. To ensure that we perform disk I/O operations on block boundaries, we set n to the maximum number of tree nodes that can fit into a single block, i.e., $\lfloor \frac{4096}{48} \rfloor = 85$ for a system with 4K logical blocks. In this way, we can perform disk I/O on all children of a node with a single block access. This also allows us to perform cryptographic operations on the blocks with a single pass, because page sizes are often equal to or multiples of block sizes. Other configurations are also possible as long as n is chosen so that nodes fall within block boundaries.

Note that the structure of an FKT can be estimated fairly accurately at the time of system initialization, and the tree structure will remain static throughout the system's life. In our approach, there is a leaf node corresponding to each file. The number of files is limited by either the number of inodes a file system can support on a device of given capacity or, in the case of file systems that allocate inode indices dynamically, by the space reserved on the device for the key store. With this knowledge of how many leaves are going to be available in the FKT at any given time, we can further optimize the tree structure for space efficiency.

We do this by first calculating the minimum tree height required based on the number of inodes we need to support, and then decreasing the fan-out of the root node to a value smaller than n in order to cull unused, empty subtrees of the root. For instance, an Ext4 file system created on a device with a 100GB capacity would default to allocating 6,553,600 inodes. To create a tree with 6,553,600 leaves working back towards the root (with $n = 64$ to simplify calculations), we would need $\frac{6553600}{64} = 102400$, $\frac{102400}{64} = 1600$, and $\frac{1600}{64} = 25$ nodes at each level. Consequently, a fan-out of 25 for the root would be sufficient for this configuration. The recurrence relationship for calculating the total number of tree nodes required to support $|F|$ files is given below, excluding the root node stored externally. As a result, our implementation reserves $48R(|F|)$ bytes of storage space for a file system that can represent a maximum of $|F|$ files.

$$R(|F|) = \begin{cases} |F| + R(\lceil \frac{|F|}{n} \rceil), & \text{if } |F| > n \\ |F|, & \text{if } |F| \leq n \end{cases}$$

Recall that our approach requires $\log_n |F|$ disk accesses (i.e., the height of the tree) to retrieve or discard the required keys with each file access and deletion. In order to mitigate the I/O overhead caused by this necessity, our approach is twofold. First, we always keep the decrypted nodes in levels 1 and 2 in memory (e.g., following the previous example with 6,553,600 leaves and $n = 64$, this would require less than 7MB of memory). Modified nodes are written to disk periodically. Next, we employ a caching strategy for the leaf nodes so that the keys for frequently accessed files are available in memory. A dedicated kernel thread periodically synchronizes dirty cache entries to their disk blocks and evicts old cache entries. Note that we experimented with various caching strategies and data structures for searching the cache efficiently. Our measurements show that having a cache, as opposed to always reading the keys from the disk, results in a significant performance gain. However,

1. For most Linux file systems, the maximum number of possible inodes can be determined or predicted based on storage size. For exceptions where the inode values can grow dynamically, a fixed value needs to be specified by the user when formatting a drive.

fine-tuning the cache organization had no discernible impact on performance. This indicates that ERASER's performance is primarily I/O bound as expected, and that cache searches are overshadowed by I/O operations.

4.6. Master Key Vault

ERASER stores the master key in the NVRAM of a TPM chip. This enables us to reliably discard (i.e., overwrite) an obsolete key when the master key needs to be regenerated after a file deletion, and also provides a strong defense against unauthorized retrieval of the master key.

While it would be possible to interact with the TPM chip directly from within the kernel, ERASER instead utilizes a user space helper application to access the NVRAM. This is a conscious design choice to make it possible to extend the system to support different secure storage modules in the future, without requiring modifications to the kernel core. ERASER coordinates with this helper using `netlink`, a standard Linux mechanism for kernel-to-user space communication.

