Selective Versioning in a Secure Disk System

A Thesis Presented
by

Swaminathan Sundararaman

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

August 2007

Stony Brook University

The Graduate School

Swaminathan Sundararaman

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. Erez Zadok, Thesis Advisor
Professor Computer Science

Dr. Alexander Mohr, Chairman of Defense
Professor Computer Science

Dr. Radu Grosu
Professor Computer Science

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

Selective Versioning in a Secure Disk System

by

Swaminathan Sundararaman

Master of Science

in

Computer Science

Stony Brook University

2007

Making vital disk data recoverable in the event of malicious system attacks has been a hard problem in storage-system design. Today, an attacker who has gained super-user privileges in a system can easily wipe out or forge all data stored on disk. The fundamental reason for this vulnerability is that almost all security mechanisms in today's systems exist at or above the operating system level. To prevent damage to data even in the event of OS compromise, protection mechanisms have to be implemented at a layer below the operating system: the disk.

In this thesis we present Selective Versioning in a Secure Disk System (SVSDS), that internally performs selective block-level versioning of data by exploiting higher-level data semantics. Because this internal versioning is completely transparent to the operating system and the higher layers of the system, it cannot be by-passed or disabled by malicious attackers. SVSDS leverages the mechanism of Type-Safe Disks to obtain pointer information at the disk level, and versions all meta-data and chosen data items specified by the system administrator. Therefore, our versioning disk system has significantly lower space and performance overheads compared to semantic-agnostic block-level versioning systems, thereby accommodating more versions to improve recoverability.

SVSDS also provides two basic constraints to protect vital data against damage: read-only and append-only. Important configuration and executable files that are rarely updated can be marked as read-only at the disk level, and the disk disallows writes to the corresponding blocks. The append-only constraint can be used to protect system log files that act as the basis for intrusion detection.

We have implemented a prototype SVSDS in the Linux kernel, and our evaluation confirms that it can be built with acceptable space and performance overheads.

To my parents.

Contents

Acknowledgments

I would like to express my utmost gratitude to the following people...

To my advisor Dr. Erez Zadok, for the opportunity to work in FSL, for his encouragement, motivation, allowing me the freedom to pursue my own directions in research, and for always wanting the best for me. To Dr. Alex Mohr and Dr. Radu Grosu, for being on my committee and being generous with their input and comments on this work. To Gopalan Sivathanu, who truly has been a great friend, philosopher, and a guide. To Sean Callanan and Avishay Traeger for their valuable inputs and help throughout. And finally, to all my lab-mates in the FSL, and to Dr. Erez Zadok, for making it such a great place to work and learn.

Chapter 1

Introduction

Data protection is an important problem in systems research. Organizations and institutions spend millions of dollars in research and procurement to protect their data. Today's disks do not have any built-in security mechanisms, so the security of the stored data is tied to the operating system. Operating systems are not completely secure; vulnerabilities are constantly exploited by root kit attacks, buffer overflow attacks, malware and malicious intruders. This makes protecting data stored on a disk a very hard problem. There are some solutions [26, 29] to protect data in the event of an operating-system compromise. The problem with these approaches are that they are either not flexible or incomplete.

Data protection systems can be broadly classified into three categories according to their features: (1) confidentiality, (2) integrity, and (3) recoverability. Confidentiality ensures that data cannot be read by any unauthorized person. Integrity ensures that unauthorized changes to the data do not go unnoticed and availability ensures that either previous versions of the data or copies of the data in exist in a remote location. Data recoverability in the event of system compromise is a primary requirement in critical systems because data loss could lead to serious consequences in most cases. Existing solutions such as encryption and file system or disk level versioning are either ineffective or inefficient. For example, encryption cannot protect against deletion of data.

In this thesis, we propose a selective versioning secure disk system (SVSDS) that transparently versions selected blocks inside the disk. SVSDS versions important (i.e., meta-data) blocks and user-selected files and directories. SVSDS uses pointer information that is available inside Type-Safe Disks (TSD) [26] and leverages it to aid in selectively versioning data inside the disk. A TSD is a disk system that infers block relationships through pointers to enforce invariants on data access, and to provide better security and other semantic-aware optimizations inside the disk.

