Secure File System Versioning at the Block Level

by

Jacob Taylor Wires

B.Sc., The University of California at Santa Barbara, 2004

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

September, 2006

Abstract

Information is capital; disk space is a mere commodity. Versioning file systems offer an appealing storage model that prevents users from unintentionally deleting or overwriting important data by transparently retaining old versions. However, improving storage reliability by adding versioning to a file system is problematic in two important ways. First, the complexity of file systems and the operating systems in which they reside leaves data vulnerable to bugs and viruses, even when versioning is added. Second, the mission-critical nature of file systems makes users and OS vendors justifiably hesitant to adopt new file system features like versioning, regardless of the potential benefits they might provide.

This thesis presents VDisk, a block layer system capable of providing file-grain versioning to existing, unmodified file systems. VDisk features a novel division of labor to enhance security and reliability. Write-access to versioned data is restricted to two very simple, reliable, file system agnostic components: a block logger and a log cleaner. These crucial components are isolated in a virtual machine, where they are protected from the errors and attacks that plague operating systems. More complicated, untrusted, read-only utilities operate in user space. These utilities, which are free to use sophisticated, off-the-shelf tools not appropriate for trusted kernels, support version browsing and reconstruction without degrading system reliability.

VDisk employs a policy-driven approach to block reclamation. A retention policy specifies a set of constraints that describe which file versions must be retained and which need not be. A user-space tool periodically invokes the secure cleaner by submitting a set of delete requests along with a proof that these requests satisfy the retention policy. The secure cleaner verifies the proof and reclaims the specified blocks if applicable. Experimental results show that the cleaner is capable of reclaiming more than 80% of logged data.

Table of Contents

List of Tables

List of Figures

Acknowledgements

Many thanks to Mike Feeley and Norm Hutchinson. Thanks also to Abhishek Gupta, Kan Cai, Cuong Le, Andre Lifchits, and the DSG lab.

Chapter 1

Introduction

1.1 Motivation

Computers have become repositories for a variety of information, including business records, intellectual property, and sentimental keepsakes. Some of this information has direct financial value; some of it is invaluable; all of it is entrusted to computers, which must store it securely and dependably. Computers are quickly becoming an integral component of every aspect of our lives, but they are only truly useful insofar as we can trust them to operate correctly. Storage subsystems, which are responsible for ensuring data persistence, have thus become one of the most crucial components of computing environments.

Advances in hardware technology have resulted in storage systems with capacities commensurate to the value of the data they are meant to retain. Hard drives have become so affordable and so capacious that in many cases they can take the place of slower, more cumbersome tertiary storage devices such as magnetic tape drives. This surplus of disk space has introduced new opportunities for improving storage reliability—and increased the complexities of doing so: modern storage systems must now be capable of scaling to immense proportions and supporting large numbers of users.

The complexities involved in providing reliable storage on a vast scale have become onerous to system administrators. While the cost of hard drives has decreased significantly in the past years, the cost of storage administration has not. In fact, the cost of administration has been estimated to exceed the price of storage hardware by several hundred percent [19]. Increased system administration is undesirable not only because it increases costs, but also because it increases the potential for human error, thereby reducing system reliability.

Modern hardware is no longer the primary cause of data loss; recent studies impute 60% to 80% of data loss to human error, software defects, virus attacks, power failures and site failures [63]. The users themselves can constitute a surprisingly hazardous threat to data: in addition to erroneously deleting important files, users may intentionally modify a file one day only to

find themselves desperate for the original version a week later. A dependable storage system charged with the task of protecting data from all dangers is thus charged with the task of protecting users from themselves.

Data is valuable, disk space is cheap, administrative complexities are burdensome, and users are fallible. These trends suggest the need for a new model of data storage; a natural design response is to provide liberal undo capabilities by transparently increasing data redundancy at the expense of increased storage requirements.

1.2 Data Versioning

Versioning utilities use the abundance of storage capacity available to modern machines to preserve multiple versions of important data. Versioning is a general service that can be provided by a number of entities, including user-space applications, virtual and on-disk file systems, and block-level drivers. To be effective, a versioning utility should operate transparently and automatically, thereby reducing the potential for human error. To be dependable, a versioning utility should be exceptionally simple.

A number of versioning file systems [13, 23, 34, 39, 47, 53] have been created to provide users with a new model for data management. Versioning file systems maintain multiple versions of files: the current version is functionally identical to conventional files, while older versions are immutable and often hidden during normal operation. Users can browse a file's history and revert to an older copy at any time. Many versioning file systems also provide extra features, such as the ability to stipulate per-file or per-file-group retention policies [47] and the ability to transparently compress old versions [34].

Versioning file systems provide rich functionality, but the intrinsic complexity of these systems significantly compromises their dependability. File systems export sophisticated interfaces and must support parallel, asynchronous interactions with both application and block layer code. The file system constitutes a fundamental component of most operating systems; its performance must be fine-tuned, well coordinated, and utterly reliable to ensure the integrity of the system. Moreover, it is exceptionally difficult, if not impossible, to verify the correctness of file system code with formal auditing. Even ext3 [1], a mature file system renowned for its reliability, is not flawless: model checking has been used to uncover five correctness errors in this file system, including one that resulted in permanent data loss [62]; it is likely that callow versioning file systems would require years of wide-spread

use and refinement to achieve even this imperfect level of dependability. For these reasons, operating system vendors and users are typically very reluctant to adopt new file systems, even when they offer innovative and enticing features.

User-level applications like CVS [18] and PRCS [30] and stackable file systems such as VersionFS [34] and Wayback [13] add a degree of flexibility not attainable by on-disk file systems, as these higher-level systems can be used to provide versioning to extant file systems. However, versioning applications fail to provide the transparency required of a truly dependable system because they leave the user responsible for initiating data preservation. Moreover, both applications and stackable file systems rely upon underlying file systems to maintain important versioning information—and thus can be no more secure than the on-disk systems over which they operate.

1.3 Secure Logging

File system undo capabilities protect data by interposing an extra level of indirection between users and data, much in the same manner that the recycle bin metaphor protects against undesired deletions by requiring an additional confirmation of a user's intent. In a trusted environment, this extra layer of insulation is often enough to prevent the destruction of important data. However, in a less sheltered environment, all the effort invested in retaining versioned data might go for naught if the data is not carefully secured.

If a versioning system supports the removal of versioned data—and for the sake of practicability, it must somehow do so—it runs the risk of succumbing to the very dangers it seeks to protect against. Although users would conceivably be more circumspect when deleting a file's version history than they would when performing common file system operations, the presence of a mechanism which is capable of destroying version histories poses the same dangers to versioned data that conventional file system operations pose to unversioned data: in both cases, there exists the same potential for the erroneous loss of data, and, perhaps more importantly, there exists the same susceptibility to the malicious destruction of data.

Version histories can be valuable for a number of reasons. In addition to protecting users from their own mistakes, version histories can be used to audit compromised systems. Version histories can record the actions of viruses and other malware, and they can be used both to discover the system vulnerabilities exploited by such agents as well as to recover from

the harmful consequences engendered thereby—but only if they survive the attack. If version histories are not protected, viruses could cover their trails by destroying versioned data, rendering log-based recovery impossible.

To guard against both incidental and malicious data destruction, versioned data must be protected. In particular, untrusted processes should never be capable of overwriting or deleting versioned data. This implies that only privileged, dependable processes should have write-access to versioned data. There are number of methods by which versioning can be implemented, but given the complexity and vulnerability of large code-bases, the most secure way to implement versioning is to do so at the lowest possible level.

A dependable system is one in which all functioning components are trusted to adhere to their intended purposes. A vulnerability in any one component may be propagated throughout the entire system, undermining the system's dependability. To reduce or eliminate the vulnerabilities of a system, it is beneficial to minimize the trusted computing base (TCB) of the system, thereby minimizing the probability that the system is reliant upon a faulty component.

The process of data versioning admits of a natural division into two tasks: *version preservation* and *version browsing*. In an ideal world, users would never be compelled to delete data, but even the immense storage capacity provided by modern hard disks is not infinite; thus, a third task must be supported by any practical versioning system: *version pruning*.

File systems perform all of these tasks as a single, monolithic unit. The complexity of these systems derogates from the appeal of including them in a trusted computing base. If versioning is to be implemented in a dependable manner, it should be done below the file system, at the block layer. The block layer interface is very narrow: block-level drivers deal only with raw blocks of data and remain blissfully ignorant of complicated file system semantics. This tractable interface enables the development of block-level services that are much simpler than their bulky file system counterparts and therefore much easier to audit for correctness.

The TCB of a file system includes the file system itself, the kernel, the device drivers, and the device hardware; a failure in any one of these components could result in the irretrievable loss of data. Block-level drivers are situated much closer to hardware and thus suffer from far fewer vulnerabilities than file systems. Moreover, the situation of drivers beneath the kernel makes it possible to completely isolate these extensions from the operating system. With the use of virtual machine monitors (VMMs) such as Xen [7], block-level drivers can be hermetically isolated, protected by the VMM from

the vagaries of kernel operations.

However, new difficulties arise when endeavoring to offer satisfactory functionality from the block layer. Conventional block layer versioning systems like Clotho [19] and Peabody [32] can only provide coarse-grain versioning of logical volumes. These systems are compatible with a number of existing file systems, and their trusted computing bases are significantly smaller than those of versioning file systems, but their limited version browsing and version pruning facilities yield sub-par utility. Users predominantly interact with storage systems at the granularity of files, and thus a truly effective versioning system should support versioning on a per-file basis.

The challenge, then, to providing a dependable versioning system is to achieve file versioning at the block layer. Moreover, the extra complexity required to support file versioning should not detract from the system's reliability.

With these considerations in mind, we have designed and implemented the Versioning Disk (VDisk), a dependable versioning system. The most notable characteristic of VDisk is its stratified design: to improve dependability, we have incorporated only the simplest mechanisms into the critical components of VDisk; complicated—and possibly undependable—tasks are executed in user space, where they can cause no harm.

VDisk executes as a block-level driver, and is therefore compatible with multiple file systems. The critical components of VDisk include a simple logging utility and a secure log cleaner. File system writes are routed through the VDisk driver, which simply appends these writes to an immutable log before passing the requests down to the underlying device driver. While this approach requires the duplication of all disk writes, it has the advantage of leaving the original on-disk file system layout unmodified, meaning that VDisk does not introduce new vulnerabilities into existing file systems.

To support version browsing at a per-file granularity, an untrusted user-space application uses the logged data to reconstruct file versions. There are numerous advantages to performing reconstruction in user space. For one, user-space development is much simpler than kernel development. Powerful debuggers and a protected execution environment greatly ease the coding process. As well, user space applications can leverage a plethora of tools not available to kernel modules. For example, we use a relational database to organize log metadata, which greatly simplifies file reconstruction. Finally, because the reconstruction process never modifies the log, it is incapable of erroneously or maliciously destroying logged data, and thus poses no risks to the system's dependability.

A concern for all versioning systems is resource management. Even with

cheap, abundant storage, disk space will ultimately need to be reclaimed. This imposes new dependability issues, as any process which modifies versioned data introduces potential dangers. However, following our stratified design approach, we have faced this challenge by dividing version pruning into two sub-tasks: an untrusted and innocuous user-space application makes suggestions about what data is eligible for deletion, while a simple, secure cleaner verifies that these suggestions adhere to the system's retention policies and performs the deletions. In this manner the advantages of user-space development can be exploited without introducing new vulnerabilities into the trusted logging subsystem.

Chapter 2

Related Work

The tremendous value of data stored on modern computers has motivated the creation of a number of tools designed to preserve electronic documents. User-initiated applications such as CVS [18], RCS [56], and PRCS [30] facilitate the maintenance and organization of file versions by allowing users to commit important versions of selected files to a repository. These applications are particularly useful in well-managed, dynamic environments where users a make conscious effort to retain important file versions, but they fail to provide the transparency and security required of a truly dependable versioning system. Snapshotting, checkpointing, and transparent versioning at or below the file system layer preserve data versions automatically and thus reduce the potential for human error. In addition to providing increased protection of important user documents, comprehensive versioning systems are useful in other domains, such as post-intrusion analysis [14, 60] and kernel debugging [28].

However, the increased functionality obtained by implementing versioning in the kernel does incur a cost. Adding code to file systems and kernels increases the difficulties of technology adoption and introduces potential new security holes. Moreover, the typically vast size and complexity of these systems makes any formal verification of their correctness extremely difficult, thereby diminishing their trustworthiness. These predicaments have prompted the development of new technologies—such as gray-box design and trusted computing infrastructures—that can be used to mitigate the difficulties of kernel expansion.

2.1 Snapshot Utilities

Snapshot utilities support the coarse-grain versioning of data. Such utilities enable the periodic creation of file system images, allowing for the production of a series of instances from a file system's history which can be accessed online or archived in tertiary storage. These utilities do not support the versioning of individual files and do not maintain comprehensive histories of file system activities: each snapshot preserves the entire file system state

at a single instant, but any data updates made between two snapshots are irrecoverable.

The Write Anywhere File Layout system (WAFL) [24] was designed to operate as a dedicated network storage appliance interfaced via NFS [46]. WAFL's hierarchical block layout enables the rapid generation of file system snapshots: given a live block tree, a read-only snapshot can be made simply by copying the root node of the tree. Subsequent writes to the live tree are done in a copy-on-write fashion so that the data blocks referenced by the snapshot's tree are not overwritten. Modified blocks are written to new locations on the disk, and all intermediary nodes linking the leaf to the root must be updated to reflect these relocations; these updates are buffered with non-volatile RAM and batched to improve efficiency. WAFL's block reallocation scheme sets a hard limit on the number of concurrent snapshots the system can support. Each block is tracked with a 32-bit reference counter, with each bit indicating the block's allocation status in exactly one snapshot. Thus the system can only support 32 snapshots at any given time, a limitation which makes WAFL ill-suited for a number of versioning scenarios, including landmark preservation [47] and post-intrusion analysis.

WAFL achieves efficient snapshotting because of its unconventional, hierarchical block store. Traditional UNIX file systems are intellectual descendants of the Berkeley Fast File System [31] and as such share a common disk layout which is significantly different than that of WAFL. FFS was designed to minimize disk seeks, the slowest of disk operations. Adhering to the assumption that a directory and the files it contains will often be accessed concurrently, FFS endeavors to place the data blocks which compose these objects together on the same disk cylinder, hoping thereby to curtail disk seeks. The Sprite Log-structured File System (LFS) [44] takes a radically different approach to disk layout. Positing that the majority of file system reads can be served by ever-growing in-memory file caches, the designers of LFS made optimizing disk writes their main priority. In LFS, disk writes are organized sequentially and sent to the disk in batches: many individual file system updates are clustered into a single large write, which is appended to a contiguous region of the disk. This process ensures that collections of multiple small writes, which may have required many disk seeks in FFS, will be written to one region by LFS and will thus require far fewer seeks.

The consequences of this design are manifold. Most notably, data is not directly overwritten—once committed to disk, data blocks remain immutable until they are ultimately reallocated by a system cleaner. LFS can thus be considered an implicit versioning system, and indeed, LFS makes

good, if limited, use of its versioning capabilities. LFS supports the notion of checkpoints, or positions in the log which contain consistent instants of the system's data and metadata. If a machine shuts down while the on-disk representation of LFS is inconsistent, it is possible to revert to a recent consistent state simply by returning to the last checkpoint.

Although LFS contains the information required to support a more comprehensive versioning policy, it only preserves two checkpoints. Blocks not belonging to these checkpoints must be reallocated to ensure the availability of contiguous extents for efficient writes; this process of reallocation entails new difficulties. In particular, much effort must be expended by the LFS cleaner to determine which segments of the disk contain blocks that are no longer referenced by current files. This cleaner, which operates as a separate background process, transfers the live data from multiple segments into a few segments; the remaining empty segments are then available for reallocation. However, this cleaner is the source of some controversy, as it has been found to decrease overall file system performance by more than 34% in transactional environments, with the cleaner accounting for up to 80% of data written to disk [48, 49]. The advantages and new capabilities provided by LFS thus come with a concomitant increase in complexity.

A variety of systems provide snapshotting functionality similar to that of WAFL and LFS [11, 25, 29, 33, 38, 41, 59]. While the individual designs of these systems vary widely, they all share the common goal of providing coarse-grain, system-wide versioning. This type of versioning can be useful, but it cannot provide sufficient guarantees against inadvertent data loss. To provide absolute protection of important data from malicious and unintended deletion, a more exhaustive approach is required.

2.2 Versioning File Systems

Versioning file systems employ a copy-on-write strategy to create versions on a per-file, rather than a per-system, basis, thereby creating much more detailed histories of file system activities. Resource management is a key challenge faced by versioning file systems—even on modern, capacious disks, the retention of every version of every file will ultimately result in a complete exhaustion of storage space. A number of versioning file systems have been designed and implemented, each offering its own unique advantages and drawbacks.

The Cedar File System [23] was one of the first file systems to automatically preserve immutable versions of files. Cedar is a distributed file system

designed to support file sharing between multiple concurrent users. Files in Cedar are designated local or remote; users operate directly on local files, and can choose to share them by copying them to a remote file server. All remote files are retained automatically, but local versions are silently pruned according to a simple per-file policy.

Elephant [47], which operates beneath FreeBSD's Virtual File System layer, creates a new file version upon the first write to an opened file. All subsequent writes before the file is closed are performed on the new version. Elephant supports the application of sophisticated versioning policies to make the best use of storage space. These policies, which include Keep One, Keep All, Keep Safe, and Keep Landmarks, can be applied on a per-file or per-file-group basis. Keep One retention only preserves the current version of a file, while Keep All retention preserves all versions of a file. Keep Landmarks employs a heuristic to retain important milestone versions of a file while reclaiming all others. The heuristic is based on the assumption that minor differences between infrequently accessed versions lose meaning to the user as the files age. Thus for very old versions, only relatively stable instances—i.e., those that come at the end of a spurt of revisions and remain unchanged for a substantial period of time—should be retained, while for recently accessed files, all versions should be retained. The Elephant cleaner cleans a file by examining the log of inodes which represents its version history and reclaiming eligible versions. Files are selected for cleaning on the basis of a heuristic value which is updated on every file close. File cleaning in Elephant is significantly easier than segment cleaning in LFS, because the Elephant cleaner need only read file metadata to perform its duty, while the LFS cleaner must read entire segments and rewrite live blocks.

Elephant was built from scratch to provide versioning from within the file system; the adoption of Elephant thus requires the replacement of existing, tried and true file systems with a new, untested system. In contrast, VersionFS [34] is implemented as a stackable file system and operates within the VFS layer, making it compatible with all standard on-disk file systems. VersionFS adopts a copy-on-change policy to reduce the amount of redundant data stored in file versions and supports the transparent compression of versioned files.

In a similar vein, Wayback [13] logs all version changes in user space, relying on a kernel module to intercept file system calls and trap to the application-layer server. Both VersionFS and Wayback operate above on-disk file systems, and as such can be used to incorporate versioning functionality into any existing file system. However, the inability of these systems to directly control on-disk data structures poses new problems, as they are in-

capable of preventing stale file versions from polluting the kernel page cache; moreover, VersionFS cannot guarantee that a file's inode number will remain constant throughout its lifetime, a property required for compatibility with NFS.

Ext3cow [39, 40] is a modification of ext3 that makes use of retrofitted on-disk ext3 inodes to accommodate the additional metadata required for maintaining versions, thereby reducing the maximum supportable file size by 16%. Ext3cow can provide transparent snapshot capabilities as well as individual file versioning. Unlike Elephant, ext3cow is an extension of a popular, robust file system. This approach both simplified the implementation of ext3cow and reduced the amount of new code required to achieve versioning. However, ext3cow suffers from the same compatibility issues as Elephant—neither of these on-disk file systems can provide general versioning services to arbitrary file systems.