While stored inside the NVRAM, or in transit between the user space and kernel, the master key itself needs to be protected. This is accomplished by encrypting the master key with another encryption key derived from a user password. This process is as follows: 1) When setting up a new ERASER instance, the user enters a password that is fed into a key derivation function to produce a cryptographically strong encryption key. 2) The kernel randomly generates the first master key. 3) The kernel uses the password-derived key to encrypt the master key, and then transfers it to the NVRAM using the user space helper.

As a result, the master key is always encrypted when residing inside the TPM chip, accessed by the user space helper, or in transit through netlink. That is, the master key never leaves the TPM chip in plaintext, and is also never exposed inside the user space helper. When the kernel module generates a new key, it is encrypted in the kernel space, and only then transferred through the user space helper. At any given time, only the kernel has access to the master key.

Practicality. We opted to utilize a TPM chip to implement the master key vault due to its wide availability on modern computers. Specifically, many major vendors now provide TPM chips by default on their business-oriented systems. Further adoption of TPM chips is motivated by the rapid deployment of security technologies that require them (e.g., recent versions of Microsoft Windows security tools, such as BitLocker, require TPM chips to work at their full capacity [22], [23]). External TPM chips can also be added to many modern motherboards at a low cost [2]. When modifying the hardware configuration is not possible, tokens with similar secure storage properties can be used instead (e.g., YubiKey is a suitable USB option and costs $40 as of this writing). All in all, ERASER's secure storage requirement could easily and inexpensively be fulfilled in many use cases.

Write Wear and Security. Another important consideration pertaining to external secure storage modules is the write wear caused to the medium by key rotations. In our case, we can avoid early exhaustion of the NVRAM write wear capacity by limiting the key rotations to an acceptable frequency. Specifically, it is not strictly required to refresh the master key and write it to the TPM chip *every time* a file is deleted. This operation can be performed less frequently, but periodically, as configured by the user.

An important implication of limiting storage wear in this way is that ERASER's security guarantees also get weaker with less frequent key rotations. Recall that ERASER only guarantees secure deletion once the old master key is discarded. Delaying this process extends the time frame in which the files deleted after the most recent key refresh can be successfully recovered. Therefore, users should choose an appropriate key refresh frequency according to their applicable threat model. For instance, as little as one rotation at every reboot could be sufficient for a home user, while a system storing sensitive business data could write the new key to the TPM chip more frequently, every hour.

4.7. Encrypting Non-File System Blocks

ERASER provides security guarantees similar to standard file encryption tools as a side benefit (recall that the master key is further protected with a user password). However, this is still inferior to full disk encryption, because non-file disk blocks (e.g., the file system's internal blocks) remain unencrypted. This would necessitate a user desiring both full disk encryption *and* secure deletion to run ERASER on top of another disk encryption solution, such as dm-crypt, hurting disk performance.

To address this limitation, we extended ERASER to provide full disk encryption for non-file blocks as well. In short, ERASER operates in file encryption mode, as described in Section 4.3, if the I/O request is for a file block. In all other cases, it performs regular disk sector encryption using a fixed key generated on system initialization and protected with a user password. The IVs in this mode are derived from disk sector numbers, using the "ESSIV" method [13]. In this way, ERASER becomes a full replacement option for other disk encryption solutions, offering secure deletion guarantees on top of the usual confidentiality characteristics of disk encryption.

4.8. Managing ERASER Partitions

Users interact with ERASER through a user space application that allows them to format physical devices to create the required internal metadata. During this setup process, users are required to configure a password from which encryption keys are derived for securing the master key while it is being transported from the TPM chip to the kernel, and also to encrypt non-file system blocks.

2. As of this writing, an Amazon store search for external TPM chips returns options ranging from $10 to $30.

Later, ERASER partitions can be activated with this tool to expose the securely-deleting virtual device node by supplying the correct password. In the same way, users can view active instances of ERASER, made available by the driver through a `/proc` node, and deactivate them when no longer needed. Through this application, users can also configure ERASER to use any of the supported vault devices for master key storage.