All blocks inside the disk may not have the same level of importance. Meta-data blocks are more important than data blocks because they impact the reachability of other blocks. For example, if the inode block of a file is corrupt, its data blocks could become unreachable. Also, all files stored on the disk may not be equally important. For example, files present in the */tmp* folder are less important than those in the */usr* folder. Some files are created and deleted within a short period of time (e.g., program installation) and versioning changes to these files would unnecessarily occupy additional space. Existing solutions such as Clotho [5], Trap [34], Peabody [12], and S4 [29] version all data inside the disk, hence they have significant space overhead, which limit the depth of the version history available to users. We would talk about these system in detail in Section 6

Apart from versioning blocks at the disk level, SVSDS also enforces operation-level constraints such as making groups of blocks read-only or append-only. This is one of the important desirable properties of a secure disk system. For example, an intrusion detection system has to protect log files from being overwritten by intruders or malware applications as the correctness of post-intrusion analysis depends on these files.

When data protection mechanisms such as selective versioning and operation-level constraints are combined together, they enable the system to provide stronger data integrity and availability guarantees. The window of time during which the data can be recovered is much wider than that provided by existing solutions that are forced to version all blocks.

SVSDS protects data stored inside the disk by transparently versioning meta-data and user-selected files and directories at regular intervals. It also implements operational-level constraints inside the disk, that help protect immutable and log files from being modified or deleted.

We evaluated our prototype implementation by using micro-benchmarks and real workloads. We found that the cost of performing selective versioing of data while enforcing operation-level constraints are quite minimal. For typical user workloads, SVSDS has an overhead of just 1% compared to regular disks.

The rest of the thesis is organized as follows. Chapter 2 surveys background work. Chapter 3 and Chapter 4 explain the design and implementation of our system. In Chapter 5 we discusses the performance evaluation of our prototype implementation. Related work is discussed in Chapter 6. Chapter 7 talks about possible future work. We conclude in Chapter 8.

Chapter 2

Background

SVSDS uses the pointer information available inside TSDs to selectively version blocks and user-selected files and directories inside the disk. In this section we first describe TSDs, then, we discuss the advantages and disadvantages of ACCESS (A Capability Conscious Extended Storage System) whose objective is to protect data confidentiality even when the operating system is compromised.

2.1 Type-Safe Disks

Today's block-based disks cannot differentiate between block types due to the limited expressiveness of the block interface. All higher-level operations are translated into a set of block read and write requests. Hence, they do not convey any semantic knowledge about the blocks they modify. This problem is popularly known as the information gap in the storage stack [4, 7], and constrains disk systems with respect to the range of functionality that they can provide.

Type-Safe Disks (TSD) try to bridge this information gap through the use of pointers. Pointers, though simple, proved effective in bridging the information gap. Pointers are the smallest unit through which file systems organize data into semantically meaningful entities such as files and directories. Pointers define three things: (1) the semantic dependency between blocks; (2) the logical grouping of blocks; and (3) the importance of blocks. Even though pointers provide vast amounts of information about relationships among blocks, today's disks are oblivious to pointers. A TSD is a disk system that is aware of pointer information and can use it to enforce invariants on data access and also perform various semantic-aware optimizations which are not possible in today's disk systems.

Pointers are the primary mechanisms by which data is organized. Most importantly, pointers define reachability of blocks; i.e., a block this is not pointed to by any other block cannot be reached or accessed. Almost all popular data structures used for storing information use pointers. For example, file systems and database systems make extensive use of pointers to organize the data stored in the disk. Storage mechanisms employed by databases like indexes, hash, lists, and b-trees use pointers to convey relationships between blocks. Popular file systems like Ext2 and VFAT have been modified to support TSDs with negligible effort [26].

TSDs widen the traditional block-based interface to enable the software layers to communicate pointer information to the disk. This allows free-space management to be been moved from the file system to the disk. File systems can use the disk API exported by TSDs to allocate blocks, create pointers between blocks, delete pointers and get free-space information from the disk. TSDs perform automatic garbage collection of deleted blocks, so there is no API call for freeing blocks.