The Comprehensive Versioning File System [53] was designed with security as a top priority. CVFS operates below S4 [54], a self-securing, dedicated storage server, and is accessed via NFS [46]. To facilitate post-intrusion analysis and auditing, CVFS maintains copies of all data and metadata written to disk for a predetermined period of time, known as the detection window. This system allows administrators to investigate the propagation of malicious data and thereby determine the source of a system's ailments. Because CVFS maintains versions of every write to disk, it must take special care to optimize the storage of versioned data. In particular, CVFS introduces metadata journaling and multiversion b-trees to reduce metadata storage requirements, which could otherwise equal the storage requirements of the data itself.

2.3 Block Logs

While it is intuitive to offer file versioning at the file system level, there are advantages to be gained in providing data versioning from beneath file systems. A mechanism that can export versioning capabilities to any number of file systems can provide a degree of flexibility and simplicity not achievable by monolithic file systems. One means of providing flexibility is the use of stackable or user-space versioning systems like VersionFS and Wayback. A simpler approach is to offer versioning at the block level.

Clotho [19], which was designed to exploit the increasing computing power of storage systems, supports versioning of volumes. Clotho is implemented as an addition to the block layer, and as such can operate beneath

any file system. Clotho organizes data into extents, introducing a level of indirection between logical and physical block addresses. Like WAFL, versioning in Clotho is achieved by writing versions of data blocks to new disk locations and updating the mapping to reflect these relocations. Clotho is a snapshot system, and can be configured to create read-only snapshots of an entire volume upon the execution of any number of file system operations, including file writes. Snapshots are accessed through virtual devices: each snapshot results in the creation of a new, read-only virtual device, which can be mounted and browsed in the conventional manner.

Peabody [32] is a block-level logging utility designed to operate on network-attached storage systems. Peabody preserves data by appending all writes to an on-disk log in a manner similar to that of LFS. For any file system with a consistency checker, Peabody can provide undo capabilities by simply rolling the log back to the desired time and using file system tools to verify system consistency; if the target time cannot be made consistent, the log can be rolled forward or backward until consistency is achieved. Peabody employs a number of optimizations, such as content hashing and silent write prevention, to avoid redundant writes to the log, thereby reducing the size of the log.

While Peabody relies upon content hashing to limit log size explosion, a different approach has been employed with modified RAID arrays. TRAP [63], or Timely Recovery to Any Point-in-time, systems store block histories by retaining exclusive-ORs of consecutive block versions. Due to a strong content locality, these exclusive-OR delta chains can represent version histories with extreme concision. Workloads studied in [63] exhibited only 5% to 20% bit changes between consecutive block versions. This small variance redounds to exclusive-ORs consisting primarily of zeros, which can be greatly compacted through simple run-length encoding. Version reconstruction is achieved by traversing these delta chains, applying the exclusive-OR of each version against the next successive version, until the desired state is reached.

All of these block-level loggers are oblivious to file system semantics, making them compatible with a number of different file systems. This nescience of file system semantics greatly simplifies the task of data preservation. However, none of these block-level utilities support versioning at a per-file granularity; thus the flexibility and simplicity achieved by these systems comes with restricted functionality.

2.4 Gray-Box Systems

A significant problem faced by systems researchers is the difficulty of convincing users to adopt new technologies. Many innovative ideas have been sent quietly to their graves because of the difficulty of incorporating them into widely used systems. Gray-box technologies [5] have been designed to allow researchers to implement new ideas on commodity systems without modifying kernel code. This technique both improves the adoptability of new technologies and avoids the introduction of new vulnerabilities into heavily distributed kernels. The knowledge which can help to turn black-box systems into gray-box systems is obtained from three main sources: an *a priori* understanding of the system's algorithms, controlled observation of the system's performance, and inferences of the system's internal state.

Semantically Smart Disks [52] present a prime example of the power of gray-box techniques. Typical operating systems provide a very narrow interface between file systems and block-level I/O. This interface is useful in that it greatly simplifies block-level drivers, which need only deal with blocks of raw data. However, it also limits the functionality that typical block-level drivers can provide, because it hides all information of higher level abstractions, such as files and directories, from the drivers. Gray-box techniques can be used to reacquire this knowledge at the block layer. When a semantically smart disk is installed, a five phase process is executed to discern the location of important file system structures on disk. An application-layer process probes the gray-box system by executing a number of carefully planned file system functions while a block-level module observes the read and write requests which are consequently sent to the disk. By combining these observations with a deep understanding of supported file system algorithms, the module can infer the type of the file system which is using the disk, and can thus perform file system-aware optimizations at the block level without modifying the kernel's block-level interface.

A convincing use of file system knowledge at the block layer can be found in D-GRAID [50], a driver which exploits knowledge provided by gray-box techniques to optimally organize file system objects for use with RAID [37] systems. RAID systems stripe data across multiple disks to improve performance. Data is often replicated on multiple disks to reduce the potential for data loss upon disk failure, but because RAID controllers operate beneath the block-layer interface, they are typically unable to stripe and replicate data intelligently. For example, if a file composed of multiple blocks is striped across multiple disks, a failure of any one of these disks (assuming no replication) will render the file unusable, even though the

majority of its data is still accessible. D-GRAID uses its knowledge of file system semantics to ensure that all data blocks associated with a file are stored on the same disk; thus the failure of a disk will only result in the loss of files which are entirely stored on that disk, rather than resulting in the loss of any file which has even a single block stored on the disk. This system is a pointed example of the enhancements achievable through a multi-level implementation of block layer mechanisms.

2.5 System Logs

Versioning file systems provide typical users with an obvious benefit: the ability to undo any file system operation. Data versioning can also enhance system security by enabling post-intrusion analysis and recovery.

Chronus [60], which operates within a μDenali virtual machine [61], records all changes made to the disks of its descendant VMs in an append-only log. If a child VM begins malfunctioning, an analysis of the log can expose the cause of the error and enable a reversion to the last functioning disk state, effectively undoing the problem. Chronus uses a binary search trial and error method to automate this analysis, rebooting its child VM with different versions of the disk until the problematic change is found.

ReVirt [14], implemented as a part of UMLinux [9], logs non-deterministic events of guest operating systems. This log can then be used to replay the guest operating system's activities instruction by instruction. The log only maintains a history of events that cannot be reproduced during the re-enactment process. Similarly, BackTracker [27] logs higher-level operating system events and objects. This log can be used to generate an easy-to-read flow-chart which graphs the actions of intruders and the objects they affected, helping administrators determine what system vulnerabilities were exploited during an attack.

Operator Undo [8] uses time-travelling disks to provide system-wide undo functionality to administrators. Designed as a general infrastructure for supporting application-neutral undos, Operator Undo provides an interface for recording system actions, which are described as an abstract verb data structure. Front-end proxies are charged with marshalling application-specific data into verbs; Operator Undo records these verbs and can use them to rewind to previous system states *and* to replay system activities, incorporating repairs made along the way.

2.6 Trusted Computing

While gray-box technologies attempt to extend kernel functionality without introducing new kernel code, their applications are limited; some features simply cannot be implemented without modifying the kernel. But while kernel changes may, in some cases, be inevitable, the impact of these changes can often be mitigated. One strategy for facilitating kernel modification is to isolate the existing, robust, trusted code from the new, experimental, potentially incorrect extensions.

Terra [21] is a framework intended to facilitate trusted computing on commodity hardware. Citing the complexity of full-featured operating systems as an inherent limitation of assurance, Terra's designers strove to minimize the trusted computing base of sensitive applications by simulating closed platform functionality with virtual machines. Terra isolates critical applications within specialized, trusted domains, which can employ user-specified customized operating systems to provide optimal security. These trusted domains are protected from malicious tampering by a Trusted Virtual Machine Manager. In addition, Terra provides an attestation interface which enables the verification of critical applications through the use of certificates documenting the validity of applications, drivers, firmware and hardware.

Nooks [55] aims to transparently isolate kernel code from modules and drivers, which are responsible for a significant proportion of crashes in commodity operating systems [10, 35]. Although this isolation is sought to protect the kernel from extensions rather than vice versa, the principle behind Nooks is similar to Terra and other trusted computing frameworks. The intention of these systems is to mitigate the danger of executing untrusted code by establishing controlled operating environments. Nooks achieves this by interposing a management layer between the kernel and its extensions; this layer ensures that untrusted extensions operate within protected domains. Any undesired operations attempted by extensions can be prevented or reversed by the management layer, thus shielding the kernel from erroneous extensions. Nooks is based on the observation that in commodity systems, the majority of bugs are introduced by drivers, and thus the kernel should be protected from extensions, while Terra endeavors to protect critical, thoroughly audited applications from large, complicated operating systems.

The primary intention of the Exokernel [17] is to enable application-level management of operating system resources. Advocates of end-to-end design [45] maintain that although a single, monolithic operating system

can provide satisfactory functionality to a broad range of applications, its performance for any particular application is often hindered by unnecessary generality. The Exokernel therefore endeavors to grant applications as much leeway as possible in designing and managing system abstractions such as inter-process communication and virtual memory. Library operating systems run in the address space of the applications they support, and can be optimized for individual performance requirements. A low-level exokernel controls resource allocation and revocation, maintaining equity between competing library operating systems. Library operating systems can create rich system abstractions to facilitate application development and execution; the exokernel, however, remains ignorant of the semantics of such abstractions. The exokernel enforces resource protection by evaluating simple predicates, provided by library operating systems, which express resource requirements in a language that the low-level exokernel can understand. This infrastructure allows the exokernel to protect system resources without understanding their application-level semantics. The exokernel thus isolates the complexity of richly featured operating systems from the mechanism responsible for protecting system resources, greatly reducing the trusted computing base of the system's critical components.

While the Exokernel supports customized library operating systems, it is incompatible with unmodified commodity operating systems. In contrast, Xen [7] has sought to provide some of the same features as the Exokernel while supporting standard operating systems such as Linux and Microsoft XP. Xen, a virtual machine monitor, is designed specifically for the x86 architecture. Xen *paravirtualizes* hardware by providing very slightly modified interfaces to guest operating systems. The Xen hypervisor can support multiple guest operating systems simultaneously; each guest OS operates in its own isolated domain, and is protected by the hypervisor from all other guest operating systems. Additionally, Xen supports the isolation of individual device drivers; any driver can operate in complete isolation by running in its own virtual machine [20]. Guest operating systems communicate with these isolated drivers via an asynchronous *device channel* primitive which is offered by the hypervisor. This isolation enables Xen to protect guest operating systems and device drivers from each other; as well, the device channel which links these two entities can be redirected to user space processes in the driver's domain, thus facilitating the implementation of virtual devices from the application layer [58].

2.7 Summary

The increasing value of digital information has motivated the design of a number of systems intended to preserve on-disk data. Many file systems provide snapshotting functionality, allowing users to revert to previous system images to recover old data. More comprehensive systems decrease version granularity by retaining all versions of all files, or by enforcing user-specified policies to retain important files. Versioning can also be provided by block layer systems, which are generally oblivious to file system semantics and can thus provide volume or virtual disk versions to a number of unmodified file systems. Gray-box technologies have explored methods for providing some file system information to the block layer without modifying existing kernel code.

In addition to providing users with file system undo capabilities, versioning file systems (or time-travelling disks) can be used to increase system security by enabling post-intrusion analysis and recovery. Versioning systems designed for such security applications must retain all versions of all data and must handle version deletion in a secure manner to prevent malicious users from erasing incriminating evidence.

Empirical evidence shows that in typical commodity systems, kernel extensions such as drivers significantly reduce system dependability. Conversely, the complexity of monolithic kernels diminishes their appeal for use with truly critical applications. In both cases, increased dependability can be achieved by isolating the untrusted code; virtual machine monitors provide an ideal infrastructure with which to enforce this isolation.

Chapter 3

Secure Versioning

Versioning systems provide an attractive model for data storage. The abundant disk capacity available to modern systems has made traditional storage models, in which data is frequently destroyed in order to reclaim disk space, obsolete. Modern users should enjoy the confidence that, barring a disk failure, all data entrusted to the storage subsystem will be available under any circumstances—including user error.

There are a number of different manners in which the versioning model can be implemented, each with its own particular strengths and weaknesses. Current versioning systems suffer from one of two major shortcomings: inadequate security and inadequate utility. The problem faced by versioning system designers consists of coupling adequate flexibility with unimpeachable reliability.

3.1 The Ideal Versioning System

As evidenced by the previous chapter, versioning systems come in a variety of guises, from user-space applications to block-level drivers. For purposes of comparison and conceptualization, it is useful to enumerate a prioritized list of criteria by which to judge different designs. These ideal standards, listed below in descending order of importance, provide a vocabulary with which to conduct a rigorous evaluation of current systems and a means of articulating the abstract concepts from which an ideal versioning system can be composed.

Reliability: The degree to which a system can be relied upon to store and retrieve all data entrusted to it.

The primary goal of a versioning system is to safeguard data; thus we must first and foremost evaluate a versioning system according to its ability to reliably do so. A susceptibility to undesired loss of data constitutes an intolerable flaw.

Security: The degree to which a system can survive malicious attacks.

If a principal danger to data lies in buggy software, a no less insidious threat lurks behind every firewall: a menagerie of viruses, spyware, malware and other malicious entities pose a real hazard to users' data, and thus a versioning system should protect against these dangers as much as possible.

Flexibility: The degree to which versioning services can be adjusted to satisfy users' needs.

Versioning systems should accommodate users' needs rather than the converse. Systems which impose unnecessary or cumbersome restraints upon users are less appealing than systems which afford users as much freedom as possible without introducing vulnerabilities.

Adoptability: The ease with which a versioning system can be incorporated into a working system.

A system can be useful only insofar as it is used. If the difficulties of transitioning to a system are too great, it will quickly be relegated to the software graveyard, there to be mourned by its creators and forgotten. The ideal versioning system should easily integrate with current working systems.

Efficiency: The degree to which a versioning system impacts system performance.

While it is accepted that the costs associated with data versioning are warranted by the services rendered, ideal versioning systems should impose minimal performance degradation, both in terms of time and space.

Put concisely, the ideal versioning system should reliably protect data from both users and malicious software, offer users an appropriate degree of flexibility, and integrate easily into current working systems without imposing burdensome overhead. Even given the number and variety of current versioning systems, no existing solution presently satisfies all of the above criteria.

3.2 Existing Systems

There are two general classes of versioning systems: those which operate on file system objects, and those which operate on disk blocks. Each of these classes is broad enough to contain a number of diverse systems, but each

represents a fundamental design choice that significantly impacts the degree to which a system can satisfy the proposed criteria.

3.2.1 File-Level Versioning Systems

File-level versioning systems can be implemented as user-space applications, stackable file systems, or on-disk file systems. These systems deal with file system objects like directories and files, and as such are ideally situated to provide versioning at a per-file granularity—and because users typically interact with the storage subsystem by manipulating individual file system objects, per-file versioning is more appropriate than, say, per-volume or per-disk versioning.

Additionally, file-level versioning systems can enforce per-file policies. For instance, Elephant [47] allows users to specify version retention policies on a per-file or per-file-type basis. This is useful because the value of individual files is often related to their file types; for example, word processing documents are often much more valuable to a user than cached HTML objects. Another example of the advantages of versioning on a per-file basis can be found in VersionFS [34], which uses its knowledge of file system objects to employ a copy-on-change (rather than copy-on-write) approach to versioning. With copy-on-change, version histories are saved as delta chains of individual versions, and no two versions contain redundant data. This approach, which requires a knowledge of file system objects, reduces the amount of storage space required to maintain version histories.

Stackable file systems like VersionFS and Wayback [13] are highly adoptable, as they are compatible with a number of existing file systems. However, these systems have limited control of data structures maintained by the kernel and on-disk file systems. For this reason, these types of systems cannot prevent stale file versions from polluting the kernel's page cache; nor can they manage the allocation of inode numbers to ensure that file identifiers remain stable for use with systems like NFS. On-disk file systems like Elephant and ext3cow [39] have the ability to manage kernel data structures, and can thus make better use of the page cache and inode numbers. However, these systems are not compatible with other file systems, and are thus not easily adopted.

All file-level versioning systems, by virtue of their awareness of file system objects, share one common trait: complexity. File systems are large and complicated; they export rich interfaces, and are oftentimes used in unexpected ways; and they maintain a significant amount of sophisticated metadata to manage the objects they support. For these reasons, file systems

are notoriously difficult to construct, and even well-established, thoroughly-tested file systems like ext3 [1], JFS [2], and ReiserFS [4] have been found to contain logical errors capable of wiping out important data [62].

In general, file-level versioning systems are particularly well-suited to satisfy the flexibility and efficiency criteria listed above. Per-file policies allow file-level systems to provide rich, customizable functionality while minimizing the overhead required to do so. User-level applications and stackable file systems satisfy the adoptability criterion, as they are compatible with multiple file systems, but they incur slight performance penalties in doing so. On the other hand, on-disk file systems make better use of kernel data structures, but are not amenable to incremental adoption. Finally, all file-level versioning systems suffer from considerable reliability and security issues, as the complexity of such systems can introduce significant vulnerabilities.

3.2.2 Block-Level Versioning Systems

Block-level versioning systems like Clotho [19] and Peabody [32] operate beneath file systems and offer versioning at a per-volume or per-disk granularity. Unlike file-level versioning systems, block-level systems are generally incapable of supporting per-file policies. These systems deal exclusively with blocks, and this limitation is a double-edged sword: on the one hand, providing versioning at the block layer is much simpler than doing so at the file system layer, but on the other hand, block-level versioning often provides unsatisfactory utility.

Typical usage scenarios for versioning systems include allowing users to recover an individual file that has been erroneously destroyed. They do not often include recovering entire logical volumes. However, most conventional block-level versioning systems impose precisely this constraint; for instance, if a user wishes to roll back to a previous version of a single file with the Peabody system, she must roll the entire file system back to the time of interest and then wait for a file system consistency checker to verify that the disk is valid at the chosen time. There are applications, like kernel debugging and post-intrusion analysis, which can benefit from such coarse-grain version recovery, but for the common case of browsing the versions of one or a few files, this approach is cumbersome.

Additionally, block-level versioning systems often end up storing much more data than do file-level versioning systems. File systems always write data in units of blocks; this means that changing even a single byte of a file results in an entire block—up to as much as 4 KB—being written to disk. File-level versioners like VersionFS can avoid retaining redundant data by

analyzing file system objects, but block-level versioners are less capable of doing this and thus typically retain entire blocks. However, recent work in block-level delta-chaining has been shown to dramatically reduce block log sizes at the expense of an increased latency for random access to block versions [63].

Given a finite storage capacity, a versioning system's recovery window—or the period in which it can guarantee the recovery of a file version—is inversely proportional to the rate at which its log grows [53]; thus, the ability of a versioning system to support arbitrary version recoveries hinges upon its ability to curb log growth. Block-level versioning systems are particularly susceptible to log size explosions, and thus their efficacy is closely related to their ability to effectively manage storage space.

Block-level versioning systems do enjoy a few advantages over file-level versioning systems: block-level systems, due to their nescience of file system semantics, are compatible with a number of different file systems, and thus are adopted more easily than many versioning file systems; block-level systems do not pollute the page cache with stale data, nor do they alter the operation of the file systems they support; and the location of block-level systems beneath the kernel allows prudent users to isolate these versioning systems from erroneous or malicious impingements from higher-level software. In fact, block-level versioning systems can be made simple enough that they can be confidently included in a trusted computing base, thereby bolstering the security and reliability of the system.

The biggest advantage of versioning at the block layer is the increased reliability and security obtained by doing so. Block-level versioning systems are much simpler than versioning file systems, they are protected by a narrow, tractable interface, and they can be further isolated through standard trusted computing techniques—in short, they are considerably superior to file-level versioning systems both in terms of reliability and security. As well, they are compatible with multiple file systems and are thus more easily adopted than many versioning file systems. However, they often suffer from poor flexibility, as they typically only offer versioning at an unacceptably coarse granularity, and they are unable to implement per-file policies and optimizations.