5. Evaluation

Many of our design and implementation choices were geared toward achieving good I/O performance on commodity computers, to build a system with practical impact. Here, we present two sets of experiments to evaluate the performance overhead of ERASER, and compare it to ordinary full disk encryption. Experiments were performed on a system with an Intel i7-930 2.2GHz CPU, 9GB of RAM, running Arch Linux x86-64 with an unmodified 4.17.0 kernel. The storage device used was a Samsung 950 PRO SSD with 1TB capacity, formatted with Ext4 using the default file system settings. Tests were run directly on the hardware, without virtualization.

Note that the prototype we evaluate in this section also implements the non-file system block encryption extension described in Section 4.7, and therefore, functions as a full disk encryption solution as well.

5.1. I/O Benchmarks

To observe how ERASER impacts the I/O performance of the underlying storage device, we first put our system under stress using the popular disk and file system benchmark Bonnie++ [1]. For file I/O tests, we configured Bonnie++ to *write* and *read* 20×1GB files. This size was chosen to be more than twice the system RAM, following the tool's recommendation. Next, file *creation* and *deletion* tests were performed with 512×1024 small files each containing 512 bytes of data, distributed among 10 directories. All tests were also repeated using dm-crypt, the standard Linux full disk encryption solution. While our discussion will focus on comparing ERASER with dm-crypt, we also provide benchmark results obtained without running either, as a baseline. The results are shown in Table 1.[3] These results were averaged over five runs, and the maximum relative standard deviation we observed was below 2% in all cases.

The results reveal that when performing reads and writes on a small number of large files ERASER exhibits similar performance to dm-crypt, with the overhead staying below 1%. This is not surprising; once ERASER obtains the encryption key for the processed file with a one-time performance hit, the remaining task of encrypting and decrypting the file blocks in-flight is nearly identical to how dm-crypt performs disk block encryption. However, in the file creation tests, ERASER incurs a more noticeable performance impact. This is likely due to the fact that ERASER now needs to perform a larger number of additional I/O operations to repeatedly access its key store, and decrypt the corresponding FKT nodes to obtain keys corresponding to each newly created file.

A significant performance hit is observed during file deletions, where ERASER falls behind dm-crypt by 20%. This outcome is in line with our expectations; a file deletion is costly: ERASER intercepts the unlink system call, performs multiple accesses to the FKT, and replaces all involved keys with freshly generated ones, also encrypting and writing them back to the key store if there is cache contention. Despite this drawback, the actual number of files processed per second by ERASER remains considerably high. We next explore how ERASER performs with small file operations in more detail and show that the relatively high deletion overhead does not significantly impact real-life workloads.

5.2. Tests with Many Small Files

Prompted by ERASER's relatively high performance overhead when dealing with many small files under benchmark conditions, we next investigated how it would perform in more realistic scenarios. We chose six tasks performed on a large directory tree – the Linux kernel source code – and measured the time elapsed to complete each task. During these tests, our entire Linux setup (i.e., not only the kernel source tree) was installed on an ERASER partition to reflect a realistic use case.

Tests were performed first with ERASER, and then dm-crypt. Measurements on a vanilla system with no disk encryption are also provided as a baseline. Our tests included the following tasks: *(i)* Unpacking the XZ-compressed source code archive, *(ii)* making a copy of the directory tree, *(iii)* deleting the directory tree, *(iv)* grepping the entire directory for a static term, *(v)* computing an MD5 hash over all the files, and finally *(vi)* compiling the kernel. All tasks were chosen to include a large number of file operations, including reads, writes, deletions, and new file creations. Certain tasks such as kernel compilation combined small file I/O with a CPU-bound component to cover different scenarios. The results are shown in Table 2. These results were averaged over five runs, and the maximum relative standard deviation observed was below 5% in all cases. Operating system caches were dropped between tests to ensure that measurements were not affected by prior runs.