The garbage-collection process performed in TSDs is different from the traditional garbage-collection mechanism employed in most programming languages. A TSD garbage collects deleted blocks in an online fashion as opposed to the traditional offline mechanism in most programming languages. TSDs maintain a reference count for each block (i.e., the number of pointers pointing to that block). When the reference count of a block decreases to zero, the block is garbage collected; the space is reclaimed by the disk and the block is added to the list of free blocks. It is important to note that its the pointer information provided by TSD that allow the disk to track the liveness of blocks, which cannot be done in traditional disks [27].

2.2 ACCESS

Some of the guarantees provided by encryption file systems and data-integrity mechanisms do not hold when the operating system is compromised. In order to overcome this limitation, ACCESS [26] provides a security perimeter at the disk level. ACCESS protects blocks or a chain of blocks through capabilities. Users must provide capabilities to read from and write to blocks. With the help of pointer information inside the disk, ACCESS uses implicit capabilities to access unprotected blocks. Blocks that are pointed to by protected blocks are implicitly protected by the capability of the protected block. To access any block a user must either provide the capability for that block or provide the capability for the reference block that points to it. A session is created when a user provides a capability to access a block, and remains active until it times out. During an active session, all data that could be reached by the protected blocks is vulnerable to attack; ACCESS in its present form does not handle this situation (i.e., it does not detect intrusion and revert back to previously known good state).

Chapter 3

Design

When designing SVSDS, we had the following four goals in mind:

- **Security** We designed our system to ensure data stored inside the disk is protected even in the event of an operating system compromise.

- **Transparent Versioning** We designed our system such that a disk can start versioning a block with little or no user intervention. This is to ensure that disk-level versioning is not bypassed by applications at higher layers. We wanted minimal or no modifications to the file systems that support TSDs to work with SVSDS. For normal operations SVSDS should be completely transparent.

- **Flexibility** We wanted our disk system to be more flexible to allow users to version data at different block granularities. Users can specify per-file or per-directory as the granularity for versions, as users are less concerned with entire file system or disks being versioned. Disks should intelligently version important blocks.

- **Performance** We designed our system such that versioning inside the disks had small overheads as compared to regular operations.

Figure 3 shows the architecture of SVSDS. It is made up of 4 major components: the pointer management layer, the storage virtualization layer (SVL), version manager, and operation manager. The pointer management layer helps in tracking relationships between blocks. The SVL performs transparent versioning of blocks inside the disk. The version manager efficiently tracks and manages versions, and the operation manager implements operation-level constraints as specified by the user.

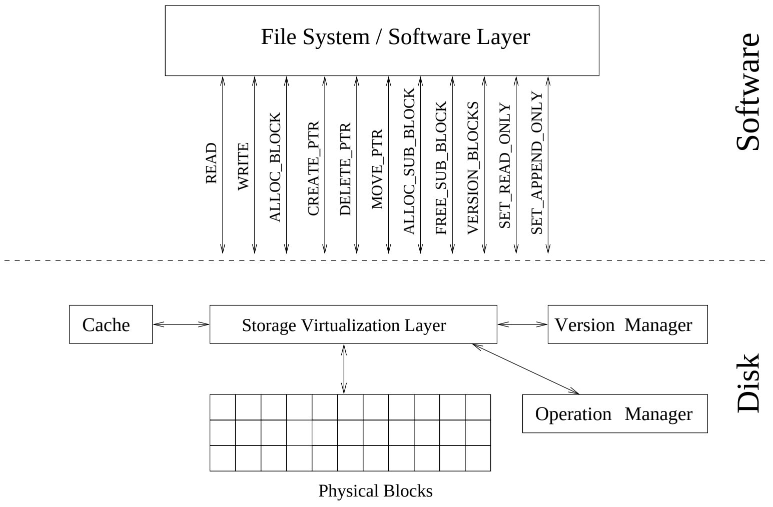

Figure 3.1: Design of Selective Versioning in a Secure Disk System

The rest of the this chapter is organized as follows. Section 3.1 describes how the disk is virtualized. Section 3.2 describes the versioning methodology and how the disk selectively and transparently versions blocks. Section 3.3 describes how the disk tracks modifications to blocks, its block allocation policy, and the garbage collection of deleted blocks. Section 3.4 describes the recovery policy for data blocks. Section 3.5 describes how the versioning process is automated inside the disk. Section 3.6 describes how versions can be reverted back. Section 3.7 describes how operation-level constraints are enforced inside the disk.