3.3 Summary

When the value of data is significant, the primary criteria by which a versioning system should be evaluated are reliability and security. File-level

versioning systems, due to their inherent complexity, are simply unable to provide optimal reliability and security. An ideal versioning system should be simple enough that its correctness can be verified *a priori* through auditing, but the complexity of richly-featured file systems precludes any confident verification of their correctness.

Block-level versioning systems, due to their simplicity, can be certified as correct, but—also due to their simplicity—they cannot provide adequate versioning functionality. The instant a versioning mechanism is saddled with the responsibility of understanding files and their metadata—requisite knowledge for supporting per-file policies—its reliability decreases.

Table 3.1 summarizes the differences between the various types of versioning systems and indicates the need for a hybrid approach. In particular, the reliability, security, and adoptability of block-level versioning systems should be combined with the flexibility and utility of file-level systems.

Table 3.1: Comparison of Versioning System Attributes

	Reliability	Security	Flexibility	Adoptability	Efficiency
User-space tools			•	•	•
Stackable FS			•	•	•
On-disk FS			•		•
Block-level systems	•	•		•	

Chapter 4

Design

Secure versioning compels a stratified design. The mechanisms responsible for safeguarding data must be simple, perspicuous, modular—and isolated from the more complicated, less reliable mechanisms that are required to provide adequate utility.

VDisk achieves this separation of responsibilities by isolating the mechanisms which have write-access to versioned data. VDisk consists of three primary components: a secure logging module, a version reconstruction utility, and a secure log cleaner. Only the logging module and the cleaner have write access to versioned data; these two units perform only simple tasks, and compose the trusted component of the entire system. The version reconstruction process, which reads from the log but never writes to it, is the only component of the system which deals with file system objects. It is separated from the critical components and thus does not introduce vulnerabilities to the system.

4.1 Version Preservation

For maximum reliability, security, and adoptability, VDisk provides versioning from the block layer. To achieve this, a simple logging utility is situated beneath the file system as a virtual device driver in an isolated virtual machine. All write requests pass through this utility, where they are duplicated; one instance of the request is relayed to its original destination on the file system disk, while the other is appended to a secure log on a separate partition.

Conventional block layer logging utilities like Clotho [19] avoid duplicating write requests by introducing a level of indirection between file systems and the disk: write requests to logical block addresses are redirected to available physical locations, and all subsequent read requests to those logical blocks are mapped onto their corresponding physical addresses. This scheme is attractive because it obviates copy-on-write penalties by performing updates to the metadata that describes the logical-to-physical mapping. However, it poses two problems: it can vitiate any work done by file sys-

tems to achieve spatial locality, and it introduces new potentials for data loss—if the logical-to-physical mapping ever becomes corrupted, the entire file system could be lost.

Moreover, maintaining this indirection is a complicated task; to guarantee integrity in the face of arbitrary system failures, the logical-to-physical mapping must be handled with extreme care. Journaling and transactional semantics could be adopted to ensure mapping metadata is committed to disk appropriately, or perhaps a block-level consistency checker could be implemented to facilitate recovery from system failures, but something must be done to protect the mapping metadata. For this reason, VDisk does not interpose an extra level of indirection. Instead, it takes a performance hit to ensure the integrity of the file system's on-disk layout by duplicating write requests. Every write request thus commits data to both the original file system disk and the secure log.

The VDisk log consists of a metadata log and a data log. For each request submitted to VDisk's data log, a small entry is added to the metadata log which describes the salient features of the request—namely, the file system block address, the log block address, the size of the request, and the time at which it was committed; these entries also contain a flag byte used by the cleaner to record block deletions.

The log disk partition is sub-divided into large, fixed-sized segments that are threaded together on three lists: the *metadata log*, the *data log* and the *free list*. The metadata and data logs are written in append-only fashion, with new segments allocated from the free list when needed.

4.1.1 Log Isolation

The simplicity of VDisk's logging mechanism lends reliability to the system. It is reasonably well protected from erroneous higher-level software by the narrow block-level interface—no read or write request issued from upper-layer software can result in the destruction of logged data. However, if situated within the kernel, the security of the logging mechanism is limited to that of the kernel itself. In particular, if the kernel is compromised, the logged data is likewise put at risk.

To eliminate this vulnerability, the logging mechanism can be placed in its own protected domain through the use of a virtual machine monitor (VMM) like Xen [7]. VMMs enable the creation of multiple virtual machines on a single computer; each virtual machine is isolated and protected by the VMM, which prevents interference from other virtual machines. The VDisk logger, which is implemented as a block-level driver, can be placed

in its own protected domain; from there it can export its interface to an untrusted virtual machine while remaining beyond the control of the kernel it services. In this manner, the security of versioned data is decoupled from the security of a user's kernel—even a compromised VM operating system cannot destroy logged data. This approach is similar to that of S4 [54], which uses a network interface to protect its versioning file system.

4.1.2 Optimizations

VDisk's logging mechanism benefits significantly in terms of reliability, security, and adoptability by operating at the block layer. However, the constraints imposed by the simplicity of the mechanism and the narrowness of the block-level interface present a few impediments to achieving optimal performance. The most notable obstacle is the necessity of writing data twice—this requirement alone immediately halves the available bus bandwidth of the original system. While this bandwidth reduction substantially degrades throughput for bandwidth-bound workloads, the application-perceived penalty of VDisk is much smaller for seek-bound workloads. VDisk writes its data sequentially, while in many cases file systems scatter their writes across the disk; the time required for these file system disk seeks often almost completely overshadows the increased latency introduced by VDisk's duplicated writes. In addition to imposing temporal overhead, versioning data at the granularity of entire blocks can lead to explosions in log size, as even single-byte updates in the file system will result in entire blocks being written to disk.

To address the write performance issue, a lazy writeback approach could be adopted during bursty writes. When the logger is barraged with many writes in a short period of time, instead of immediately writing data to the log, it can add a small entry to an in-memory hash table indicating the file system block addresses and timestamps of the writes. During the burst, all writes to distinct block addresses can be copied to the log lazily; only when multiple blocks are written to the same address do the older versions need to be logged punctually. When the bursty period has passed, the logger can then copy all blocks listed in the hash table from the file system partition into the log without having to compete with the file system for disk bandwidth. This optimization could improve performance for cases in which large files are being written to disk, as such scenarios typically do not entail multiple writes to a single block address. However, this technique would introduce extra overhead when blocks are rewritten within a bursty period, as this will require reading the old versions of the blocks from disk before the new

versions can be committed. We have not implemented this optimization.

To limit log size explosion, we have implemented a content-hashing mechanism to avoid retaining redundant data. File systems often send superfluous data to the disk; for instance, creating a file with the ext2 file system results in seven 4KB blocks being sent to disk, only one of which actually contains the contents of the file [32]. The other six blocks are metadata blocks, of which only a few bytes per block—128 bytes in the case of the inode block, one bit each in the case of the inode and data bitmap blocks—have actually been updated as a result of creating the file. If this superfluous data can be recognized as redundant, it need not be preserved. To this end, the VDisk logger compares certain write requests to a table of content summaries of recently-read blocks; if the summary of a block being written to disk is identical to its corresponding cached summary, it need not be logged.

In general, this strategy could impose a security vulnerability, as the accuracy of the log would be dependent upon the cryptographic strength of the hash function used: if a malicious agent could alter a data block in such a way that the new, corrupted block hashed to the same summary as its original block, the agent could modify files in the file system without the changes being logged. This could be protected against by making a full comparison of the blocks whose hashes are identical, but this would require either an extra disk read to fetch the original block, or a much larger memory footprint to cache recently read blocks.

While data blocks are vulnerable to such attacks, it seems unlikely that a malicious agent would be able to modify *inode* blocks in such a manner that it could realize its sinister ambitions—by creating new inode blocks whose hashes collide with those of their original counterparts—without corrupting the file system. That is, it would be highly infeasible to perform many unlogged operations before the entire file system became useless. Thus, there is less of a security concern in applying this optimization to inode blocks.

Even limiting content hashing to inode blocks can result in significant improvements: inodes are updated frequently, and the blocks in which they reside can quickly become a large portion of logged data—file system metadata blocks can ultimately require as much disk space as data blocks [53]—if they are preserved blindly. Thus in cases in which it is feasible to distinguish inode blocks from data blocks at the block layer, as it is with the ext2 and ext3 file systems, content hashing can help curtail log growth.

4.2 Version Browsing

VDisk's logging mechanism is simple and file system agnostic. To provide adequate utility, a suite of sophisticated user-space tools are relied upon to interpret file system objects like files and directories. These tools combine information stored in the log with a deep understanding of file system internals to enable users to interact with versioned file objects rather than versioned blocks. These tools never modify the logged data, and can thus safely operate outside of the system's trusted computing base.

4.2.1 User-Space Tools

The capabilities of VDisk's version browsing tools are not limited merely to individual file reconstruction. Theoretically, an entire read-only, user-space file system could be built on top of the log, providing users with the same features as standard file systems. However, considerably more work is required to reconstruct a file from the log than is required to read a file from a typical file system, as the reconstruction process can involve searching large portions of the log for particular versions of particular blocks. An operation such as giving a detailed listing of a directory's contents could be prohibitively expensive for a file system built on top of the log because the log is temporally structured and may therefore fail to preserve spatial locality. This performance penalty could possibly be sufficiently mitigated with the use of a relational database to expedite the searching process, but for the purpose of our prototype, we decided against implementing an entire file system.

Instead, we provide two reconstruction tools. The first can recover a specific version of a specific file; the user indicates a file path and a time, and the tool reconstructs the newest version of the file that existed before the given time. The second tool is slightly more general: given a file path, it lists all versions of that file stored in the log; the user is then able to choose one or a number of versions to reconstruct. Similar tools could display or reconstruct the contents of a particular directory version. If desired, such tools could even reconstruct entire volumes, although in most cases the applicability of such coarse-grain reconstruction is limited to highly specific tasks like kernel debugging [28] and post-intrusion analysis [14, 27].

While the ability to browse a file's version history is crucial, we expect that it will not be used very frequently and thus does not warrant optimizations that could adversely affect common operations like reading and writing to current file versions. The mechanisms responsible for browsing read-only

version histories need not—and indeed should not—be implemented within the kernel.

There is a fundamental difference between current file versions and version histories: the former are mutable and frequently accessed, while the latter are immutable and infrequently accessed. The latency associated with using current versions needs to be minimized, which is one reason why standard file systems are included within the kernel, but there is simply no need to include version browsing mechanisms in the kernel. Moreover, log-structured, read-only version histories are amenable to current data management trends which make use of powerful indexing tools to organize and search data. Just as complicated indexing processes, such as those required by Google's search engine, are more appropriately implemented in user space, the tools needed to index and browse VDisk's read-only file histories are better suited to user space implementations.

4.2.2 VDisk Metadata Database

For the process of reconstruction, the most frequently used portion of the log is the VDisk metadata—in particular, file system block addresses and timestamps are referred to regularly while interpreting file system objects and their versions. As mentioned, VDisk metadata entries are grouped together in log segments; these segments are linked together to form the entire metadata log. For convenience and efficiency, VDisk's user-space tools copy this metadata log, which typically constitutes less than 0.5% of the log for file systems with a 4KB block size, into a relational database.

Whenever a user-space tool is used, any new VDisk metadata entries in the log are added to the database; the reconstruction tools then deal exclusively with the metadata log contained within the database. This greatly facilitates the reconstruction process, as the brunt of the work is handled by the database, which indexes the log and services queries based on file system block addresses and timestamp values. Leveraging a relational database in the version reconstruction process is just one example of the benefits to be gained from implementing version management tools in user space; the database could also be used in the implementation of additional features, such as content indexing of versioned files.

4.2.3 File Reconstruction

The process of file reconstruction is a file system-specific process of scanning through the logged data—interpreting file system objects like inodes and

directories along the way—until the desired file version is located. It is based upon the assumption that all file system metadata is eventually written to disk—and duly logged—and can thus be reconstructed to obtain any file system object at any time in history. Our reconstruction approach is similar in spirit to that of gray-box systems such as D-GRAID [50]. However, while D-GRAID uses its understanding of file system data structures to identify the type of the file system stored on its disk, VDisk combines a similar understanding of data structures with *a priori* knowledge of the file system type to reconstruct file system objects from the block log.

VDisk can only be used to reconstruct file versions which are committed to disk; versions which are overwritten in memory are irrecoverable. As well, while both trusted components of VDisk—namely, the logger and the cleaner—are compatible with any file system, the reconstruction tools are not; a distinct set of tools must be developed for each supported file system. However, a collection of auxiliary libraries—such as the VDisk-specific interface to the metadata database—can simplify this development process.

The general algorithm for reconstructing a file version consists of two tasks: finding the inode of a file which corresponds to its desired version, and collecting the correct versions of the data blocks referenced by the inode.[1] Most of the work entailed by this algorithm lies in querying the log's metadata database in search of entries for particular file system blocks at particular times. Relational databases are ideally suited for these types of searches, and are thus employed liberally in the implementation of this process.

The first task of the algorithm is similar to the standard path resolution algorithm: starting from the root directory, the process checks the contents of each directory for the corresponding name listed in the path. If a match is found, the process recurses on the newly found directory. If not, the process returns a message indicating a failure to resolve the path.

However, when resolving *all* versions of a path, this algorithm is complicated by the introduction of a new temporal dimension. This means that a single path resolution failure does not indicate an absence of the desired path—the path could exist in a different version of the file system. Thus the path resolution procedure must be repeated on each version of the file system until either the path is fully resolved or the given time constraints are exceeded.

For example, to list the contents of a directory that we know existed

[1] This algorithm will also work for file systems that use metadata structures other than inodes, like VFAT; we limit the following discussion to inodes for convenience.

at time t, the following query can be used to locate the appropriate root directory inode, assuming that the root directory has a block address R and the file system adheres to an ordered write model:

```
SELECT log_block_address FROM vdisk_metadata
   WHERE fs_block_address = R AND time < t
   ORDER BY time DESC LIMIT 1
```

Because we know the directory existed at time t, we know that the newest version of the root directory written before t will contain a path to the target directory. This query returns the log address of the block containing the newest root inode before t with a path to the target directory. Given this address, we can find the inode number of the next directory in the path, translate this inode number into a block address, and perform the next iteration of the path resolution algorithm.

If we do not know the exact time that a directory existed, this process must be modified. For instance, imagine that we are given a time range (t, $t + \delta$), and we want to find the newest version of a directory that existed at some point within this range. To do this, we would query the database for a list of all appropriate root directory versions with the following command:

```
SELECT log_block_address FROM vdisk_metadata
   WHERE fs_block_address = R AND time < t + δ
   ORDER BY time DESC
```

Notice that in this case we cannot limit the query to a single result. Instead, we must collect a list of root directory versions that were written to disk before $t + \delta$. We sort this list in reverse chronological order and perform the path resolution process on each version until we successfully find the target directory. This query places no lower bound on the time of the root directory, because even though we are only interested in versions of the target directory that existed after time t, it is possible that no versions of the root directory were logged between (t, $t + \delta$). For instance, if the target directory was created at time t_0 such that $t_0 << t$ and the root directory persisted unchanged after the creation of the target directory, the newest version of the root directory may be much older than the newest version of the target directory, since subsequent updates to the target directory will not produce new versions of the root directory. Thus lower time bounds for the target directory do not always apply to every directory along the target path.

In the most general case, we can search the entire file system history for every version of a particular directory. To do this, we generate a list of all versions of the root directory:

```
SELECT log_block_address FROM vdisk_metadata
```

```
WHERE fs_block_address = R ORDER BY time ASC
```

While this first query might return a large set of blocks for aged file systems, subsequent queries on subdirectories will be performed with converging time constraints. For example, if the first root version existed at time t_0 but the next directory in the target path, residing in block B, was not created until time t_x, the second query performed in the recursive path resolution algorithm need not include the extraneous times:

```
SELECT log_block_address FROM vdisk_metadata
   WHERE fs_block_address = B and time >= t_x
   ORDER BY time ASC
```

Once the appropriate version of the file's inode is found, the second task of reconstruction is commenced: appropriate versions of all the blocks referenced by the inode are collected and written in order to a user-specified output file. The process of collecting the appropriate versions of blocks is simple for file systems which impose order on writeback operations (i.e., file systems which flush all data to disk before writing its corresponding metadata): given a version of an inode, we simply take the newest versions of the referenced data blocks that are older than the inode. For file systems that might write metadata blocks to disk before flushing data blocks, this process is more complicated.

Consistency Issues

To successfully reconstruct file system objects, the VDisk reconstruction utility must be able to understand the relationship between versioned blocks at any given time. Part of this understanding comes from the file system metadata, such as inodes. However, some of it must be inferred from the implicit temporal information contained in the log. For example, simply by inspecting an inode, the reconstruction utility can determine the addresses of any blocks that belonged to a particular file version, but it must infer the period in time during which the blocks belonged to that version by evaluating the time at which the blocks were written to the log.

If a file system imposes write ordering, guaranteeing that data is written to disk before its corresponding metadata, it is straightforward for the reconstruction utility to infer the temporal relationships of blocks. Many newer file systems do in fact impose such write ordering, as it helps to avoid file system inconsistencies during arbitrary system failures. However, some file systems, such as ext2, do not impose write ordering, making file reconstruction more difficult.

Moreover, the write ordering imposed by some file systems does not typ-

ically preserve application-level consistency. For example, if an application writes block A then B of a file, a version of that file that contains the new value of B, but not A, is inconsistent. In the VDisk metadata log, these writes are represented by two timestamped entries. They are written to the log and timestamped, however, in the order the blocks are *delivered to the block layer*, which may be different from the order the application writes them if disk writes are asynchronous to the application, as they typically are in local Unix file systems. As a result, the log might record B before A. If so, VDisk's version reconstruction mechanism risks delivering an inconsistent version of the file if a user requests a version at a time that falls between the timestamps of these two entries.

For file systems that impose write ordering, this problem is confined to a single file. For file systems that do not impose write ordering, inconsistencies can extend across multiple files. For example, if an inode is truncated and one of its blocks is allocated to a new inode shortly thereafter, the log may record two different inodes that reference the same data block at nearly the same time. If the time gap between the two inode entries and the data block entry is short enough, it is impossible for VDisk to tell whether the block version belongs to the old inode or the new one.

In all of these cases, the problem is resolved if there is a bound on the time between when an application writes to a file and when the resulting block modifications are written to disk and if there is a period of update quiescence for each file that is at least as long as this bound. In this case, the version-access tool can ensure consistency by only delivering file versions that have been quiescent for the disk-write-bound period of time. For example, most UNIX systems flush dirty blocks to disk periodically, every 30 seconds. For these systems, the version-access tool can guarantee application-level consistency by restricting the file versions it reconstructs to those that remain unchanged for at least 30 seconds. It is easy for the tool to establish this constraint by examining the timestamps of metadata entries and rolling forward when necessary until the gap between the block versions it selects and the next version of those blocks in the log is at least 30 seconds.

A potential problem remains, however, for files that are accessed so frequently that there are insufficient periods of quiescence or for file systems that provide no bound on how long updates can be cached in memory. In these cases there is little VDisk can do but rely on higher-level tools to determine which reconstructed versions are consistent.

Fortunately, applications that care about the consistency of disk data typically have application-level (or in the case of file system metadata, file

system-level) consistency constraints that, when combined with the VDisk version information, can be used to extract consistent versions in a straightforward manner. An application that uses atomic transactions to update a file, for example, places specific ordering constraints on transaction log updates and between certain log entries and target-file checkpoints. As a result, the transaction log is properly ordered in the VDisk logs and any transaction log entry that establishes a consistent checkpoint is properly ordered with respect to updates to the target file. It is thus straightforward for the transaction system's recovery manager to establish version consistency in much the same manner it would when recovering from a crash. As another example, consider a file updated in append-only fashion. In this case, the ordering constraint on VDisk log entries is the logical block number of updates and not their timestamp, so the VDisk-log order is irrelevant.