These results confirm our findings from the Bonnie++ benchmarks that ERASER has a noticeable file deletion overhead, this time manifesting itself at 21% during the directory removal task. However, we point out that, in terms of the time elapsed, real-life impact of this performance loss is measured in **less than a second**. In all other tasks, ERASER performed comparably to dm-crypt, and surpassed it in certain cases. However, this should not be taken to mean

3. In all tests we measured higher write speeds than reads. Although unexpected, Internet sources indicate that this is an issue common to SSDs by this vendor, potentially due to a firmware quirk. Notwithstanding the reasons, we point out that this issue does not have any bearing on the relative performance overhead observed between the three test setups in our evaluation.

Table 1. DISK I/O AND FILE SYSTEM PERFORMANCE OF ERASER COMPARED TO FULL DISK ENCRYPTION WITH DM-CRYPT. BENCHMARK RESULTS ON AN UNENCRYPTED DEVICE ARE ALSO PRESENTED AS A BASELINE.

	No Encryption	dm-crypt		ERASER		
Bonnie++ Tests	Performance	Performance	Overhead vs. No Enc.	Performance	Overhead vs. No Enc.	Overhead vs. dm-crypt
Write	255300.00 KB/s	254990.00 KB/s	0.12 %	253530.20 KB/s	0.70 %	**0.58** %
Read	213778.00 KB/s	142174.20 KB/s	50.36 %	141747.60 KB/s	50.82 %	**0.30** %
Create	37183.60 files/s	35850.80 files/s	3.72 %	34266.00 files/s	8.52 %	**4.63** %
Delete	59418.80 files/s	59098.00 files/s	0.54 %	49230.80 files/s	20.69 %	**20.04** %

Table 2. TIMED EXPERIMENTS WITH THE LINUX KERNEL SOURCE CODE DIRECTORY TO COMPARE THE SMALL-FILE PERFORMANCE OF ERASER TO FULL DISK ENCRYPTION WITH DM-CRYPT. TESTS RESULTS ON AN UNENCRYPTED DEVICE ARE ALSO PRESENTED AS A BASELINE.

	No Encryption	dm-crypt		ERASER		
Kernel Source Tests	Time (s)	Time (s)	Overhead vs. No Enc.	Time (s)	Overhead vs. No Enc.	Overhead vs. dm-crypt
Unpack	10.60	**10.84**	2.26 %	**11.39**	7.45 %	**5.07** %
Copy	11.44	23.59	106.21 %	22.61	97.64 %	−**4.15** %
Remove	3.26	**4.17**	27.91 %	**5.04**	54.60 %	**20.86** %
Grep	11.11	25.18	126.64 %	24.12	117.10 %	−**4.21** %
MD5 Hash	10.39	24.20	132.92 %	22.20	113.67 %	−**8.27** %
Compile	1564.13	1564.15	$<$ 0.01 %	1568.13	0.26 %	**0.26** %

that ERASER is faster than dm-crypt. Instead, we conclude that they perform similarly in real-life tasks. The differences in our measurements are likely due to variations in how the underlying system performs.

Tests with Applications. Our focus in this section was on maximizing and isolating small file accesses using programmer-oriented workloads. We also tested ERASER with more user-oriented tasks that are not primarily I/O bound, but that still heavily use the file system. Specifically, we experimented with two web browsers (Chromium, Firefox), and an office suite (LibreOffice), which create and delete large numbers of temporary files when running.

ERASER always incurred an overhead less than 5% compared to dm-crypt in these experiments, as expected. However, many test cases with normal use of these applications (e.g., automated web browsing with Chromium & Selenium) did not yield any reliably-measurable overhead. Therefore, we opt to present the benchmarks shown in Table 2 that specifically stress the bottleneck operations of ERASER as our primary evaluation effort for this work. Results of these benchmarks are useful and meaningful, as they depict worse-case performance characteristics of the system. The tests are also reliable and reproducible. This demonstrates that ERASER is practical for everyday tasks.