Section 3.8 discusses the consistency issues, administrative interface and denial of service attacks. Section 3.9 describes the limitations of our system.

3.1 Virtualizing the disk

Transparent versioning is an important requirement for our system. This is because versioning at the disk level should not by bypassed by applications or file systems that use it. Also, blocks that contain previous versions should not be accessible to applications and file systems. This protects the versioned data even when the operating system is compromised. To transparently version blocks at the disk level, SVSDS virtualizes the physical disk blocks and presents a logical view of these blocks to upper layers. The layer that virtualizes the disk is called the Storage Virtualization Layer (SVL). Its primary function is to export a logical block layer to the applications and maintain the mapping between logical and physical blocks internally.

The *m-table* or mapping table is used by SVSDS to store the relationships between logical and physical blocks. During each block request, the m-table is referred to by the SVL to translate and redirect the request. Flags are also associated with each entry to denote the type and status of the logical block.

3.2 Versioning methodology

SVSDS provides file system level flexibility in versioning blocks inside the disks. It automatically versions reference blocks and user-selected files and directories. To create a new version of a block, SVSDS allocates a new physical block through the SVL, and changes the corresponding entry in the m-table to point to the newly-allocated physical block. An entry is added for the old block in the *v-table* (or version table) with its version number set to the previous version. Reverting to previous versions is discussed in Section 3.6.

The rest of the section is organized as follows. Section 3.2.1 describes why selective versioning is required inside the disk. Section 3.2.2 describes how SVSDS versions meta-data blocks. Section 3.2.3 describes how SVSDS versions user-selected files and directories. Section 3.2.4 describes the additional API calls exported by SVSDS to allow applications to selectively version files and directories, and to notify the disks about operation-level constraints.

3.2.1 Selective block versioning

Versioning all blocks inside the disk would quickly consume all available free space on the disk. Also, versioning all blocks is not a good idea for the following two reasons: (1) it is not desirable to version short lived temporary data (e.g., data in the /tmp folder), and (2) even persistent data blocks do not have the same level of importance. For example, in the Ext2 file system, versioning super, inode, or indirect blocks is more important than versioning data blocks because these blocks define the reachability of other blocks.

3.2.2 Meta-data

In order to selectively version blocks, SVSDS make use of the pointer information available inside TSDs. SVSDS exploits this information to selectively version meta-data (or reference) blocks. When the *create_ptr* call succeeds, SVSDS marks the source block as a reference block in the m-table. This information helps the version manager decide which blocks to version.

3.2.3 User-specified Data

A user may want the disk to automatically version certain files and directories. To selectively version files and directories, SVSDS provides the *version_blocks* API call. SVSDS does a Breadth First Search (BFS) on the p-table starting from the block that is passed to this function. All the blocks traversed during the search are marked for versioning in the m-table. Cycles are common in the p-table (for example, in Ext2TSD [26] there is a pointer from the directory block to the inode block of the sub directory and vice versa). SVSDS detects and skip blocks that have already been marked for versioning.

To version blocks that are added to the file or the directory after the *version_blocks* call, SVSDS checks the flags present in the m-table of the source block during *create_ptr* operation. If the source block is versioned then the destination block that it points to is also marked to be versioned. For example, file systems that use SVSDS to version a file should pass the block number containing the inode of the file to this function.

3.2.4 Disk API

We have widened the disk interface to provide support for versioning files and directories and for applications to specify the operation-level constraints on blocks to the disk.

- VERSION_BLOCKS($Block$): Marks all blocks in the sub-tree starting from block $Block$ to be versioned. The data blocks present in the sub-tree will be versioned along with the reference blocks.

- MARK_READ_ONLY($Block$): Marks all blocks in the sub-tree starting from block $Block$ as read-only.

- MARK_APPEND_ONLY($Block$): Marks all blocks in the sub-tree starting from block $Block$ as append-only. $Block$ itself will not be append-only as it should be a metadata block, with possible non-sequential updates.

3.3 Tracking modifications

This section is organized as follows. Section 3.3.1 discusses how reads and writes are handled by SVSDS. Section 3.3.2 discusses how applications gets free blocks from the disk and Section 3.3.3 discusses the garbage collection mechanism inside the disk.