These consistency issues are an undesired artifact of the difficulties of providing file-grain versioning at the block level. Such difficulties can lead to situations in which reconstructing data can be difficult, sometimes requiring help from higher-level applications—but the data is still available. In contrast, more complicated versioning file systems are better suited to handle these consistency issues, at the cost of a less dependable system. Thus while it is less likely that a user will encounter consistency issues with a versioning file system, it is arguably more likely that a user will encounter security and dependability issues, which could result in the irrevocable loss of data.

4.3 Version Pruning

Version pruning is a sensitive process that must be handled with extreme care: an insecure delete mechanism could easily undermine the reliability of the entire versioning system. Moreover, providing support for arbitrary, user-initiated version pruning would make VDisk susceptible to the very dangers that versioning aims to protect against. For these reasons, a more structured deletion interface is called for.

A policy-based deletion scheme is attractive because policies can prevent users from erroneously deleting important information. Policies require that users articulate retention strategies only once, during log installation; after this, the established policies will protect information automatically and transparently, even in the event of user error. From a reliability standpoint, policies are appealing because they can enable secure version pruning. However, their feasibility as a practical deletion strategy depends upon whether or not efficacious policies—i.e., policies that protect important information

and enable the reclamation of unneeded blocks—can be suitably expressed.

To protect against both erroneous and malicious mishaps, VDisk provides a trusted log cleaner, which is responsible for serving all delete requests. Before any delete request can be satisfied, the log cleaner must verify its legality with respect to the established retention policies.

The log cleaner is trusted, and as such should be as simple as possible. However, it is often difficult to verify that a delete request will not destroy important file system information: retention policies can impose sophisticated constraints that can be difficult to evaluate, especially at the block layer. For this reason, every delete request is couched in terms of a simple proof. In addition to describing which blocks should be deleted, this proof provides adequate evidence that the deletion is in accordance with the retention policies established during the installation of the log. The secure cleaner needs only to verify the correctness of the proof and, if appropriate, perform the deletion.

4.3.1 Deletion Proofs

In designing the secure cleaner, we have again followed a stratified approach. We use an untrusted, user-space application to query the metadata database in search of segments that contain a large number of deletable blocks according to the policies being used. This application then sends a list of proofs to the secure cleaner, which performs the simple verifications required to confirm that no important blocks will be deleted.

In general, much more effort is required to construct proofs than to verify them. By separating the tasks of creating and checking proofs, we have developed a two-tiered system in which the difficult work is done in user space while the critical task of protecting data is reduced to simple proof verifications. Our approach is similar to that of Exokernel [17], in which a simple resource manager ensures equitable resource allocation by evaluating predicates submitted by higher-level systems.

The feasibility of this approach hinges upon the ability to construct proofs which adequately evince the expendability of a block. One possible method of expressing these proofs would be to develop a domain-specific language in which both block layer and file system layer information could be communicated. For instance, upon installing the log, a user could register a number of templates with the secure cleaner; these templates could be used to verify the type of a particular block and also to extract important information from it, such as which other blocks it is dependent upon. If the relationships between these templates are adequately described, the secure

cleaner could use them to check sophisticated, file system-aware proofs.

For example, consider a name-based policy that stipulates the retention of all versions of any file with a ".doc" suffix while allowing the reclamation all other files. With this policy, the simplest way to prove that a block is deletable—assuming that files do not share blocks—is to demonstrate that it belongs to a file (or file system object) whose name is not suffixed with ".doc". This entails producing a version of an inode that references the block and producing a version of a directory entry that names the inode. For complete security, the directory entry must be authenticated as actually belonging to the file system metadata (rather than existing as the data block of a specially-contrived file); for file systems with dynamic directory block addresses, this requires tracing the path of the directory containing the proffered entry all the way to some system invariant, such as the root inode or superblock, whose address remains constant throughout the lifetime of the file system.

To enforce this policy at the block layer, VDisk's secure cleaner would require template functions capable of:

- authenticating the system invariant (e.g., identifying a block as the root directory inode)
- interpreting inode data block references
- interpreting directory entries
- translating the inode identifiers given by directory entries into inode block addresses

With these templates at its disposal, the secure cleaner could verify a block's eligibility for deletion by evaluating a proof consisting of a link of block addresses, starting with the system invariant and ending with the data block in question. The cleaner would authenticate the root directory and interpret it to ensure that one of its entries named the inode designated by the next block address in the proof; at this point, the cleaner can consider that inode authenticated. This process, which is essentially a process of verifying the translation of a file system path into a sequence of block addresses, would be continued on each successive block in the proof, until the cleaner could verify the name and authenticity of the inode which references the block to be deleted. Having done this, the cleaner would also have to verify that no other versions of the inspected blocks exist with timestamps greater than the inspected versions and less than the block to be deleted; this is necessary to prove that the correct version of the path is being evaluated

and that, for instance, there does not exist a newer version of the path in which the target file name has a ".doc" suffix.

The complexity of implementing this mechanism would vary with file systems. For instance, in ext2 and ext3, the first and last template functions can be implemented with simple block address evaluations: the root inode always has an inode number of two, and inode numbers are deterministically mapped to block addresses. For file systems with dynamic inode allocation—such as ReiserFS, for example—the process of translating an inode number into a block address would involve traversing the inode tree which maps inode numbers to block addresses; additional template functions might be required to perform this translation at the block layer.

While templated deletion proofs would enable the enforcement of semantically richer retention policies at the block layer, we have excluded file system information from our proofs. A central idea of VDisk is to keep critical components as simple as possible, so we have opted not to implement a file system-aware cleaner. As Section 7.2.3 shows, this is a viable approach, because the secure cleaner is still able to reclaim a substantial number of unneeded blocks without interpreting file system semantics.

Our proofs are limited to ordered lists of block version descriptors, which contain a block's file system and log block addresses as well as the time at which it was written. This makes the secure cleaner's job exceedingly simple, but it imposes the constraint that all retention policies must be expressible solely in terms of blocks.

This proof-based deletion system provides a dependable infrastructure that can be used to guarantee the security of every logged block. This infrastructure allows users to submit deletion requests at will while ensuring that any operation that could result in the destruction of logged data must be explicitly validated by the secure cleaner. The secure cleaner provides reliable protection against both incidental and malicious data destruction by evaluating simple proofs, and the logic required to implement this cleaner is quite simple; all of the difficult logic is pushed into the user-space process of constructing deletion proofs, where an error will result in an invalid proof rather than the illegal destruction of data.

4.3.2 Retention Policies

The secure cleaner is simply an enforcer: it ensures that all deletion requests adhere to the retention policies established during the initialization of the log. The retention policies are crucial because they must express the rules that will prevent the loss of important information. Developing general

guidelines about which blocks should be eligible for deletion can be a difficult task, and in the case of VDisk, this difficulty is exacerbated by the fact the these rules must be expressible purely in terms of blocks.

VDisk's retention policies are derived from a model of file system access patterns developed by Elephant [47]. One key notion of this model is that important files which are modified by users typically vacillate between two states: a volatile state, in which the files are updated frequently as new changes are made, and a stable state, in which landmark versions of the files persist unmodified for a period of time. Landmark versions are important for two reasons: the first is that they seem to indicate versions of files which users are satisfied enough with to leave unchanged; the second is that they represent versions of files which users are likely to be more familiar with and remember for longer periods of time.. Volatile versions of files, on the other hand, are not around for very long, and seem therefore to be of less value to users, who quickly overwrite them; it also seems less likely that these ephemeral versions will stand out in users' minds for very long. As time progresses, we expect that the minute differences between various volatile versions of a file will fade from a user's memory, while the differences of landmark versions will remain more memorable.

With this model in mind, we have developed two policies, *Keep Safe* and *Keep Milestones*, which are enforceable from the block layer and which we feel will be of value to typical users. These policies can be used individually or in conjunction, and additional policies can be added to the system as seen fit.

Keep Safe

The Keep Safe policy stipulates that any file system update must be reversible for a specified interval, called the *Keep Safe Window*. This policy is employed by S4 [54], which is intended to support post-intrusion analysis and thus must retain all file modifications to enable accurate log replays. The Keep Safe policy is also helpful for typical users, as it provides liberal undo capabilities [47].

To ensure that any file system modification occurring within the Keep Safe Window can be undone, it is necessary to keep some block versions that are outside of the Keep Safe Window. For example, consider a file that has not been modified for one year in system with a Keep Safe Window of thirty days. If this file is updated, there will be two versions of it in the log: one which is a year old, and one which is current. To ensure that the newest update can be undone for thirty days, the year-old version cannot be

reclaimed until it is 395 days old, at which point the new update is pushed out of the Keep Safe Window and is no longer protected by the Keep Safe policy.

The proof that a block may be deleted according to the Keep Safe policy consists a reference to the candidate block and a reference to a proof block, which is a different version of the candidate block. The cleaner must verify that the following constraints are satisfied:

1. the candidate block and the proof block both correspond to different versions of the same file system address
2. the difference between the current time and the timestamp of each block is greater than the Keep Safe interval
3. the candidate block is older than the proof block

If all three of these conditions hold, then the candidate block may be deleted. Otherwise, the proof is invalid and the delete request is denied.

Keep Milestones

The Keep Milestones policy is similar to the *Keep Landmarks* policy of Elephant [47]. The Keep Landmarks policy retains stable versions of files, with the exact definition of "stable" changing as versions age. For example, one-month old files may have to persist unchanged for only one day to qualify as landmark versions, while one-year old files may have to persist unchanged for a month before they are considered landmarks.

The Keep Landmarks policy operates exclusively on file system objects, and is thus difficult to express in the block-level proofs required by VDisk's secure cleaner. The Keep Milestones policy is an approximation of the Keep Landmarks policy. Because VDisk operates beneath the file system, it has no way of ascertaining when a particular file has been closed, and as such, it cannot determine what block writes constitute the last update to a particular version of a file. It *can* observe the write patterns to individual blocks, however, and any block which goes unmodified for some threshold period of time can be considered a milestone block.

Note that not all milestone blocks necessarily correspond to landmark file versions—a volatile file version may contain one block that has persisted unchanged for a long period of time—but all landmark files are composed entirely of milestone blocks. Thus by retaining all milestone versions of blocks, VDisk can ensure that all landmark files will be reconstructable. However, the Keep Milestones policy will result in the retention of blocks

that could have been reclaimed by a file system enforcing a Keep Landmarks policy.

For instance, consider the scenario in which a file, originally composed of three blocks, is truncated to a size of zero. In this scenario, a new inode for the file will be written to the log upon the truncation, but the three truncated blocks will not. If the file persists unchanged in its truncated form for an adequate period of time, the Keep Landmarks policy will mandate the retention of the empty file, while the three truncated blocks could be reclaimed by the file system. However, the Keep Milestones policy is concerned only with block writes: any block that is not overwritten for a predetermined amount of time must be retained. Thus, if the three truncated blocks are not allocated to a new file and written within the milestone window, the Keep Milestones policy will mandate the retention of these blocks, even though they do not belong to a landmark file version.

The primary disadvantage of the Keep Milestones policy is that it cannot reclaim blocks as aggressively as can the Keep Landmarks policy: due to a dearth of file system information, the Keep Milestones policy must be conservative in its retention rules. However, this conservatism does result in a policy that will retain all landmark file versions, and it is applicable to a number of file systems. Moreover, because it can be expressed solely in terms of blocks, it fits nicely within the secure framework of VDisk. So long as this policy allows for the reclamation of an adequate number of blocks, it constitutes an appealing secure deletion policy.

The conservatism of the Keep Milestones policy arises from an additional constraint that is not imposed by the Keep Landmarks policy. According to the Keep Landmarks policy with a retention window of δ seconds, a file version must be retained if it does not change for at least δ seconds. However, an analogous Keep Milestones policy, again with a retention window of δ seconds, must impose two constraints: (1) a block version must be retained if it does not change for at least δ seconds, and (2) a block version must be retained if there do not exist two additional versions of the block, one older and one newer, that fall within δ seconds of each other.

This extra constraint is required by the Keep Milestones policy because the secure cleaner does not distinguish file system inode blocks from other data blocks; at the block level all blocks look the same. This poses a potential problem for inode blocks, because they store multiple inodes and are thus shared among multiple files.

For example, consider the following update sequence to two files X and Y that share an inode block: extend X at t_1, extend Y at t_2, extend X at t_3. Figure 4.1 shows the four inode block versions this sequence would

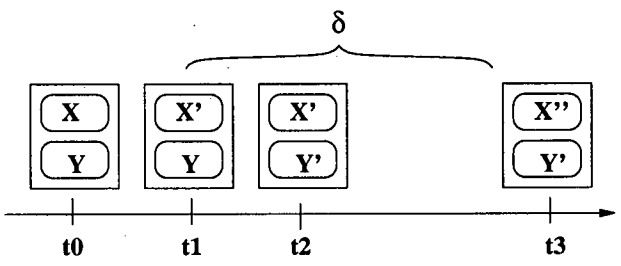

Figure 4.1: Inode Block Example

produce in the log. Assume the milestone interval for this example is δ. At first glance, it seems that only the block version at time t_3 should be retained, because no other block versions persist unchanged for longer than δ seconds. However, the version of file X that was extended at time t_1 persisted unchanged for more than δ seconds, and so its inode, which exists in the block versions at t_1 and t_2, must be retained. With the extra Keep Milestones constraint, the block version at t_0 must be kept because no older versions of the block exist; the version at t_1 may be deleted because it is bracketed by two versions that are within one milestone period of each other, while the version at time t_2 must be kept because (after the version at t_1 is deleted) it is *not* similarly bracketed; and the version at t_3 must be kept because there is no newer version of the block.

Because blocks can be shared between files, it is not sufficient merely to keep entire block versions that persist unchanged for the milestone interval: VDisk must retain any *portion* of any block that persists unchanged for the interval. The milestone proof constraints guarantee that this requirement is upheld by retaining the first and last version in each sequence of versions that exist within the same milestone interval. Any byte of any intermediate block in such a sequence will either be identical to the corresponding byte of the first version in the sequence, or identical to the corresponding byte of the last version in the sequence, or different than the corresponding bytes of both the first and the last versions in the sequence. In the first two cases, the block can be deleted because it contains redundant data. In the last case, the block can be deleted because it did not persist unchanged for the milestone interval.

A milestone policy for a candidate block version must contain a reference to two proof block versions. The secure cleaner verifies the Keep Milestones constraints by reading the metadata entries of the candidate block and its

two proof blocks and ensuring that the following criteria hold:

1. all three entries correspond to different versions of the same file system block address
2. the candidate entry's timestamp falls within the range defined by the two proof blocks' timestamps
3. the difference of the two proof blocks' timestamps is less than the milestone interval

If these constraints are not satisfied, the secure cleaner denies the request.

Combination of Policies

Both the Keep Safe and the Keep Milestones policies can be used individually, but we feel they are better used in conjunction. A Keep Safe retention period—perhaps of one week, or one month—can be established, within which all versions of all blocks are retained. After this period, blocks can be reclaimed according to the Keep Milestones policy. This combines the liberal undo capabilities of Keep Safe with a more selective retention policy for older files in an attempt to find a reasonable balance between storage reclamation and data preservation.

4.3.3 Security Issues

VDisk's secure cleaner must be capable of operating correctly and protecting data even if the user-space tool misbehaves or is compromised. The secure cleaner overcomes its distrust of the user-space tool by using retention policies, translated into the form of block constraints, to verify each proof submitted from user space before any block is deleted. While the secure cleaner expects that the proofs it receives will be valid, there are a number of reasons why this might not always be the case.

In the simplest scenario, the user-space tool might construct an erroneous proof. For instance, when constructing a milestone proof for a block with file system address B, a bug in the user-space tool might cause it to submit a proof referencing a version of block with address B'. Because a milestone proof requires references to three different versions of the same file system block address, the secure cleaner would quickly find that this proof is invalid and would therefore deny the delete request.

A more insidious scenario could arise if the user-space tool was compromised. For example, an attacker might construct a list of proofs in which

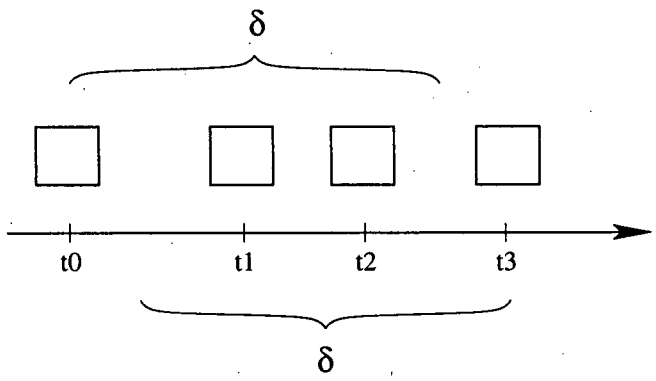

Figure 4.2: Block Version Timeline

certain entries have been omitted in an attempt to fool the secure cleaner into marking blocks as eligible for deletion when they are not. However, the secure cleaner can survive such an attack because the algorithm for enforcing the block policies is such that any omissions can only result in the retention of blocks which would otherwise be eligible for deletion, and never the converse. The milestone check is implemented by comparing a given block to a previous version and a newer version; if a malicious process provided a list which omitted a block's true previous version and instead indicated an even older version, this would only increase the interval between the two proof block versions, reducing the chances that the candidate block could be deleted. Similarly, an attacker could not achieve the illegal deletion of a block by providing an inaccurate reference to a newer version, because doing so would again increase the interval between the two proof blocks.

However, the Keep Milestones policy does allow some room for manipulation by an attacker. For example, consider the case illustrated in Figure 4.2, in which four versions of a block are written to the log at times t_0 through t_3. The version at time t_0 must be retained because there is no earlier version, and the version at time t_3 must be retained because there is no newer version. There is some ambiguity concerning the versions at times t_1 and t_2: if they are both retained, they are both eligible for deletion, but when one of them is deleted, the other must be retained. In a sense, block reclamation is a contract: the secure cleaner will allow the deletion of some blocks, so long as doing so will not disrupt the invariant that one version of a block is retained for every milestone interval in the log that contains one or more versions of that block. In this example, an attacker could contrive to have

either the version at t_1 deleted, or the version at t_2 deleted, but she could not have both versions deleted.

This is still in accordance with the Keep Milestones policy—which mandates that any block version which goes unchanged for δ seconds must be retained—because if some portion of the version at time t_1 did not change for δ seconds, it must be identical to the corresponding portion of either the version at time t_0 or t_2, and similarly, if some portion of the version at time t_2 remained unchanged for δ seconds, it must be identical to the corresponding portion of either the version at time t_1 or t_3. Thus any important information that would have been retained by keeping the block version at t_2 will likewise be retained by keeping the block version at t_1, and vice versa. However, if an attacker made incriminating changes at t_1 and overwrote them at t_2, she could destroy the evidence of her attack by deleting the version at t_1 and retaining the version at t_2.

This illustrates an important characteristic of the Keep Milestones policy: it does not guarantee the specific times at which block versions will be retained, but it does guarantee the retention of block versions which persist unchanged for a sufficiently long time. This policy is clearly not appropriate for post-intrusion analysis, as any changes made between milestone versions of blocks will not be retained.

As well, the Keep Milestones policy is susceptible to an attack in which an intruder could cause a file that would otherwise become a milestone file to be reclaimed. To achieve this, the attacker would have to update a user's file version within the milestone interval of its last milestone version and then ensure that the file persisted unchanged for the rest of the milestone interval. At this point the intruder's version is a milestone version and the user's is not, though it would have been without the intruder's update. The user's version is retained for the Keep Safe period, but following this, the user's version can be deleted without violating the Keep Milestones policy.

Keep Milestones thus provides a weaker intrusion detection window once the Keep Safe interval has expired. Nevertheless, it is weakened by a declarative constraint that limits the versions that can be deleted. The constraint ensures, for example, that once a version becomes a milestone it is invulnerable to attack. We hope that this principled approach to version deletion will be of substantial benefit in limiting the damage an intruder can inflict even after the Keep Safe interval.