5.3. Discussion of Results

Our evaluation confirms that ERASER's performance is directly correlated with the number of files it accesses. I/O performed in big chunks, and on a small number of files, incurs no significant overhead. In contrast, accessing a new file, or deleting an existing one, triggers additional I/O operations to retrieve the corresponding keys from the FKT, or to rebuild branches of the FKT with fresh keys. Therefore, accessing large numbers of small files results in a noticeable loss of throughput compared to ordinary full disk encryption. However, in comparison to ordinary full disk encryption, ERASER guarantees secure data deletion and is useful in scenarios where privacy guarantees are of utmost importance.

Our evaluation also shows that this reduction in throughput does not always translate negatively to realistic workloads such as working with large directory trees; the performance loss is often measured in merely seconds. In fact, in many workloads, also including those that have processor-heavy components, ERASER matches dm-crypt in performance. We find these results encouraging, especially considering that dm-crypt is a standard, well-optimized component of the Linux kernel. We conclude that in most practical use cases, ERASER offers performance comparable to regular full disk encryption with the added benefit of secure deletion.

5.4. Overhead vs. Unencrypted Disk I/O

In our evaluation, we primarily focused on comparing ERASER's performance to regular full disk encryption with dm-crypt. The motivation for this decision was twofold. First, ERASER's observed performance loss is a direct result of disk encryption, and the overhead of secure deletion-specific operations are low in comparison. Second, ERASER provides security guarantees equivalent to modern full disk encryption technologies, making it a viable substitute in their place.

The negative impact of full disk encryption on I/O performance is a well-understood and accepted trade-off in the face of modern security threats. Therefore, we believe that presenting dm-crypt as a baseline for our evaluation is appropriate in many threat scenarios and computing environments.

Still, it is important to point out that, as shown in Tables 1 and 2, a vanilla system offers significantly higher I/O performance than both ERASER and dm-crypt. For instance, compared to an unencrypted disk, ERASER incurs an overhead of 50.82% for big I/O, and 117.10% for small file operations in the worst-case experiments. In a setting where users do not need the benefits of full disk encryption, and are not readily using a technology such as dm-crypt, this considerable performance hit could be an impediment to adopting ERASER.

5.5. Impact of File System on Performance

Our implementation's performance is not tied to the file system being used in combination with ERASER. This is a direct result of our implementation working at the block device layer. We verified this property by further testing ERASER on various other file systems supported on Linux, such as ReiserFS and VFAT. In each case, we obtained similar results to those we presented in this section.

Similarly, file system utilization has no effect on ERASER's performance. Most Linux file systems (e.g., Ext4) support a fixed number of inodes, configured when formatting the storage medium. As explained in Section 4.4, ERASER uses this information to initialize a static FKT, and changes to the tree structure will never be necessary.

With exceptional file systems that can dynamically adjust inode counts, we would need to reconstruct the FKT when the number of files exceed the keys initially supported; this would be a rare event with a one-time performance hit. However, we note that our prototype implementation of ERASER currently only supports a static FKT, and therefore, only file systems that use a fixed number of inodes.

6. Discussion & Future Work

Prior Tree-based Secure Deletion Work. As mentioned in Section 2, Reardon et al. [27] implemented a B-tree-based approach to secure file deletion that also made use of cryptographic erasure and key wrapping. This work is highly related; however, a significant difference between their prototype and ERASER lies in our focus on developing a high performance secure deletion technique, and subsequently, presenting a practical and usable system that can act as a viable substitute for existing, well-established full disk encryption tools.

First, while Reardon's B-tree prototype shows promising performance characteristics when combined with a suitable caching policy, our evaluation of ERASER shows that an FKT implementation can closely approach the performance characteristics of a heavily used and optimized production-level full disk encryption implementation (i.e., dm-crypt). We stress that we are not the first to propose tree-based cryptographic erasure using key wrapping. However, we believe that FKTs and our prototype implementation are the first to show that it can be performant for everyday use.

Next, Reardon's work leverages the Linux kernel's network block device facility [5], which routes block I/O requests over a TCP connection, and is typically used for accessing remote storage devices. The authors utilize this technique to present a proof-of-concept implementation of their approach for their experiments. In contrast, one of our primary goals when developing ERASER was to provide a robust, practical, and usable system that could easily be adopted for everyday use, on a typical Linux system. As a result, much of the novelty of our work lies in addressing this different set of design and implementation challenges.