3.3.1 Reads and writes

SVSDS virtualizes the disk space through the SVL. All read and write requests must be translated to the new physical location before they are processed. When processing a write request, the disk checks if the target is a block that has to be versioned. For versioned blocks, the disk checks if there exists a mapping for the current version in m-table. If yes, it redirects the write to the mapped block. If the mapped version and the current version do not match, the disk requests the SVL to allocate a new physical block that is close to the existing mapped physical block. It then stores the old mapping in the v-table table with the last version number (i.e., current version number - 1).

Finally it changes the m-table entry for the logical block to point to the newly allocated physical block. For data blocks, it gets the mapped physical block from the SVL and redirects the request to the mapped physical block.

3.3.2 Block allocation

The block allocation policy for SVSDS is built on top of that present in TSDs. SVL manages both physical and logical disk space and tracks the relationship between them. Whenever SVSDS gets a request from the software layer to allocate blocks, it passes this request to the SVL. SVL first tries to allocate a physical block from the disk. If the allocation request succeeds it then allocates a new logical block which is returned back to the caller; and it creates a entry in the m-table for the logical and the physical block pair. If the SVL is unable to find a free physical block it tries to free a block from the deleted block list. If the deleted block list is empty it then frees the space occupied by the oldest version.

3.3.3 Garbage collection

Even though SVSDS selectively versions blocks inside the disk, it may eventually use up all the available free space. When all of the disk space is used up, it attempts to free up some space as quickly as possible. It first tries to reclaim deleted data blocks from the deleted blocks LRU list. Once all of the blocks are reclaimed, it them tries to free up space used by older versions. When the disk is reverted back to a previous version, the list of blocks that are in the deleted block list is reclaimed by the garbage collector. This is because as SVSDS does not allow users to go forward in the version history.

3.4 Recoverability of data blocks

When a user reverts back to a previous version, SVSDS cannot perform a perfect reversion in most cases. To perfectly reproduce the previous state, the disk would need to revert all mappings for all blocks, including those that were not versioned. Data blocks are not versioned by default and they could be deleted in the current version, making their recovery difficult if not impossible. SVSDS tries to reuse deleted data blocks as late as possible. To do this, SVSDS maintains a separate LRU list for storing the deleted data blocks. When the system is reverted back to a previous version, the delete pointer operation that caused the deletion of the data block is replayed. During this time SVSDS checks the deleted block list and replays the create pointer operation only if the deleted blocks are still present in the LRU list. This policy of lazy garbage collection allows users to recover not only reference blocks but also those deleted data blocks that have not yet been garbage collected yet. For example, a user may want to revert after inadvertently deleting on entire entire directory. If all data blocks that belong to the directory are not garbage collected, then the user can get back the entire directory. If some of the blocks are already reclaimed by the disk, the user would get back the deleted directory with some missing files.

3.5 Automating the process

SVSDS automatically creates versions at regular intervals. A users or an administrator need not explicitly version data on the disk. However, administrators can specify the versioning frequency through the administrative interface of SVSDS. When a new version is to be created, the SVSDS increments the current version count. It then creates, a entry for the new version in the v-table to store the changes to the blocks that are versioned and the pointer operations that happened during the time the new session is active.

SVSDS in its current form has the same versioning interval for all blocks. Alternatively, we could also implement different policies for different block groups according to user requirements and importance of the data stored inside the blocks. This would provide deeper histories for selected files.

3.6 Reverting back versions

In the event of an intrusion or an operating system compromise, an administrator may want to revert back to a previous safe state of the file system. SVSDS allows administrators to preform cascading reverts to reach a previous version. Even though it is possible to see the contents of a group of blocks for any previous version, it is not possible for SVSDS to jump to any arbitrary version. This is because TSDs use pointer information to track block relationships, and to garbage collect deleted blocks. Suppose we are trying to go from version x to version y. Reverting back v-table entries for version y alone would not work because blocks that were allocated or pointers that were created or deleted between version y and version x would not be reverted back. These blocks cannot be reclaimed back and we would be leaking space. In other words, the reference count for certain blocks could have increased after the version was created (e.g., new blocks are added to increase the size of a file), which would break the garbage collection mechanism in the TSD. Hence, SVSDS allows users to revert back one version at a time.