VDisk's deletion proofs allow the secure cleaner to evaluate a block's eligibility for deletion without expending a lot of effort. In the case of milestone proofs, the secure cleaner is spared the effort of scanning the metadata log in search of a block's previous version and its next occurring

version; instead, this work is done in user space with the aid of a relational database, while the secure cleaner's task is limited to performing a few simple verifications. Even when dealing with erroneously or maliciously constructed proofs, the secure cleaner is able to quickly determine whether or not a block is really eligible for deletion.

4.4 Summary

VDisk is designed to provide secure, reliable versioning. To attain this goal, we have isolated the critical components of VDisk. The mechanisms which constitute potential vulnerabilities to the system, namely the logger and the cleaner, perform only very simple tasks. These mechanisms are so simple that they can be audited for correctness and confidently included in the system's trusted computing base. Furthermore, these mechanisms can be protected by a virtual machine monitor, and can thus safeguard versioned data even when a user's kernel has been compromised. This simplicity and security is achieved at the expense of write throughput.

While the critical components of VDisk remain ignorant of file system semantics, user-space utilities make use of a thorough understanding of file system internals to enable the reconstruction of individual file versions. The process of file reconstruction entails querying the log for desired block versions, which are interpreted as file system objects to produce the desired files or directories. While file reconstruction can be an involved process, it does not modify versioned data and thus does not introduce reliability or security vulnerabilities.

Because user-initiated storage space reclamation can constitute a significant threat to versioned data, we have designed an automated log cleaning process. A secure log cleaning mechanism operates from within the trusted computing base to enforce deletion policies and protect against erroneous or malicious delete requests. An untrusted user-space application is responsible for the more difficult tasks of locating segments which will most benefit from cleaning and providing the secure mechanism with proofs of each block's eligibility for deletion.

Chapter 5

Algorithm Details

VDisk logs blocks rather than file system objects, but a primary objective of VDisk is to support file system-level versioning and cleaning. To achieve this, some means of translating the block log into a file system history is necessary. This process of translation is of course highly dependent upon file system characteristics. There are two aspects of some file systems that can make this translation particularly difficult: a lack of write ordering [51], and the sharing of blocks by multiple file system objects.

Newer file systems, such as ext3 and Reiserfs, impose constraints on the order in which data is written to disk. To avoid inconsistencies, these file systems ensure that when a metadata object references data blocks, the data blocks are committed to disk before the metadata blocks. VDisk can use these ordering constraints to make important inferences about the sequence of file system operations captured in a log. However, some older file systems, such as ext2, do not impose any write ordering constraints, thereby reducing the amount of information available to VDisk during file reconstruction.

Additionally, some of the policies employed by VDisk's secure cleaner work best under the assumption that blocks are not shared between files. While this is typically the case for data blocks, it is not for metadata blocks. Because most file descriptors are significantly smaller than a block, file systems tend to pack multiple descriptors into a single block. For this reason, special care must be taken when designing VDisk's retention policies.

This chapter presents a formal description of some of VDisk's algorithms and provides a more detailed discussion of the difficulties mentioned above.

5.1 Notation and Definitions

- A **block variable** B is defined by the tuple $B_{(A,L,T)}$, where
  - A indicates a file system block address
  - L indicates a block log location
  - T indicates a block timestamp value

- A **block instance** is a block variable whose parameters are bound. Block instances are represented with lowercase parameter variables: $B_{(a,l,t)}$. Block variables may also be only partially bound. For example, $B_{(a,L,T)}$ represents the set of all blocks with file system block address a; this set may contain many block instances with different log locations and timestamp values, and thus these parameters are left in uppercase, indicating their free status.

- An asterisk next to a block instance indicates that the block is *live*:

$$B^*_{(a,l_a,t_a)} \longrightarrow \neg\exists B_{(b,l_b,t_b)}[(a = b) \wedge (t_b > t_a)] \tag{5.1}$$

- A subscripted set variable contains exactly the number of elements indicated by its subscript:

$$S_n \longrightarrow |S_n| = n \tag{5.2}$$

- $A \preceq B$ denotes that A *immediately precedes* B. That is, there is no block instance C with a file system block address identical to A which existed after A and before B:

$$(A_{(a,l_a,t_a)} \preceq B_{(b,l_b,t_b)}) \longrightarrow \neg\exists C_{(c,l_c,t_c)}[(c = a) \wedge (t_a < t_c < t_b)] \tag{5.3}$$

- $B_{<\delta>(a,l,t)}$ denotes that $B_{(a,l,t)}$ is a milestone block version with a milestone lifetime δ. The formal definition of a milestone block is given in Section 5.3.3.

- A **file system** with n blocks is defined by the set FS where

$$FS_n = \{B_{(0,L,T)}, B_{(1,L,T)}, ..., B_{(n-1,L,T)}\} \tag{5.4}$$

- The most recent version of a file system of n blocks is thus defined as

$$FS^*_n = \{B^*_{(0,l_0,t_0)}, B^*_{(1,l_1,t_1)}, ..., B^*_{(n-1,l_{n-1},t_{n-1})}\} \tag{5.5}$$

- A **file set** FF is defined by the tuple $FF_{(M,D)}$, where
  - M = the set of all block instances that compose the file's descriptors
  - D = the set of all block instances that compose the file's data

Unlike a file system, the size of a file set is not constant. In particular, the file set grows as new blocks are allotted to the file.

- A **file instance** is a subset of the corresponding file set. For simplicity, we assume that a file's metadata will comprise one or a fraction of one block, while a file's data will comprise an arbitrary number of blocks. A file instance is defined by the tuple $F_{(M_{(i,l_i,t_i)},D,t_f)}$, where
  - t_f is the instance time of the file
  - $M_{(i,l_i,t_i)}$ is the block instance that contains the file's descriptor at time t_f
  - D is the set of all blocks that compose the file's data at time t_f

  It is not necessary that all blocks within the set F share the timestamp value t_f of the file itself. Rather,

  $$\forall B_{(a,l,t)}[(B_{(a,l,t)} \in F_n) \longrightarrow ((t \leq t_f) \wedge \neg \exists B_{(b,l_b,t_b)}[(b = a) \wedge (t < t_b < t_f)])],$$

  or

  $$\forall B_{(a,l,t)}[(B_{(a,l,t)} \in F_n) \longrightarrow (B_{(a,l,t)} \preceq t_f)] \tag{5.6}$$

- When we speak of the size of a file, we refer to the number of data blocks which belong to the file at any given instant. A file F with n data blocks is denoted F_n, where $F_n = (M_{(i,l_i,t_i)}, D_n, t)$. The most recent version of the file is denoted F_n^*.

- Note that a file instance is live if and only if

  $$\forall B_{(a,l,t)}[B_{(a,l,t)} \in F_n \longrightarrow B^*_{(a,l_a,t_a)}] \tag{5.7}$$

5.2 File Reconstruction

Given a time t_f and a file descriptor block address i, we reconstruct the file $F_{(M_{(i,L,T)},D_n,t_f)}$ as follows:

1. We find the file descriptor block $M_{(i,l_i,t_i)}$ which is closest in time to, but earlier than, t_f.
2. From the file descriptor, we determine the set D_n, which enumerates the blocks contained in the file instance F at time t_i. That is,

   $$D_n = \{B_{(0,L_0,T_0)}, B_{(1,L_1,T_1)}, ..., B_{(n-1,L_{n-1},T_{n-1})}\}$$

3. Finally, we collect one instance of each block enumerated by D_n. Each block instance should have a timestamp value which appropriately corresponds to t_i.

As noted in Equation 5.6, not all block instances in D_n will necessarily have timestamp values of t_i. In fact, depending on the file system's write ordering, the timestamp values of the block instances described by the file descriptor block instance $M_{(i,l_i,t_i)}$ may not even be less than t_i—that is, some file systems may write a file's metadata to disk before writing its data to disk. For this reason, choosing appropriately correspondent timestamp values for file reconstruction is not always a trivial process.

We consider two common write ordering models, based on the journaling modes of ext3: *writeback* and *ordered*.

5.2.1 Ordered Models

File systems adhering to an ordered model write a file's data blocks to disk before writing its metadata blocks. This ensures that file metadata will never reference stale data blocks, even in the case of an arbitrary system failure. ext3's *ordered mode* adheres to this model.

Under this model, file reconstruction is simple: to choose data block instances which appropriately correspond to a file's metadata block instance, we merely choose the data block instances which were written immediately prior to the metadata block instance in question. That is, for a reconstructed file $F_{(M_{(i,l_i,t_i)},D_n,t_f)}$

$$B_{(a,l,t)} \in F_n \longrightarrow [B_{(a,l,t)} \preceq M_{(i,l_i,t_i)}] \tag{5.8}$$

This method is guaranteed to reconstruct a consistent file (with respect to the file system, but not necessarily the application), because the presence of a file descriptor block instance in the log indicates that all data block instances to which it refers have already been committed to disk.

5.2.2 Writeback Models

File systems adhering to a writeback model impose no write ordering. After an unexpected system failure, file descriptors may reference data blocks which were not committed to disk before the failure; such files are said to contain stale data. ext3's *writeback mode* and ext2's only mode adhere to this model.

Under this model, file reconstruction is more difficult, because the presence of a file descriptor block instance $M_{(i,l_i,t_i)}$ in the log does not guarantee that all the data block instances to which it refers have timestamp values less than or equal to t_i. Thus under such a model,

$$\neg(B_{(a,l,t)} \in F_{(M_{(i,l_i,t_i)},D_n,t_f)} \longrightarrow [B_{(a,l,t)} \preceq M_{(i,l_i,t_i)}]).$$

In this case, determining which data block instances compose the set D_n is not simple, because some data block instances may be much older than $M_{(i,l_i,t_i)}$ while others might be slightly newer. However, we have developed a heuristic to simplify the task.

Linux kernels provide a dedicated process which is responsible for periodically flushing dirty buffers to the disk to limit data loss upon a system failure. This process, known as *pdflush*, is triggered by the occurrence of one of two events:

1. The number of dirty buffers in the kernel exceeds a predetermined threshold value
2. The flush timer expires

In typical kernel configurations, the flush timer is set to 30 seconds ($t_{ft} = 30$). This sets an upper bound on the extent to which a file's metadata can precede its data in reaching the disk:

$$\forall B_{(a,l,t)} \in F_{(M_{(i,l_i,t_i)},D_n,t_f)}[t \leq (t_i + t_{ft})] \tag{5.9}$$

To reconstruct a file in this case, we take t_i as our reference point. For each file system block address j referenced by the descriptor block $M_{(i,l_i,t_i)}$, there are three possible scenarios:

1. $\neg\exists B_{(a,l_a,t_a)}[(a = j) \wedge (t_a > t_i)]$

   In this case we choose $\mathrm{B}_{(a,l_a,t_a)}$ such that $\mathrm{B} \preceq \mathrm{M}$.

2. $\neg\exists B_{(a,l_a,t_a)}[(a = j) \wedge (t_a < t_i)]$

   In this case we choose $\mathrm{B}_{(a,l_a,t_a)}$ such that $(a = j) \wedge \neg\exists B_{(b,l_b,t_b)}[(b = j) \wedge (t_b < t_a)]$

3. $\exists B_{(a,l_a,t_a)}, B_{(b,l_b,t_b)}[(a = b = j) \wedge (t_a < t_i < t_b)]$

   This case is ambiguous: perhaps block instance $B_{(b,l_b,t_b)}$ belonged to

the file at time t_f but was not committed to disk until *pdflush* was triggered by a timer expiration, or perhaps block instance $B_{(b,l_b,t_b)}$ was not added to the file until some time after t_f.

In some scenarios, file system-specific information can be used to resolve the difficulty. For instance, if $M_{(i,l_i,t_i)}$ contains a generation identifier, we can find the first version of $M_{(i,L,T)}$ with the same generation number. Call this origin block descriptor $M_{(i,l_o,t_o)}$; if there are no data block instances with file system address j existing in the log after $(t_o - t_{ft})$ and before t_i, then we know that all block instances of address j existing before t_i did not belong to file $F_{(M_{(i,l_i,t_i)},D_n,t_f)}$. That is,

$$\forall B_{(a,l,t)}(a = j)[(t < (t_o - t_{ft})) \longrightarrow (B_{(a,l,t)} \notin F_{(M_{(i,l_i,t_i)},D_n,t_f)})]$$

In this case, we can ignore $\mathrm{B}_{(a,l_a,t_a)}$ and choose $B_{(b,l_b,t_b)}$.

If no heuristics can be used to accurately resolve the problem, the reconstruction program must inform the user of the situation. For convenience, the program can present multiple reconstructed versions and allow the user to choose the desired result.

5.3 Log Cleaning

Log cleaning is a critical task and thus must be trusted. In order to improve reliability, the cleaning mechanism should be as simple as possible. We provide three retention policies, *Keep Safe*, *Keep Milestones*, and *Keep Safe + Keep Milestones*, which are enforced by a simple kernel component.

5.3.1 Keep Safe

The Keep Safe policy is quite simple: any file system update must be reversible throughout the established retention interval δ_s.

Given a block instance $B_{(a,l,t)}$, a Keep Safe window δ_s, and a current time t_{cur}, the following suffices as proof that $B_{(a,l,t)}$ is deletable according to the Keep Safe policy:

$$Deletable(B_{(a,l,t)}) \longleftrightarrow (t_{cur} - t > \delta_s) \wedge$$
$$\exists B_{(b,l_b,t_b)}[(a = b) \wedge (t_b > t) \wedge (t_{cur} - t_b > \delta_s)] \tag{5.10}$$

5.3.2 Keep Landmarks

Elephant's Keep Landmarks policy retains versions of files that persist unchanged for a predetermined period of time [47]. Given a file set FF, the Keep Landmarks policy mandates the retention of every block instance referenced by a file instance of FF that persisted unchanged for δ_l seconds. Block instances belonging to FF which last less than δ_l seconds before being overwritten can be deleted. That is,

$$\forall B_{(a,l,t)} \in FF[Deletable(B_{(a,l,t)}) \longleftrightarrow$$

$$\exists B_{(b,l_b,t_b)}[B_{(b,l_b,t_b)} \in FF \wedge (a = b) \wedge (0 < t_b - t < \delta_l)]] \tag{5.11}$$

5.3.3 Keep Milestones

VDisk's deletion proofs must be couched exclusively in terms of blocks. However, as is evident from Equation 5.11, the landmark deletion proof requires some means of expressing file sets and their relationships to block instances. For this reason, we have developed the Keep Milestones policy, which is an approximation of the Keep Landmarks policy.

The goal of the Keep Milestones policy is to preserve all landmark versions of *files*; however, the policy is applied to blocks. The file system layer retention semantics are not quite as simple as the block layer semantics.

For the sake of security, it is acceptable if the Keep Milestones policy mandates the retention of blocks that would have been deletable according to the Keep Landmarks policy. However, the Keep Milestones should not result in the reclamation of any file not reclaimed by the Keep Landmarks policy.

For a file version to be designated a landmark version, the entire file, including its descriptor block, must have persisted unmodified for some period of time δ_m. For a file instance to remain unmodified, all of the data block instances which it comprises must remain unmodified. That is,

$$F_{<\delta_m>(M,D_n,t_f)} \longrightarrow (\forall B_{(a,l,t)} \in F_{<\delta_m>(M,D_n,t_f)}[B_{<\delta_m>(a,l,t)}]) \tag{5.12}$$

Thus to approximate the Keep Landmarks policy, the Keep Milestones policy retains all versions of all milestone blocks.

Naive Keep Milestones

The *Naive Keep Milestones* policy operates under the assumption that blocks are not shared between files. This policy retains all block versions

which persist unchanged for a significant period of time. More specifically, we define a milestone period, δ_m seconds; if any block remains alive for longer than δ_m, it will not be reclaimed during the cleaning process. That is, for all blocks $B_{(a,l_a,t_a)}$ contained in the log,

$$(\neg\exists B_{(b,l_b,t_b)}[(a=b)\wedge(t_a<t_b<t_a+\delta_m)])\longrightarrow B_{<\delta_m>(a,l_a,t_a)} \qquad (5.13)$$

Note that by Equation 5.1, this policy will retain all live versions of all blocks.

Full Keep Milestones

If the assumption that files obtain exclusive ownership of their blocks holds, then the Naive Keep Milestones policy described above will guarantee the retention of all landmark files. However, while this assumption does hold for data blocks in most file systems, it often does not hold for metadata blocks. This poses a problem for the Naive Keep Milestones policy.

Consider two distinct file sets $FF_{1(M_{(a,L,T)},D_1)}$ and $FF_{2(M_{(a,L,T)},D_2)}$. These file sets represent two individual files, but both files have file descriptors located in the same block $M_{(a,L,T)}$.

Assume that a file instance F_1 contained in FF_1 remains unchanged long enough to become a landmark file; thus, the Keep Milestones policy should retain the block $M_{(a,L,T)}$ and all the data block instances belonging to F_1. However, if block $M_{(a,L,T)}$ is frequently updated due to changes in the file set FF_2, it is possible that no milestone version of block $M_{(a,L,T)}$ will exist in the log.

To protect against unwanted deletions, the Naive Keep Milestones policy must be modified. Not only must it keep all blocks that remain unchanged for δ_m seconds; it also must retain the last block instance written within δ_m of the previous retained version of that block. This stipulation is necessary because it is possible that a portion of the block instance belonged to one file and persisted unchanged for δ_m seconds even though other portions of the block, which may have belonged to different files, did not. Thus, for all block instances $B_{(a,l_a,t_a)}$,

$$\neg\exists B_{(b,l_b,t_b)}, B_{(c,l_c,t_c)}[(a=b=c)\wedge(t_b<t_a<t_c)\wedge(t_c-t_b<\delta_m)]\longrightarrow$$
$$B_{<\delta_m>(a,l_a,t_a)} \qquad (5.14)$$

With this policy, any file descriptor which remains unchanged for δ_m seconds will be retained, regardless of how frequently the block in which it

resides is updated. In the case of $FF_{1(M_{(a,L,T)},D_1)}$ and $FF_{2(M_{(a,L,T)},D_2)}$, one version of $M_{(a,L,T)}$ will be retained for every δ_m seconds it is written, regardless of how many times $FF_{2(M_{(a,L,T)},D_2)}$ modifies its metadata. Thus, the inode for $FF_{1(M_{(a,L,T)},D_1)}$, which remained constant for at least δ_m seconds, will be available for file reconstruction.

Finally, as proof that a block instance $B_{(a,l_a,t_a)}$ is deletable according to the full keep milestone policy, the following must hold:

$$Deletable(B_{(a,l_a,t_a)}) \longleftrightarrow$$

$$\exists B_{(b,l_b,t_b)}, B_{(c,l_c,t_c)}[(a = b = c) \wedge (t_b < t_a < t_c) \wedge (t_c - t_b < \delta_m)] \quad (5.15)$$

Chapter 6

Implementation

We have implemented a functional VDisk prototype for the 2.6 Linux kernel. The system comprises three primary components: a logging mechanism, a file reconstruction utility, and a secure log cleaner. Both the logging mechanism and the log cleaner are implemented within a single kernel module, while the reconstruction utility is implemented as a user-space application. The current prototype supports both ext2 and ext3 file systems [1].

6.1 The Logger

The VDisk logging mechanism is implemented as a block-level kernel module. The module obtains exclusive access to two devices, one containing the file system and the other containing the log, and exports a third virtual device to the kernel. This latter is the only device made visible to users, and it exports a standard block layer interface—the logging mechanism itself is completely transparent. This is achieved by creating virtual data structures identical to those of a real disk; in particular, the module registers a standard *request queue* with the kernel, to which all IO *requests* are sent.

6.1.1 The Linux Block Layer

The Linux block layer relies upon a sophisticated mechanism to organize disk accesses in an attempt to mitigate the onerous performance penalties incurred by disk seeks. Rather than immediately performing reads and writes as they are requested by higher level applications, the kernel places these requests on a special per-volume request queue, where customizable algorithms can order and merge these requests to reduce the number of required seeks before sending them to the disk.