Implementation Limitations. ERASER makes it possible to maintain a file-level secure deletion granularity while operating at the block device layer. However, this design poses a difficulty to deleting file metadata, as matching file system-specific metadata to inodes is a non-trivial (but not impossible) task. Our prototype does not perform secure deletion of metadata, and we leave tackling this implementation challenge to future work.

ERASER uses inodes to uniquely match encryption keys to files. This is an intuitive solution when dealing with Linux file systems, such as Ext4, which internally represent files using inodes. However, it should be noted that "foreign" file systems that are ported to work under Linux (e.g., FAT, ZFS) do not necessarily have the concept of an inode. Instead, they construct inodes in memory as files are accessed, and map their own internal representation of files onto these in-memory structures as this is required to interface with the VFS. This peculiar technical detail does not currently pose any difficulty to us. However, *in theory*, it could be possible to implement a file system that does not have a fixed inode number-to-file mapping, but rather assigns arbitrary inode numbers to files every time the file system is mounted. ERASER would not be compatible with such a file system, and addressing this limitation would require us to employ a different method to identify files on that file system.

Finally, we point out that secure deletion through cryptographic erasure is only as secure as the underlying cryptographic scheme. This is an inherent limitation of any security architecture that uses cryptography.

Swap & Hibernation Considerations. The secure deletion guarantees provided by our approach require that file keys are never written to physical storage without first being encrypted by a parent key. Likewise, the master key must never be persisted outside its designated secure vault. These conditions are easily be satisfied by keeping the keys in volatile memory protected by the kernel while in use. However, implementations should take the necessary precautions to prevent inadvertent leakage of keys in case the system goes into hibernation, or when memory pages are swapped out. Specifically, sensitive memory areas containing key caches should be marked non-swappable, and before entering hibernation, all key caches must be written back to persistent storage and their corresponding memory regions

sanitized. The device-mapper framework already provides a mechanism to intercept hibernation events, allowing easy implementation of this solution.

Unavailability of Master Key Vault. ERASER's frequently refreshes the master key stored in the external vault as part of its normal operation. However, should the vault become inaccessible for any reason (e.g., a removable storage device acting as the vault, such as a smartcard, could be unplugged), ERASER needs to take the appropriate actions to prevent inadvertent loss of data. One way to deal with such situations is to delay the master key rotation until the vault becomes available once again.

If ERASER is to be taken offline under these conditions, the direct children of an FKT could be encrypted with the old master key and persisted to disk, which would temporarily forego secure deletion. Later, when the vault becomes accessible, the master key would be rotated and all its direct children in the FKT immediately re-encrypted to securely erase all previously deleted files. Note that even in this scenario, an offline ERASER partition cannot be accessed again until the vault becomes available, because the master key is required to unlock the FKT on disk before the file system can be mounted.

Alternatively, in scenarios that involve highly-sensitive files, it could be preferable to rotate the master key as soon as files are deleted regardless of the vault's availability, and opt for having the file system become inaccessible should the system be taken offline before the new key could be written to the vault. Such a policy would instead sacrifice data integrity in favor of guaranteed secure deletion.

Data Integrity. We implemented ERASER in a stackable block device driver responsible for encrypting I/O blocks in flight and manipulating the FKT. As a result, ERASER does not have a direct impact on file system integrity, or hardware wear. File system integrity guarantees are provided by the concrete file system implementations at the layers above, and hardware-specific reliability concerns are handled by the underlying block device driver and any embedded storage controllers.

However, the integrity of the FKT, or more specifically, nodes containing file encryption keys directly affects the integrity of file data they encrypt; if a FKT node is not properly persisted to disk, the corresponding file will be lost. While similar risks are necessarily present in any disk encryption technique, the use of key caches requires special attention in ERASER's case, as an overly aggressive caching policy may delay FKT updates, and increase the risk of data loss. While various techniques employed by persistent storage technologies may be applied to secure FKT transactions and reduce this risk, potential solutions would need a careful security analysis, as replicating parts of FKT may invalidate ERASER's secure deletion guarantees.