Requests contain a linked list of *bios*, which describe the mapping between memory pages and disk sectors. Each bio contains a reference to a block device, a disk address, a size, and a vector of tuples describing pages, offsets and lengths; these pages are mapped onto a contiguous range of disk

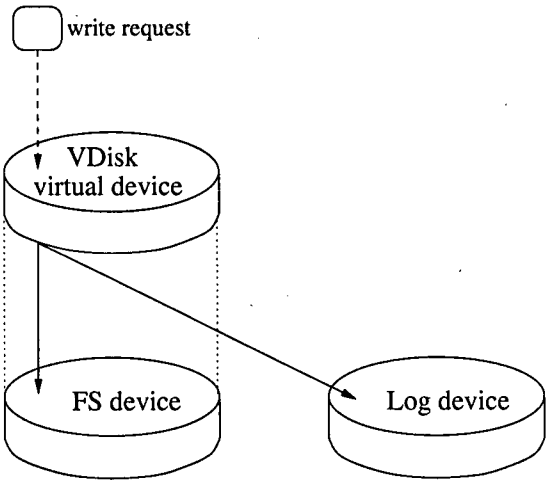

Figure 6.1: The VDisk Device

sectors. Two requests can be merged if the union of their individual maps is a contiguous disk region.

When a request is finally submitted to a disk driver, the driver works through each bio of the request, transferring data as necessary. The driver signals the completion of a data transfer via a reference to a callback function included in the bio structure.

6.1.2 The VDisk Device

Just like other devices, the VDisk logging mechanism maintains its own request queue. Unlike other devices, however, VDisk's request queue does not correspond to a specific device, and requests sent to the VDisk module are not merged or sorted in VDisk's request queue. Rather, IO requests are immediately duplicated, remapped, and redirected to the request queues of both the underlying file system device and the log device. These duplicated requests are then optimized for their particular devices.

VDisk achieves this by duplicating and modifying the bios it receives before they are merged to create requests. The original bios are already mapped to the correct disk sectors of the underlying file system device, and need merely be redirected to that device. The duplicated bios, which are intended for the log, must be remapped to appropriate sector addresses within the log and then redirected to the log device.

The VDisk module contains a simple block allocator that partitions the disk into large segments and allocates blocks consecutively from within these segments, resulting in optimal writes to contiguous disk addresses. When a new bio is to be added to the log, the allocator returns the offset within the current segment, or, if more room is needed, it allocates a new segment and updates the segment bitmap. After a new segment is allocated, the block allocator writes the old segment's reference count to a special segment descriptor at the end of the segment; this count is used later by the secure cleaner to ensure that only empty segments are reclaimed.

A metadata entry is then created, containing the sector address returned by the allocator, the sector address of the original file system device, the time of the write, the size of the write, and a flag used by the secure cleaner to indicate whether the written blocks have been reclaimed. Metadata entries are written to segments just as are data entries; however, segments containing metadata entries never contain data entries, enabling rapid scanning of metadata entries. Metadata segments are joined in a linked list to enable easy scanning of the entire metadata log.

Bios are duplicated in a zero-copy manner; this means that both bio copies share references to common pages. Consequently, these bios must by synchronized with each other—the original bio cannot be returned to the file system, where its pages may be overwritten with new data, until both bios have been successfully committed to their disks. To achieve this, VDisk updates the callback function pointer of each bio to reference its own function. When the two underlying device drivers have completed the requested IO operations, they signal these completions by calling VDisk's callback function; not until both bios have completed does VDisk inform the file system, via the original callback function, that the request is finished.

The VDisk device also maintains a superblock which describes the current state of the log, including details such as the total number of segments, the number of free segments, the address of the first metadata segment, and the addresses of the current data and metadata segments and their offsets.

In the simplest case, adding an entry to the log requires one immediate write: the data. Metadata writes are buffered in memory before being written to disk in batches. If a new segment is allocated, this requires two more writes: both the segment bitmap and the previous segment's reference count must be written to disk. For this reason, VDisk attains optimal performance with large segments—the default segment size is 32MB. Finally, the superblock and the current segment reference count are flushed to disk periodically.

VDisk and Xen

To isolate the logging mechanism, VDisk is placed in its own protected domain with the help of the Xen virtual machine monitor [7]. This is accomplished quite easily: the VDisk device is installed in Domain 0 (the management guest operating system) of the Xen system, and from here it exports its interface to untrusted guest operating systems. These guest operating systems can access the virtual VDisk device just as if it were a standard block device, but they are unable to access the actual disk which contains the VDisk log.

Content Hashing

We have implemented the content-hashing optimization mentioned in Section 4.1.2. To do this, we maintain a small in-memory hash table. To reduce memory requirements, this table can be made arbitrarily small, and thus it may be susceptible to collisions. Whenever inode blocks are read from disk, we create MD5 [42] digests of their contents and store these summaries, along with their sector addresses, in the hash table. When inode blocks are written, we first check for their summaries in the hash table; if their summaries are found to exist—and are identical with summaries of the blocks to be written—the blocks need not be committed to the log. If the summaries are different, the stale summaries in the hash table are replaced by the new, up-to-date summaries of the data to be written to disk.

To get the greatest benefit from this technique, we compare data at the granularity of sectors (512 bytes). This may result in only a few sectors of a large bio needing to be written to disk, meaning that the bio must be split up into multiple smaller bios, with the unnecessary data elided; each of these new bios requires its own entry in the metadata log.

As mentioned in Section 4.1.2, this strategy could pose a security vulnerability if applied to data blocks. For file systems like ext2 and ext3, we are able to distinguish data blocks from inode blocks simply by evaluating their file system block addresses. These file systems allocate a static number of inodes when the disk is formatted, and these inodes are located in predetermined locations on the disk; bios mapped to these locations are known to contain inodes. For file systems with dynamic inode allocation, this strategy will not work.

6.2 The Reconstruction Utility

VDisk's reconstruction utility is implemented as a user-space application and makes liberal use of a MySQL [3] relational database. The database contains a copy of the metadata log; each time the reconstruction utility is used, it compares the database metadata log to the original, and updates the database as necessary. As well, the utility relies upon a file system-specific library to locate and interpret file system objects. Our prototype supports both the ext2 and ext3 file systems [1]; the file system library provides functions to translate sector addresses to file system block numbers, to read superblocks, group descriptors, and inodes, to translate inode numbers to block addresses, and to translate logical file offsets into file system block addresses, among other things. Finally, the utility relies upon yet another library to interface with the log itself; methods for reading and seeking are provided by this library.

6.2.1 The ext2 and ext3 File Systems

The ext2 and ext3 file systems are both intellectual descendants of the Berkeley Fast File System [31]. ext2 and ext3 share nearly identical data structures and disk layout, the primary difference between them being the addition of journaling in ext3. Journaling can either be done on a separate device or in a file, and does not change the structural aspects of the system. It can, however, change semantic properties. In particular, some modes of journaling impose write ordering constraints, requiring that data be written to disk before metadata. Aside from this difference, the file reconstruction algorithms for ext2 and ext3 are identical.

These file systems organize the disk into a number of block groups. Each block group contains a number of metadata blocks at the beginning of the group; this metadata includes: the file system superblock; group descriptors; inode and block bitmaps; an inode table, which contains all the inodes of the group; and the data blocks. Both the superblock and the group descriptors are universal, and are copied in each block group for redundancy; the bitmaps, inode table and data blocks are unique to a given block group.

Both the inode and block bitmaps are limited in size to exactly one block; thus the maximum number of data blocks (and inodes) a block group can contain is equivalent to the number of bits in a single file system block. The inode table is a contiguous array of inodes, and is statically allocated. Inodes are assigned logical inode numbers, and given such an inode number, an inode's location on disk can be determined through a simple calculation.

Every file is identified by a unique inode. The inode contains a tree of pointers to the data blocks which compose the file; typical configurations allocate twelve direct pointers, one indirect, one doubly indirect, and one triply indirect pointer per inode, with indirect pointers referencing indirect blocks, which are arrays of block references.

Directories are represented as files. The data blocks of a directory contain specially formatted directory entries, each of which contains a file's name and inode number, with one directory entry describing each file contained within the directory.

6.2.2 The Reconstruction Algorithm

The reconstruction algorithm begins by determining the file system block address of the root directory's inode. For ext2 and ext3, the inode number of the root directory is always two. After translating this inode number into a block address, the utility queries the database for all metadata entries corresponding to the address and containing appropriate time stamp values. These metadata entries describe the locations of blocks in the log which contain root inodes. These inodes—and their data blocks—are read from the log and scanned in search of the target file or directory. If the target is found, its inode number is converted to a block address and the path resolution continues. If not, the path resolution for this version of the root has failed, and the next version of the root is checked, until a match is found or the time constraints are exceeded.

The most generic reconstruction utility will search all versions of the file system between two given times and return a list of the times at which a target file existed; the user can then choose to reconstruct one or a few of these versions. To compile this list, the utility creates a temporary database table containing the metadata entries that describe the path to each target file; this is done in one pass through the metadata log. An advantage to this approach is that it expedites future searches along the same path, as all the requisite information can be retrieved directly from the temporary table rather than by repeating the path resolution procedure.

To reconstruct a file, the utility translates each logical block address into a file system block address with the help of the file system specific library, and then translates each file system block address into a log block address by querying the database.

6.3 The Cleaner

The VDisk cleaner comprises two components, a user-space tool that does most of the work, and a simple kernel module which enforces the system's deletion policies. The cleaner employs a simple mark and sweep algorithm.

6.3.1 The Segment Analyzer

VDisk's log is cleaned on a per-segment basis. A segment is cleaned by moving any blocks that are not eligible for deletion to a different segment; when all such blocks have been thus transferred, the segment can be reclaimed by the segment allocator and subsequently used to store new data. This process of transferring live blocks can be onerous, and should be avoided as much as possible. The segment analyzer can reduce the impact of cleaning by choosing to clean only those segments that have a small number of live blocks, much as the LFS [44] cleaner endeavors to process only relatively empty segments. However, while the LFS cleaner—one of the more complicated components of LFS—is implemented inside the kernel, the VDisk segment analyzer operates in user space.

When a segment is chosen for cleaning, the segment analyzer compiles a list of proofs for each data entry contained in the segment. A proof contains enough information for the secure log cleaner to easily verify that its corresponding data entry is in fact eligible for deletion according to the various system policies. The segment analyzer compiles this list with the help of the database.

For the Keep Milestones policy, the analyzer creates a list of proofs sorted by file system block address; each proof contains all instances of a given file system block address sorted by time, as well as a reference to two blocks not contained in the segment: the newest version of the block contained in the preceding segment, and the next version of the block address contained in the succeeding segment. If these versions do not exist, special null values are inserted into the proof. This list is then passed to the secure cleaner, which will visit each entry referenced by the list to evaluate its eligibility for deletion and mark it appropriately.

This list is passed to the secure cleaner via the Linux `sysfs` interface, which allows kernel modules to export virtual files that can be read from and written to by user-space applications. When these files are written, the data is not stored on any disk but is instead passed directly to the module. This interface facilitates the transmission of large quantities of data from user space to the kernel, and it is used by VDisk to pass large lists of proofs.

Note that while the secure cleaner will evaluate every item in a proof, it does not scan the actual log itself. This allows for the strategic omission of entries on the part of the segment analyzer. For instance, if more sophisticated retention policies are desired by a user, perhaps stipulating that files with a particular name should never be deleted, the segment analyzer, which operates in user space and understands file system semantics, can ensure that the log entries which compose such files are never included in the deletion lists. By doing so, the segment analyzer can ensure that these entries will never be marked as eligible for deletion and will thus be retained, even if they might have been deletable according to the lower-level block retention policies. However, the converse does not hold. That is, while the segment analyzer can retain blocks which are eligible for deletion, it cannot delete blocks which are not eligible for deletion.

If a kernel is compromised and the segment analyzer is corrupted, these higher-level retention policies cannot be enforced by the secure cleaner; however, the low-level, block-oriented policies such as Keep Safe and Keep Milestones are enforced at all times, regardless of the state of the segment analyzer.

6.3.2 The Secure Cleaner

When the segment analyzer has compiled a list of proofs for a given segment, it passes it to the secure cleaner via the cleaner's special `sysfs` attribute file. The list of proofs contains the locations within the metadata log of all metadata entries that need to be inspected; thus the secure cleaner is spared the task of scanning the log sequentially, as it can jump exactly to the correct location for every inspection. In general, the secure cleaner expects that the list it receives will be valid; it must simply verify that there are no mistakes in the list and mark blocks which are eligible for deletion by setting a bit in their metadata entries.

If during this process the cleaner discovers a discrepancy between the list and the log, it returns an error to the user-space application. For instance, if the list indicates that a metadata entry at a given address should refer to a block address it does not, the secure cleaner fails that proof, ceases processing the list, and returns an error. If the list of proofs is sorted by time—as it should be—an error encountered midway through the list will not affect any of the antecedent proofs. If the list is partially ordered such that the first few proofs are correct but later proofs are incorrectly ordered, the unordered proofs could take one of two forms: they could reference block versions with the same file system block addresses described by the previous,

ordered proofs, or they could reference different blocks. The former case amounts to an omission of blocks from the proof list, which, as discussed in Section 4.3.3, will not result in the loss of milestone information. The latter case concerns blocks which do not belong to the previously validated proofs, and thus will not affect their validity. Consequently, any processing completed before an ordering discrepancy is encountered is still valid and does not need to be undone.

After the all entries of a segment have been inspected, the segment can be cleaned. When instructed to do so, the secure cleaner will scan through a metadata segment, checking the status of each entry. Any entry that is not marked as deletable is moved to a new segment, and the reference count of its original segment is decremented by one. When a data segment's reference count reaches zero, it is known to contain no live data blocks and can thus be reclaimed by the segment allocator.

Chapter 7

Evaluation

Secure versioning primarily consumes two system resources: IO bandwidth and storage space. In addition, the use of Xen to isolate VDisk can also impact the overall performance of a system. VDisk's IO bandwidth consumption is manifested in reduced write throughputs and increased write latencies. Because VDisk duplicates every disk write, it is susceptible to substantial throughput reductions in some usage scenarios. As well, logging writes at the block layer requires significantly more storage than a typical, non-versioning system.

We have conducted a number of experiments with VDisk to empirically quantify the overheads it imposes. To measure IO bandwidth costs, we have used two synthetic benchmarks: Bonnie++ [12], a bandwidth-intensive benchmark, and PostMark [26], a seek-intensive benchmark. By testing VDisk with these benchmarks, we gain insight into the performance degradation incurred by secure logging for various types of IO operations. To measure storage space requirements, we have replayed an NFS trace collected by Daniel Ellard at Harvard [15]. Replaying this trace with VDisk enables us to ascertain the rate at which logged data grows under real-world usage patterns; it also allows us to test VDisk's reclamation policies to determine how much data can be securely deleted.

Finally, we investigated the time required to reconstruct file versions. The work involved in file reconstruction depends upon both the length of the target path and the size of the log. We modified PostMark to create file system images of varying size and depth to evaluate the typical cost of reconstructing a file from a relatively large log.

7.1 Temporal Overhead

VDisk imposes substantial overhead to raw disk writes. This overhead can affect users by degrading application performance. However, given the high costs of disk IO in general, operating systems expend much effort to avoid disk accesses, and these efforts protect users from VDisk's performance

penalties just as they protect users from standard disk performance penalties.

In this section, we measure the performance of VDisk as perceived from the application layer. All experiments were conducted with a 1GHz Pentium III machine with two 320GB IDE disks, one containing an ext2 file system with a 4KB block size and the other containing the VDisk log with a 32MB segment size; each disk provides a raw IO bandwidth of approximately 15 MB/s. Experiments were run on the native machine and in a Xen 3.0 virtual machine; both the native and virtual machines were configured with 512MB of RAM. On the native machine, a standard 2.6 Linux kernel was used, while in Xen, a 2.6 XenoLinux kernel was used.

7.1.1 Bonnie++

Bonnie++ is a benchmark designed to test file system bandwidth. We configured Bonnie++ to operate on a 1GB file. The benchmark consists of five stages: the file is written one character at a time; the file is deleted and then written one block at a time; the file is rewritten one block at a time (note that this requires one read and one write per block); the file is read one character at a time; and finally, the file is read one block at a time. In each of these stages, the file is processed sequentially. To better observe the effects of disk seeks on VDisk's performance, we added one additional stage, in which block writes were performed at random offsets within the file.

Figure 7.1 shows the results of the Bonnie++ benchmark. Each value in the graph is the average of 20 runs; the maximum relative standard deviation across all tests was 3.06%. As is expected, the effect of VDisk on disk reads is negligible; this is because the original file system image is kept intact, and reads are passed directly to the underlying disk without any extra processing. For sequential writes, VDisk halves the system's throughput, as all data is written to disk twice. This is the worst-case scenario for VDisk, but it is only likely to occur in specific cases, such as when a large file is being copied. When working with applications that do not often perform of this type of sustained, sequential write, users will not experience such a substantial degradation in performance.

In the third stage of the test, in which blocks are written randomly, VDisk imposes a 14% reduction in throughput compared to a raw disk. This is due to the fact that, unlike sequential writes, which are throughput-limited, random writes are seek-limited. While writes to the file system are scattered randomly across the disk, writes to VDisk's log are sequential, and thus incur a much smaller performance penalty. The reduction in bus

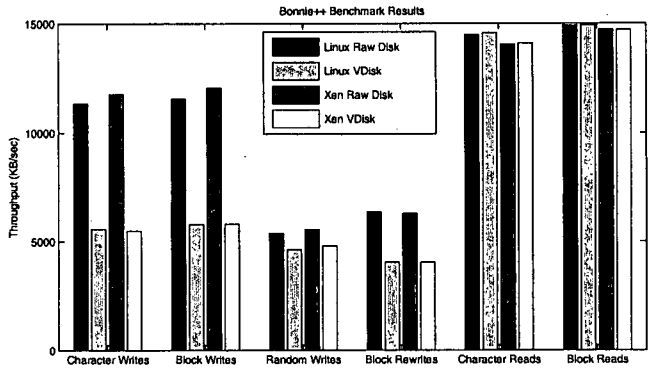

Figure 7.1: Bonnie++ Benchmark Results

bandwidth imposed by VDisk is almost completely overshadowed by the seek overhead of the file system disk, and thus both the raw disk and VDisk perform very similarly in this scenario.

Likewise, the performance of VDisk in the fourth stage of the test, in which blocks are rewritten, is closer to that of the raw disk. In this stage, VDisk only incurs a performance penalty for half of the disk operations—the writes—while both systems perform almost identically for the reads. Thus the user-perceived throughput overhead of VDisk in this case is only 37%, rather than the 51% and 50% overheads incurred during the sequential character and block writes, respectively.

It is interesting to observe that the impact of the Xen virtual machine monitor is minimal, and in the case of sequential writes, even improves performance. This improvement is an artifact of Xen's strategy of sacrificing latency for throughput. In a Xen virtual machine, write requests are slightly delayed in the guest kernel in order to improve batching; when they finally reach the underlying device, they are often much larger than their native kernel counterparts, and can thus improve performance.

7.1.2 PostMark

PostMark was designed by Network Appliances to simulate an email server. To do this, PostMark creates a large number of text files of various sizes and performs a large number of IO transactions on these files. More specifically, during the transaction stage, PostMark either reads from, appends to, or deletes an existing file, or creates a new file. Whereas Bonnie++ performs a stress-test of the IO system's throughput by preforming sequential reads and writes, PostMark mimics slightly more realistic usage patterns by randomly accessing a large number of files, and is thus a better measure of a system's seek performance. We configured PostMark to create 20,000 files between 0.5KB and 10KB in size and perform 50,000 transactions with an even read/write ratio, resulting in about 270MB of file system data being written to disk.

Figures 7.2 and 7.3 display the averaged results of twenty PostMark trials. The maximum relative standard deviation for all trials was 1.44%. As can be seen in Figure 7.2, VDisk performs similarly to the raw disk in this benchmark, imposing a 9.7% increase in overall time in a native Linux environment. As in the random block write stage of Bonnie++, the PostMark benchmark is seek-limited, and the time required to seek in the original file system disk almost completely overshadows the time needed to perform the extra sequential writes to the VDisk log. This is further evinced by Figure 7.3, which shows that VDisk imposes a 10% overhead in write throughput. Notice that the write throughput achieved by PostMark with a raw disk is just 1.5MB per second, as opposed to the 12MB per second achieved by Bonnie++.

As with Bonnie++, Xen imposes very little overhead in the PostMark benchmark. While a raw disk in native Linux slightly outperforms a raw disk in a Xen guest domain, VDisk actually performs better in the guest domain than it does in native Linux. The throughput achieved with VDisk in Xen is higher than that achieved in native Linux, again due to Xen's strategy of batching writes in the guest domain before submitting them to the disk.

7.1.3 Discussion

VDisk's strategy of duplicating all disk writes imposes a substantial overhead to throughput-limited disk operations. Bonnie++ verifies the expectation that performing large, sequential writes with VDisk will take approximately twice as long as doing so with a raw disk. However, as can be seen in the

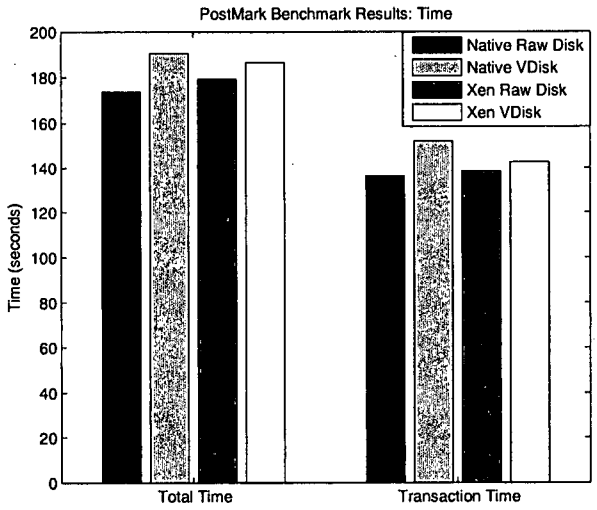

Figure 7.2: PostMark: Time

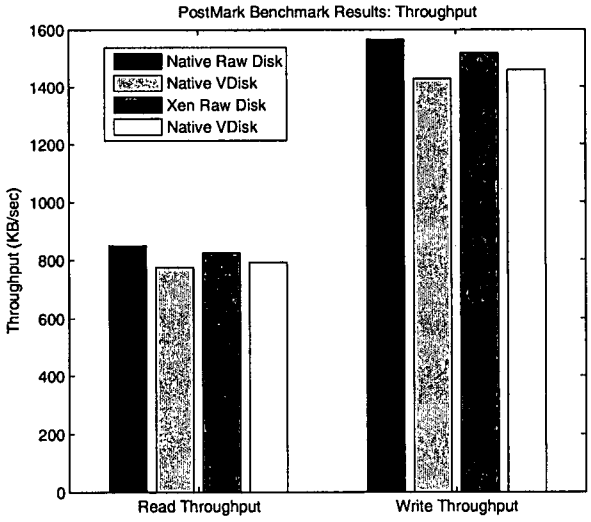

Figure 7.3: PostMark: Throughput

random write test of Bonnie++ and the PostMark tests, VDisk has a much smaller impact on seek-limited disk writes.

The degree to which VDisk will degrade performance thus hinges upon the frequency with which large, sequential writes are performed under typical workloads. A number of studies have been conducted with the intent of investigating file access patterns, and the general consensus of these efforts is that most file accesses are made to small files [6, 22, 36].

Baker et al. found that the majority of sequential file transactions in their traces were small, with about 80% of these runs transferring less than 10KB [6]. However, the size of the larger files in this trace were so large that at least 10% of all bytes were transferred in sequential runs larger than 1MB. Note that in these traces, sequentiality is determined with respect to logical file block numbers rather than physical block addresses.

This study also found that while most file accesses were made to small files, most bytes were transferred to or from larger files. In a more recent study, Vogels corroborates this observation, finding that most sequentially transferred bytes belong to files greater than 10KB in size [57]. In Vogels' study, the relative size of large files increased by an order of magnitude over those in Baker's study, growing to 100–300MB for scientific computing workloads.

Most large files in Vogel's study were system files such as executable binaries, dynamic loadable libraries, and font files; typically, larger files were not read and written in their entirety, but were instead accessed in small chunks at a time, often via memory-mapped IO. Similar access patterns for large files were observed by Roselli [43], who found that the majority of bytes in files larger than 100KB were accessed randomly. Moreover, in Roselli's NT workload, about 60% of bytes accessed were done so randomly. This number is similar to results found in an analysis of the EECS trace [16], although it is substantially higher than the other workloads analyzed by Roselli.

Modern text and image editors typically update files by deleting old versions and replacing them with new versions. This results in a large number of file deletions and creations [22], with the latter typically involving sequential writes to logical block numbers. Again, the size of most created files is small [22], meaning that most creations do not entail large sequential writes. Gibson and Miller found that while 25% of modifications in their traces were made to files that were larger than 64KB, most file modifications increased file sizes by less than 1KB at a time [22].

These studies indicate that most of the time, users interact with small files. Document editing applications often rewrite previous versions of files

with newer versions, resulting in a high number of deletions and creations of small files. Larger files often account for the majority of bytes transferred, but these files are typically accessed piecemeal in a random fashion, and, as in the case of executables and libraries, are often opened for reads only. These trends suggest that for many—if not most—scenarios in which users interact with the file system, VDisk will impose overhead similar to that exhibited by the PostMark benchmark.

7.2 Spatial Overhead

To gain insight into the rate at which VDisk's log will grow and the efficacy of our reclamation policies, we have made use of the EECS NFS trace [15]. The EECS trace collected NFS usage statistics of the primary home directory server of Harvard's computer science faculty and research groups. This server was used for research, software development and course work, storing home directories and shared project and data files. The aggregate capacity of users' local disks exceeded the capacity of the server, and its role was primarily one of facilitating sharing across multiple accounts and preserving backups. The trace contained 317 unique user IDs.

To replay the trace, we converted the logged NFS commands into local file system operations, which were then executed synchronously on a local disk with an ext2 file system with a block size of 4KB. We replayed the first 57 days of this approximately eleven week trace.

7.2.1 Log Growth

Figure 7.4 displays the daily growth of both the file system and the log during the trace; the figure also includes the size of the log when all versions of the superblock and group descriptors are omitted. While the file system grew to just over 50GB during the trace, at an average of about 1GB per day, the block log exploded to nearly nine times that size, at an average of almost 8GB per day. Figure 7.5 shows the growth of the log and file system normalized to the 317 unique `uid` values contained in the trace.

Metadata blocks constitute a large portion of the block log. Many important file system blocks, such as the superblock and group descriptor blocks, are flushed to disk upon any file system state changes, consuming a substantial portion of the block log. As Table 7.1 shows, over one quarter of the log is composed of versions of file system blocks zero through four, out of a total of 14,801,516 distinct file system block addresses written during the trace.

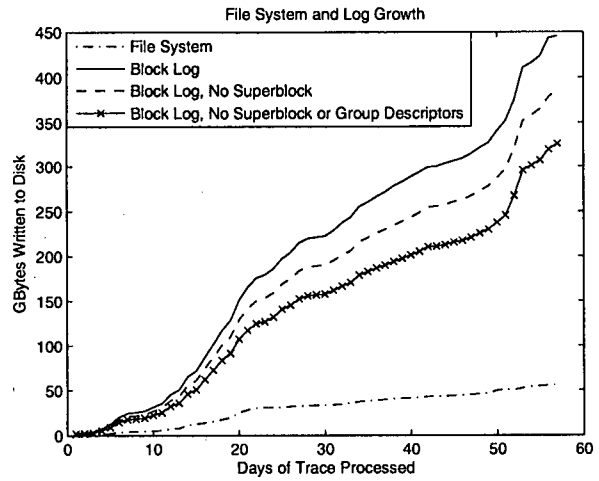

Figure 7.4: Total Log Growth

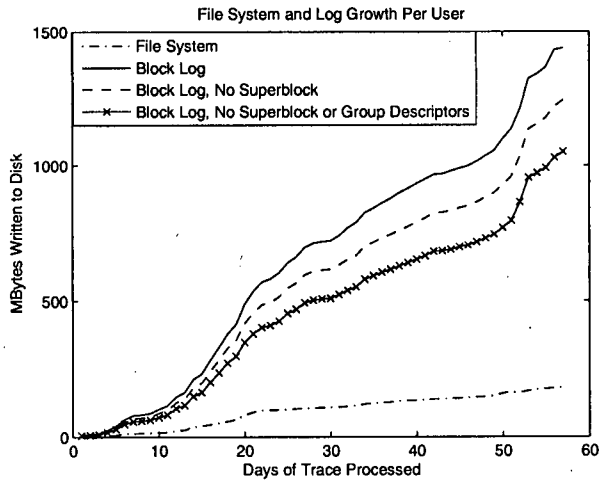

Figure 7.5: Log Growth Per User

Table 7.1: The Most Frequently Written Block Addresses

FS Block Address	Number of Versions	Percentage of Log
0	15,606,469	13.36%
2	4,823,655	4.13%
1	4,331,214	3.71%
4	3,931,097	3.37%
3	2,664,172	2.28%
886,018	225,614	0.19%
6,291,456	181,009	0.15%

However, while versions of these few file system block addresses make up a large proportion of the log, the majority of block addresses were written to only once during the trace. Figure 7.6 displays the number of writes per file system block address: just over 70% of all file system block addresses were written once, and over 95% were written less than ten times.

This evidence suggests that filtering strategies for reducing the log size should be targeted towards the few select file system block addresses with high write frequencies, where significant improvements can be realized, but that the majority of block addresses are not amenable to such strategies because they are written so rarely.

7.2.2 Content Hashing

Figure 7.7 illustrates the efficacy of content hashing in reducing log growth. This plot displays the growth of the the log during the first day of the trace. By the end of the day, 4,630,712 blocks had been written to the log. We re-ran this one day trace with two hashing policies: the first hashed only inode sectors, while the second hashed both inode sectors and the sectors composing file system blocks zero through four, which contain the superblock and group descriptors.

With both policies, we used a hash table of 256KB to store 128-bit MD5 content summaries of targeted sectors as they were read from disk. When hashing only inodes, there were 492,304 writes to targeted sector addresses, of which 450,417—or 91%—were identical to their hashed counterparts and were not written to the log. When hashing file system blocks zero through four as well as inode blocks, there were 2,061,016 writes to target sector addresses, of which 1,899,042—or 92%—were identical to their hashed counterparts.

Hashing just inode blocks resulted in a modest log size reduction of about 10% at the end of one day, while hashing the superblock and group

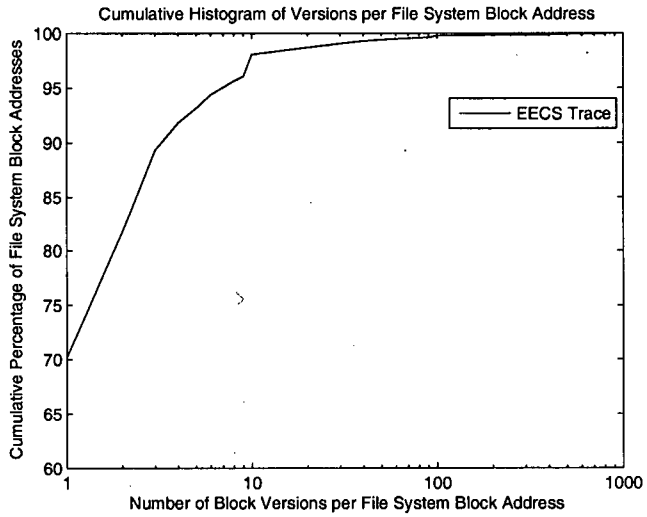

Figure 7.6: Versions Per Block Address

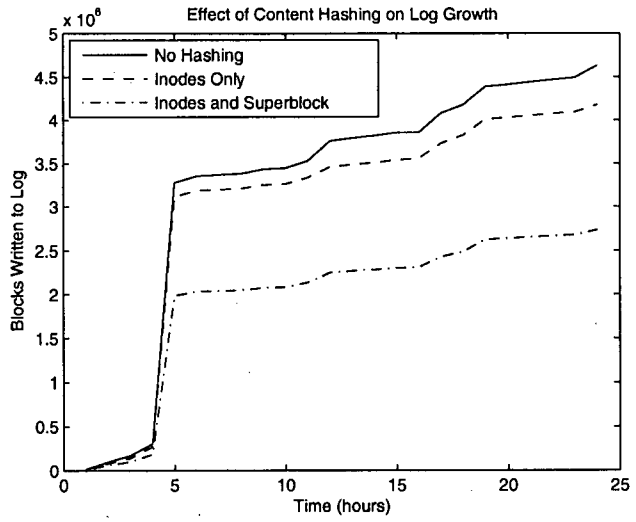

Figure 7.7: Effect of Content Hashing on Log Growth

descriptors along with inode blocks resulted in a more substantial reduction of 41%. Content hashing was particularly effective in this experiment because writes were performed synchronously, resulting in a large number of metadata blocks being flushed to disk with minimal changes. Further gains could be obtained with the use of delta-encoding to store these small changes in units more compact than sectors (512 bytes).

7.2.3 Block Reclamation

VDisk's block retention policies are based upon block lifetimes: according to the Keep Milestones policy, block versions that are overwritten in a short period of time are eligible for reclamation. Thus to evaluate the efficacy of this policy in reducing log size, it is instructive to investigate the distribution of block lifetimes.

Block Lifetimes

Figure 7.8 presents the block lifetimes we observed when replaying the EECS trace. Almost 82% of blocks logged during this trace were overwritten in under one second. This number is higher than that reported in an analysis of five days of the same trace in [16] because our replay of the trace includes block writes required for file system metadata updates that are not explicitly included in the NFS log (versions of block addresses zero through four alone constitute 11% of blocks overwritten in one second). Moreover, it is significantly higher than the 20% reported in Roselli's analysis of different traces [43]. However, both our trace replay and the analysis in [16] show that 90% or more of blocks in the EECS trace are overwritten within an hour, while between 70% and 80% of blocks in four out of five of the traces analyzed by Roselli are overwritten within an hour. This suggests that with even a very small milestone window, a large portion of VDisk's log can be reclaimed by the secure cleaner.

Landmark Retention Policy

In order to evaluate the efficacy of a file system-aware retention policy, we applied a simulated landmark policy to every file of the NFS trace. The simulation only processed data blocks belonging to files; it did not account for any file system metadata blocks.

Figure 7.9 displays the number of blocks retained by the landmark policy with landmark windows of one hour, one week, four weeks, and nine weeks. The nine week case illustrates the optimal scenario for the landmark policy;

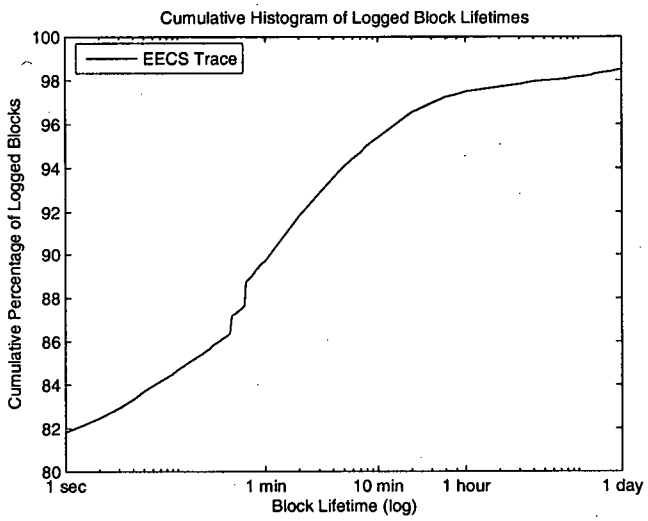

Figure 7.8: Block Lifetimes

with this landmark window, only files that remained alive throughout the entire trace are retained. With a one hour landmark window, 62% of data blocks can be reclaimed; this number increases to 64% with a landmark window of nine weeks.

Milestone Retention Policy

Figure 7.9 displays the results of applying the milestone retention policy to our log with various milestone windows. The majority of blocks that were retained were done so because they constituted either the first or last version of their file system block addresses, and under the milestone policy, these versions must be retained regardless of their lifetimes. This is not surprising, because, as we have already seen from Figure 7.6, 70% of block addresses were written to only once during the trace. Using a milestone window of nine weeks illustrates an extreme case in which all retained blocks are either the first or last versions of their file system block addresses, because the entire trace fits within the window. This case exhibits the maximum number of deletable blocks for the trace.

Using a milestone window of just one hour, 81% of logged blocks can be reclaimed. Using the best-case milestone window of nine weeks, 84%

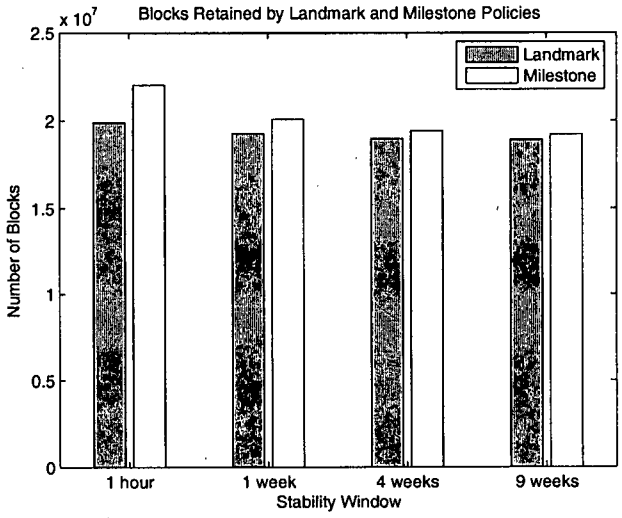

Figure 7.9: Version Cleaning

of logged blocks can be reclaimed. These results are consistent with the general characteristics of the workload: typically, blocks are either rewritten immediately or they are written less than three times during the entire trace.

Discussion

In general, both the milestone and the landmark retention policies allow for the reclamation of a large proportion of versioned data. Due to the high number of blocks with short lifetimes, this is not surprising. Although the milestone retention policy exhibits a higher reclamation percentage than the landmark policy, this is primarily due to the huge number of file system metadata blocks, i.e., blocks zero through four, written during the trace replay.

It is interesting to observe that both policies retain nearly the same number of blocks. With a one hour window, the milestone policy retains approximately 10% more blocks than the landmark policy, and this number decreases to just 2.15% with a four week window. Although our simulation of the landmark policy did not account for file system metadata blocks, we expect the overhead required for these blocks will be quite low, as the average number of retained file versions with a landmark window of one hour

is 1.11%.

However, it should be noted that our simulation of the landmark policy did not exploit one of the key capabilities of the Elephant file system, which is the support for per-file-group policies. Had we restricted our landmark simulation from versioning certain types of files, such as executable binaries and HTML objects, the number of blocks retained by the landmark policy would have been smaller. Thus our analysis represents an upper bound of the number of blocks retained by the landmark policy.

While the milestone policy is obliged to be more conservative than the landmark policy in order to ensure the preservation of landmark files, the bimodal write patterns exhibited by the EECS workload mitigate the penalties incurred by this conservatism. The milestone policy mandates that both the first and the last version of any block address must be retained, but the majority of block addresses retained by both policies were only written to once, resulting in similar performance for both policies. For the same reason, both policies are nearly as effective with a stability window of one hour as they are with a window of nine weeks. The vast majority of blocks reclaimed by either policy are overwritten in under one hour, and blocks that survive this initial hour tend to live for a long time.

Figure 7.10 shows the rate at which VDisk's log would grow if it was cleaned daily according to the landmark and milestone policies with a stability window of one hour. Both the landmark and milestone policies dramatically curtail log growth, keeping the log size to within just 36% and 50% of the original file system size, respectively.