Users' Perception of Secure Deletion. ERASER is designed to securely delete files only when a system call explicitly requests removal of the file inode in question. For instance, our prototype implementation considers the `unlink` and `truncate` family of system calls as the trigger for secure deletion. Of course, this could trivially be extended to cover other system calls.

However, file system implementations may not always explicitly destroy inodes even when, from a user's perspective, it may appear that file contents are deleted. For example, consider a scenario under Linux and Ext4 where a directory contains two files *X* and *Y*. When a user executes the command *"mv X Y"* to overwrite *Y*, the file system does not actually unlink *Y*. Instead, its inode is reused, and only the data blocks of *Y* are overwritten. In other words, ERASER would not consider this a file deletion event, and would not securely delete the contents of *Y* until the user later executes another command such as *"rm X"*, at which point all current *and* old data pointed by that inode is securely deleted.

One potential technical mitigation to this problem would be to deploy ERASER together with a simple user space toolkit that automatically corrects such unexpected behavior. For example, to address the case described above, a wrapper around "mv" can be introduced to explicitly call "rm" on overwritten files. However, in general, users of ERASER need to be aware of this semantic gap and limitation of the system, and explicitly execute deletion operations when secure deletion is desired. This demonstrates that technical solutions often do not completely obviate the need for effective security awareness, training, and practice.

Application-Level Secure Deletion. This paper focused on achieving secure deletion at the file system level, as opposed to finer-grained secure deletion of specific pieces of sensitive application data. Our approach balances this loss of granularity with strong security guarantees and good usability in everyday scenarios with a wide array of applications and operating systems. However, we note that file-level secure deletion is not necessarily the optimal granularity in all cases. For instance, Android applications may store and manipulate all of their data in a single, file-backed database; in such cases, encryption and secure deletion at database entry-granularity may be the more appropriate approach [28]. We leave exploration of this venue to future work.

7. Conclusions

Even though the problem of irrevocably deleting data from non-volatile storage has been explored by many researchers, flash-based storage media with opaque on-board controllers still make it a challenging task to provide strong secure deletion guarantees on modern computers. At the same time, previously practical secure deletion tools and techniques are rapidly becoming obsolete, and are rendered ineffective.

We leverage cryptographic erasure to design a novel, effective secure deletion technique called ERASER. Our work is distinct from the myriad of existing literature in this field in that, ERASER can guarantee secure deletion of files on storage media regardless of the underlying hardware's characteristics, treating storage devices as blackboxes. We achieve this by bootstrapping cryptographic erasure with the

help of an external, secure storage vault, which could be implemented using cheap, commodity hardware.

We present a practical implementation of ERASER, realized as a stand-alone Linux block device driver that could be deployed and used on a standard computer with a TPM chip. We demonstrate that our implementation exhibits similar performance characteristics to dm-crypt, the standard disk encryption module on Linux, and thus offers users an alternative full disk encryption solution with the added benefit of secure file deletion.

Acknowledgments

We would like to thank our shepherd Kaveh Razavi and the anonymous reviewers for their precious time and helpful comments. This work was supported by the National Science Foundation (NSF) under grant CNS-1409738, and Secure Business Austria.

References

[1] "Bonnie++," http://www.coker.com.au/bonnie++/.

[2] "Device-mapper – Linux Kernel Documentation," https://www.kernel.org/doc/Documentation/device-mapper/.

[3] "eCryptfs," https://launchpad.net/ecryptfs.

[4] "Kernel Probes (Kprobes) – Linux Kernel Documentation," https://www.kernel.org/doc/Documentation/kprobes.txt.

[5] "Network Block Device (TCP version) – Linux Kernel Documentation," https://www.kernel.org/doc/Documentation/blockdev/nbd.txt.

[6] Apple, Inc., "Mac OS X: About Disk Utility's erase free space feature," https://support.apple.com/kb/HT3680, 2016.

[7] S. Bauer and N. B. Priyantha, "Secure Data Deletion for Linux File Systems," in *USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2001.

[8] D. Boneh and R. J. Lipton, "A Revocable Backup System," in *USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 1996.

[9] S. Diesburg, C. Meyers, M. Stanovich, M. Mitchell, J. Marshall, J. Gould, A.-I. A. Wang, and G. Kuenning, "TrueErase: Per-file Secure Deletion for the Storage Data Path," in *Annual Computer Security Applications Conference*. New York, NY, USA: ACM, 2012.

[10] S. Diesburg, C. Meyers, M. Stanovich, A.-I. A. Wang, and G. Kuenning, "TrueErase: Leveraging an Auxiliary Data Path for Per-File Secure Deletion," *ACM Transactions on Storage*, vol. 12, no. 4, pp. 18:1–18:37, May 2016.

[11] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel, "Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels," in *USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012.

[12] B. Durak, "Wipe," https://github.com/berke/wipe.

[13] C. Fruhwirth, "New Methods in Hard Disk Encryption," http://clemens.endorphin.org/cryptography, 2005.

[14] J. Garlick, "diskscrub," https://code.google.com/archive/p/diskscrub/, 2008.

[15] P. Gutmann, "Secure Deletion of Data from Magnetic and Solid-State Memory," in *USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 1996.

[16] G. Hughes, T. Coughlin, and D. Commins, "Disposal of Disk and Tape Data by Secure Sanitization," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2009.

[17] S. Jia, L. Xia, B. Chen, and P. Liu, "NFPS: Adding Undetectable Secure Deletion to Flash Translation Layer," in *Asia Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2016.

[18] N. Joukov and E. Zadok, "Adding Secure Deletion to Your Favorite File System," in *IEEE Security in Storage Workshop*, 2005.

[19] N. Joukov, H. Papaxenopoulos, and E. Zadok, "Secure Deletion Myths, Issues, and Solutions," in *ACM Workshop on Storage Security and Survivability*. New York, NY, USA: ACM, 2006.

[20] M. Larabel, "Phoronix – The Performance Impact Of Linux Disk Encryption On Ubuntu 14.04 LTS," http://www.phoronix.com/scan.php?page=article&item=ubuntu_1404_encryption.

[21] J. Lee, S. Yi, J. Heo, H. Park, S. Y. Shin, and Y. Cho, "An Efficient Secure Deletion Scheme for Flash File Systems," *Journal of Information Science and Engineering*, 2010.

[22] Microsoft – Windows IT Center, "Trusted Platform Module Technology Overview," https://technet.microsoft.com/itpro/windows/keep-secure/trusted-platform-module-overview.

[23] ——, "What's new in Windows 10, versions 1507 and 1511," https://technet.microsoft.com/itpro/windows/whats-new/whats-new-windows-10-version-1507-and-1511.

[24] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda, "PrivExec: Private Execution as an Operating System Service," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2013.

[25] J. Reardon, D. Basin, and S. Capkun, "SoK: Secure Data Deletion," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.

[26] J. Reardon, S. Capkun, and D. Basin, "Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory," in *USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2012.

[27] J. Reardon, H. Ritzdorf, D. Basin, and S. Capkun, "Secure Data Deletion from Persistent Media," in *ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2013.

[28] R. Spahn, J. Bell, M. Lee, S. Bhamidipati, R. Geambasu, and G. Kaiser, "Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems," in *USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2014.

[29] S. Swanson and M. Wei, "SAFE: Fast, Verifiable Sanitization for SSDs," University of California, San Diego, Tech. Rep., 2010.

[30] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "CleanOS: Limiting Mobile Data Exposure with Idle Eviction," in *USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012.

[31] M. Wei, L. Grupp, F. Sapad, and S. Swanson, "Reliably Erasing Data from Flash-Based Solid State Drives," in *USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2011.