7.3 File Reconstruction

The time required to reconstruct a file version is dependent upon a number of variables. Some of these variables relate to file system properties, such as the depth of the target file's path, the number of files contained in each directory along that path, the number of versions of each file system object visited, and the period of time over which these versions existed. Other variables relate to block layer properties, such as the size of the log and the number of versions of each block address of interest. Due to the variability of the work involved in reconstructing a file, it is difficult to perform a universal evaluation of the reconstruction utility. However, some performance characteristics can provide a general insight into the time required to reconstruct versions.

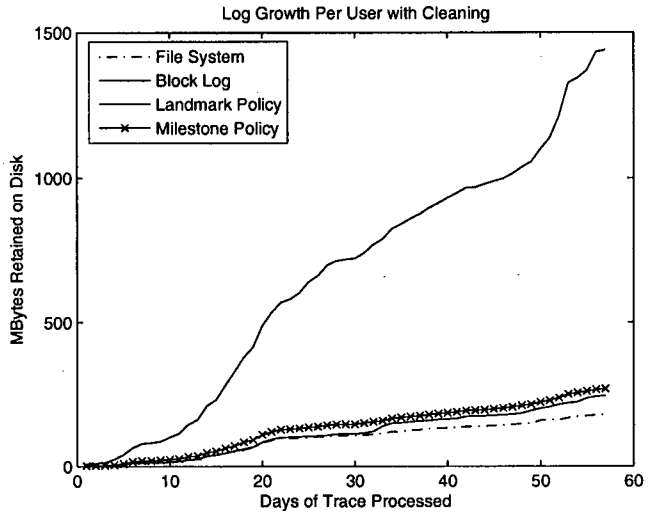

Figure 7.10: Log Growth With Cleaning

7.3.1 General Performance Characteristics

VDisk's reconstruction utilities support two tasks: reconstructing a single file version at a specified time, and finding all versions of a file given a specified name and time range. The first task is straightforward: the newest version of each directory along the path is searched, until the path is fully resolved and the target file's inode number is obtained, or the resolution fails. Thus, resolving one version of a path requires at least two metadata database queries for each directory along the target path: one to translate the file system block address of the directory's inode into its corresponding log block address, and one to perform similar translations for each of the directory's data blocks. Likewise, searching a directory requires at least two log reads: one to read the directory's inode, and one for each of its data blocks. The time required to resolve a single path version is dependent predominantly upon the size of the metadata log, the length of the path, and the average size of each directory along the path.

The second task, finding all versions of a file name, is the most taxing task performed by the file reconstruction utility. Visiting each directory along the path to be searched requires searching through each version of that directory between the given time boundaries. Clearly, the longer the

target path, the more work required to reconstruct a file.

In addition to the depth of the target path, another key variable of this task is the average size of each directory visited. Obviously, the more entries contained in a directory, the more time required to perform a linear search of the directory. However, the versioning of directories presents yet another dimension of overhead: any time a new file is added or removed from a directory, a new version of the directory is created; this new version must also be searched during the path resolution process. Updating a directory can not only increase the time required to search the directory, it increases the number of versions of the directory which must be searched.

Providing time constraints to the reconstruction utility can often dramatically reduce the time required to reconstruct a file, as these time constraints can reduce the number of versions of path directories that need to be searched.

The slowest single operation performed during file reconstruction is querying the metadata database. This is especially true when the database is large. A new query must be performed any time a new file system block address is encountered during the path resolution process. However, an optimization exists for the file reconstruction process: once a target inode is found, we generate a temporary table containing the file system block addresses of each block in the file. We then take the cross product of this table with the metadata table, limiting the results to the newest block versions that are older than the inode. This technique allows us to translate all of a file's file system block addresses into log block addresses with a single query.

7.3.2 Specific Performance Profiling

We used a modified version of the PostMark benchmark to quantify the time required to reconstruct file versions under a few specific scenarios. We forced PostMark to perform all writes synchronously, thus ensuring that every file version accessed in memory was committed to disk. We configured PostMark to write to 10,000 files distributed evenly across 50 directories, with a maximum path depth of 10. This produced about 5,500,000 distinct file versions, committing 45.07GB of data to the log.

Path Resolution

The path resolution process can be greatly expedited by a few well-chosen indexes. For instance, a user-space tool could periodically traverse recently logged file system versions, recording path names and inode numbers for

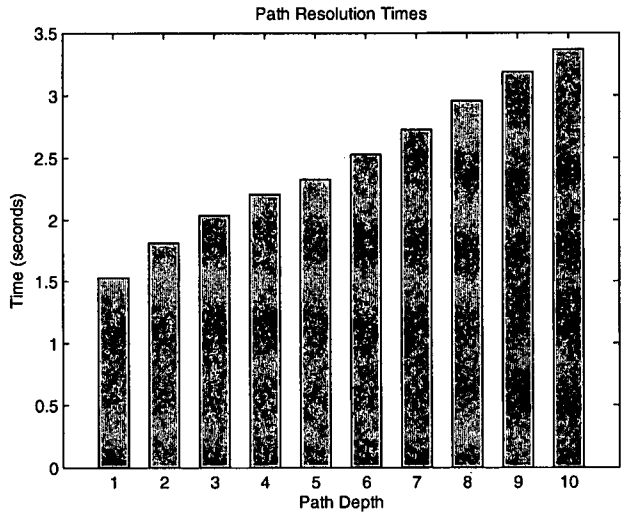

Figure 7.11: Path Resolution

each file and directory it encounters along the way; this index could completely obviate the path resolution process during later reconstruction requests. While we have not implemented this inode index, we do rely upon two MySQL indexes—keyed on target-disk sector IDs and timestamps—to speed the path resolution process; building these indexes for this experiment required 115.56 seconds and 197.04 seconds, respectively, and the combined size of these indexes was 206.7MB.

Figure 7.11 shows the times required to resolve paths of various depths from the log, given the approximate times at which each target version existed. The time required to reconstruct every version of a target path was substantially longer, ranging from 88.33 seconds for a one-directory path to 989.33 for a ten-directory path.

File Reconstruction

Once a file's inumber is known, the process of reconstructing a file version is simple: the file system address of each block contained in the file (including the inode and indirect pointer blocks) must be mapped to its corresponding log address via the metadata database.

A naive implementation of this translation process could query the database

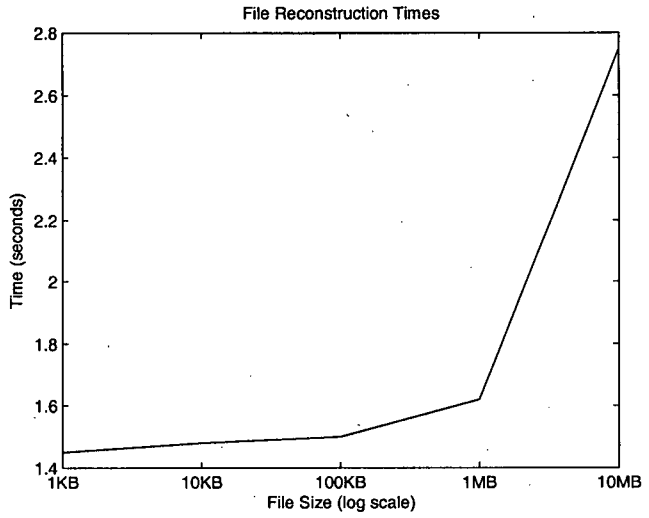

Figure 7.12: File Reconstruction

once for each block in the file. Such an approach would yield reconstruction times scaling linearly in proportion to the number of required queries. The power of user-space indexing techniques is manifested when a more sophisticated translation approach is adopted. By using the cross product optimization mentioned above, all data block translations can be achieved in a single database query. Figure 7.12 shows the times required to reconstruct file versions of various sizes, ranging from 1KB to 10MB. As the figure shows, the cross product optimization results in low reconstruction times even for large files.

7.4 Summary

Because VDisk duplicates all disk writes, bus bandwidth is reduced by 50% compared to a standard disk. For bandwidth-limited applications, this redounds to a 50% reduction in performance. However, many file system operations are seek-limited, and the delays associated with these seeks can almost completely overshadow the reduction in bandwidth imposed by VDisk, resulting in performance degradations closer to 10% or 15%.

In addition to the cost of duplicating file system writes, VDisk's cleaner imposes overhead when reclaiming segments. In the case of the EECS trace,

daily cleaning of the log with a milestone window of one hour results in the need to relocate 4.72MB of data per user per day. The amount of data that must be relocated depends upon the efficacy of the retention policy; for this trace the milestone policy was particularly effective at reclaiming blocks, allowing for the reclamation of more than 80% of the log and resulting the need to relocate the other 20% of the log during the cleaning process.

Versions of the superblock and group descriptors accounted for more than 25% of the log. As these blocks are not necessary for version reconstruction, they can be safely filtered during logging, reducing both the storage space and write bandwidth consumed by VDisk.

While comprehensive logging resulted in a log that was nearly nine times larger than the file system, filtering and cleaning drastically reduce the size of the log. By employing these techniques, VDisk was able to limit the size of the log to roughly 1.5 times the size of the file system during the two month trace. During these two months, the file system grew by about 3MB per user per day, while the log, when omitting superblock and group descriptor versions, grew by an average of 18.46MB per user per day, with a maximum daily increase of 91.59MB. Thus, for workloads similar to the EECS trace, users should expect a regularly cleaned log to consume 125% per month of the disk space consumed by the file system, and at least 100MB of free space should be reserved to accommodate daily log growth in between cleanings.

Chapter 8

Future Work and Conclusion

8.1 Future Work

A key issue in implementing secure logging is minimizing storage requirements. Comprehensive logging at the block level can result in log size explosions, thereby reducing the system's version retention window. An important area of future research is thus exploring techniques to reduce VDisk's log growth. Additionally, useful new features, such as version content indexing, are viable additions to VDisk due to its stratified design.

8.1.1 Block Delta-Chaining

Recent work in RAID-based block versioning has uncovered a highly effective means of limiting block log sizes [63]. By storing block versions as run-length encoded delta chains, the TRAP versioning system has managed to dramatically reduce storage requirements. With some modifications, VDisk could make use of this technology. The reduced log size would come with an increased latency in accessing file histories because block versions could no longer be read directly but would instead have to be reconstructed. However, if versions are not accessed frequently, this increased latency could be more than acceptable for a significantly reduced log size. An ideal compromise might be to retain full versions of file system metadata, such as inode blocks, to enable rapid browsing of versioned directories, while storing data blocks as delta-chains. Additionally, VDisk's reclamation scheme would need modifications before it could accommodate delta-chaining: deleting intermediary block versions in a delta chain would require reconstituting and retaining the full version of the block existing at the end of the deleted interval to preserve the accessibility of newer versions of the block.

8.1.2 Templated Retention Policies

As mentioned in Section 4.3.1, one method of communicating a block's eligibility for deletion would be to use a domain-specific language. File systems

could register a set of templates with VDisk's secure cleaner; these templates could be used by VDisk to extract important higher level information from blocks while still remaining file system agnostic. With this information, powerful new retention policies could be implemented, enabling the secure cleaner to recycle a greater number of blocks, as well as potentially allowing the enforcement per-file retention policies from the block layer.

8.1.3 Version Content Indexing

VDisk's log-based structure lends itself quite nicely to a version indexing scheme: whereas in typical file systems, periodic indexing must be applied to various updated files scattered across a file system, with VDisk it could simply be applied to the tail of the log. By indexing file histories, VDisk could provide users with a means of rapidly searching old versions. This capability could greatly facilitate the process of recovering older files. And thanks to VDisk's stratified design, this version indexer could be implemented entirely in user space.

8.2 Conclusion

By retaining a history of file system operations, versioning systems protect users against incidental destruction of data. Versioning systems can come in a number of forms, including user-space applications, stackable and on-disk file systems, and block-level systems. Because of their relative simplicity, we maintain that block-level versioning systems can provide a higher degree of reliability and security than can file systems, and are thus better suited to protect important information.

Due to an ignorance of file system semantics, typical block-level versioning systems only provide coarse-grain versioning. To remedy this, we present VDisk, a secure, block-level versioning system that is able to provide fine-grain, per-file versioning. The most notable aspect of VDisk is its stratified design: critical code which has write-access to versioned data is implemented in a simple, protected module, while more complicated code is implemented in user space, where it cannot endanger versioned data. This design greatly reduces VDisk's trusted computing base.

VDisk comprises three components: a secure logger, an untrusted file reconstruction utility, and a secure cleaner. Both the logger and the cleaner are protected from malicious agents by a virtual machine, meaning that VDisk can protect versioned data even if the operating system it services is

compromised. The extreme simplicity of these protected components lends reliability to the system.

VDisk provides per-file version reconstruction with the help of a user-space utility. With read-only access to VDisk's log, this utility is capable of reconstructing individual file versions. Additionally, an untrusted user-space application is responsible for initiating the reclamation of unneeded logged blocks. This application searches the log for segments which will benefit from cleaning, and constructs proofs evincing each block's eligibility for deletion. Proofs must be verified by the secure cleaner before the blocks they describe can be deleted; this allows the cleaner to enforce retention policies which prevent the destruction of important information.

While the cost of duplicating all disk writes reduces data bus throughput by 50%, the typical user-perceived overhead of VDisk is closer to 15%. Many applications are seek-bound rather than throughput-bound; while file system disks are slowed by performing many seeks, VDisk's disk minimizes seek overhead by writing data contiguously. Consequently, in many cases, VDisk's overhead is almost completely overshadowed by the file system seeks. Additionally, we found that with the EECS workload, content hashing can reduce log growth by up to 40%, while VDisk's retention policies can reclaim nearly 80% of data written to disk while still preserving important landmark files.

Bibliography

[1] The ext2/ext3 file system. http://e2fsprogs.sf.net.

[2] The IBM journaling file system for linux. http://www-124.ibm.com/jfs.

[3] MySQL. http://www.mysql.com.

[4] ReiserFS. http://www.namesys.com.

[5] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Symposium on Operating Systems Principles*, pages 43–56, 2001.

[6] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 198–212. ACM Press, 1991.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[8] Aaron B. Brown and David A. Patterson. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference*, pages 1–14, 2003.

[9] Kerstin Buchacker and Volkmar Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Third IEEE International High-Assurance Systems Engineering Symposium*, pages 95–105, 2001.

[10] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

[11] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode File System. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, San Fransisco, CA, USA, 1992.

[12] Russel Coker. The bonnie++ home page. http://www.coker.com.au/bonnie++.

[13] Brian Cornell, Peter Dinda, and Fabian Bustamante. Wayback: A user-level versioning file system for linux. In *USENIX Annual Technical Conference, FREENIX Track*, 2004.

[14] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[15] Daniel Ellard. Trace-based analyses and optimizations for network storage servers. PhD thesis, Harvard Computer Science Technical Report TR-11-04, May 2004.

[16] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive nfs tracing of email and research workloads. In *Second Annual USENIX File and Storage Technologies Conference (FAST'03)*, pages 203–216, March 2003.

[17] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[18] Per Cederqvist et al. *Version Management with CVS*. Network Theory Limited, 2002.

[19] Michail D. Flouris and Angelos Bilas. Clotho: Transparent data versioning at the block I/O level. *12th NASA/IEEE Conference on Mass Storage Systems and Technologies*, 2004.

[20] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the xen virtual machine monitor. In *The First Workshop on Operating System and Architectural Support for the On Demand IT Infrastructure (OASIS-2004)*, October 2004.

[21] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 193–206, 2003.

[22] Timothy J. Gibson and Ethan L. Miller. Long-term file activity patterns in a unix workstation environment. In *The Fifteenth IEEE Symposium on Mass Storage Systems*, March 1998.

[23] David K. Gifford, Roger M. Needham, and Michael D. Schroeder. The cedar file system. *Communications of the ACM*, 31(3):288–298, 1988.

[24] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994.

[25] James E. Johnson and William A. Laing. Overview of the Spiralog File System. *Digital Technical Journal of Digital Equipment Corporation*, 8(2):5–14, 1996.

[26] Jefferey Katcher. Postmark: A new file system benchmark. Technical Report TR3022. Network Appliance Inc, October 1997.

[27] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Symposium on Operating Systems Principles*, pages 223–236, 2003.

[28] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *The 2005 Annual USENIX Technical Conference*, April 2005.

[29] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

[30] Josh MacDonald, Paul N. Hilfinger, and Luigi Semenzato. PRCS: The project revision control system. *Lecture Notes in Computer Science*, 1439:33+, 1998.

[31] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[32] Charles B. Morrey III and Dirk Grunwald. Peabody: The time travelling disk. In *IEEE Symposium on Mass Storage Systems*, pages 241–253, 2003.

[33] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, 1986.

[34] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *Third USENIX Conference on File and Storage Technologies (FAST 2004)*, San Francisco, CA, USA, March/April 2004. USENIX Association.

[35] Brendan Murphy and Bjorn Levidow. Windows 2000 dependability. In *IEEE International Conference on Dependable Systems and Networks*, June 2000.

[36] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 15–24. ACM Press, 1985.

[37] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, New York, NY, USA, 1988. ACM Press.

[38] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: File-system-based asynchronous mirroring for disaster recovery. In *First USENIX Conference on File and Storage Technologies*, 2002.

[39] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[40] Zachary Peterson, Randal Burns, Joe Herring, Adam Stubblefield, and Aviel Rubin. Secure deletion for a versioning file system. In *The Fourth USENIX Conference on File and Storage Technologies*. USENIX Association, December 2005.

[41] Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey. Plan 9: A distributed system. In *Sprint 1991 EurOpen*, May 1991.

[42] Ron Rivest. The MD5 message-digest algorithm. IETF RFC 1321, april 1992.

[43] Drew Roselli, Jacob Lorch, and Thomas Anderson. A comparison of file system workloads. In *USENIX 2000 Technical Conference*, pages 41–54, 2000.

[44] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[45] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov 1984.

[46] Russel Sandberg, David Boldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Summer USENIX Conference*, pages 119–130, June 1985.

[47] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant File System. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[48] Margo I. Seltzer, Keith Bostic, Marshall K. McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[49] Margo I. Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata N. Padmanabhan. File system logging versus clustering: A performance comparison. In *USENIX Winter*, pages 249–264, 1995.

[50] M. Sivathanu, V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 15–30, March 2004.

[51] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or death at the block-level.

In *Symposium on Operating Systems Design and Implementation*, pages 379–394, 2004.

[52] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *Second USENIX Conference on File and Storage Technologies (FAST 2003)*, March 2003.

[53] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Greg Ganger. Metadata efficiency in versioning file systems. In *Second USENIX Conference on File and Storage Technologies (FAST 2003)*, San Francisco, CA, USA, 2003.

[54] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules, and Gregory R. Ganger. Design and implementation of a self-securing storage device. In *Symposium on Operating Systems Design and Implementation*, pages 165–179, October 2000.

[55] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *19th ACM Symposium on Operating Systems Principles*, October 2003.

[56] Walter F. Tichy. RCS: A system for version control. *Software—Practice and Experience*, 15(7):637–654, 1985.

[57] Werner Vogels. File system usage in windows nt 4.0. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 93–109. ACM Press, 1999.

[58] Andrew Warfield, Steven Hand, Ken Fraser, and Tim Deegan. Facilitating the development of soft devices. *USENIX Annual Technical Conference, General Track*, pages 379–382, 2005.

[59] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing storage for a million machines. In *The 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X)*, June 2005.

[60] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Using time travel to diagnose computer problems. In *Symposium on Operating Systems Design and Implementation*, pages 77–90, 2004.

[61] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *First Symposium on Networked Systems Design and Implementation*, pages 169–182, 2004.

[62] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Symposium on Operating Systems Design and Implementation*, pages 273–288, December 2004.

[63] Qing Yang, Weijun Xiao, and Jin Ren. TRAP-array: A disk array architecture providing timely recovery to any point-in-time. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 289–301, 2006.