SVSDS uses the *v-table* to maintain information about previous versions. In order to revert back to the previous version, SVSDS stores all of the blocks whose mappings were changed in the previous version. When the system is reverted back from version x to its previous version $x - 1$, it reverts back all the changes to the m-table stored in version x. While reverting back the mapping table entries, the current m-table (or the physical blocks) pointed to by the logical blocks are freed as they are no longer required.

To revert back pointers, SVSDS tracks pointer operations that happen during the lifetime of a version. To reduce the space required to store the pointer operations, SVSDS does not store all of the pointer operations that happen during a particular session. Blocks (or files) that are created and deleted during a session cannot be reverted back. For example, suppose *create_ptr* is called with source x and destination y. During that particular session, if a *delete_ptr* operation is called with the same source x and destination y, SVSDS removes the entry from the pointer operation list for that session.

SVSDS replays the reverse of the pointer operation in the reverse order. The reverse operation for *create_ptr* operation is *delete_ptr* and vice versa. Other operations, like *alloc_block* and *alloc_bulk_blocks*, are internally translated to *create_ptr* operations. Section 3.6.1 describes replaying pointer operators in more detail.

3.6.1 Reverting Pointer Operations

While reverting back to a previous version, the pointer operations have to be replayed in the reverse order. If not, TSD's the garbage collection mechanism would not work properly and hence it would leak space. Figure 3.2 illustrates this problem.

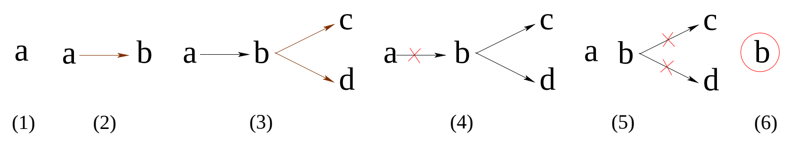

Figure 3.2: Create pointer operation

It can be seen from Figure 3.2 (1, 2 and 3) that block a is the reference block for block b, which in turn is the reference block for blocks c and d. If the pointer operations are replayed in the same order as they occurred during the session, then From figure 3.2 (4,5 and 6) it can been seen that blocks b would not be garbage collected. This is because internally TSDs do not garbage collect blocks that have out going pointers, and also do not check the reference count for the source block to garbage collect if the last outgoing pointer from it is deleted. This situation does not occur when the pointer operations are replayed in the reverse order.

When the *delete_ptr* operations are replayed in the reverse order it does not leave the system in a consistent state. This is because the applications have to issue delete pointer calls in the reverse order as TSDs check the reference count for that particular block before it is deleted.

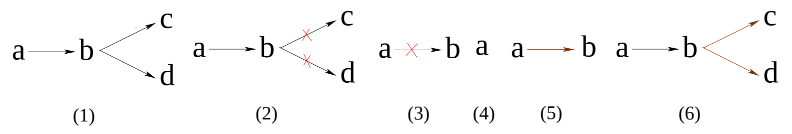

Figure 3.3: Delete pointer operation

Figure 3.3 (1, 2 and 3) shows the sequence of pointer delete operations and Figure 3.3 (4, 5 and 6) show the sequence of steps performed while reverting back the delete pointer operations.

3.7 Operation-level Constraints

In addition to versioning data inside the disk, it is also important to protect certain blocks from being modified or overwritten. SVSDS allows users to specify the types of operations that can be performed on a block. This enables users to assign flexible policies for individual files and directories. Operation-level constraints implemented by SVSDS are read-only and append-only.

The sequence of steps taken by SVSDS to mark a file or a directory as immutable or append-only is similar to marking a file or a directory to be versioned. While marking a group of blocks, the first block (or the root block) of the breadth first search is treated differently to handle atime updates. A special bit is set in the flag along with the bit to indicate that it is a read-only or append-only block. This bit would be used later to infer that this is an inode block.

3.7.1 Read-only Constraint

The read-only operation-level constraint is implemented to make block(s) immutable. For example, the system administrator could mark binaries or directories that contain libraries as read-only so that later on they cannot be modified by a virus, trojan horse, or any other malware application. Since atime updates cannot be distinguished from regular block writes using pointer information, SVSDS disallows atime updates, as they does not change the integrity of the file.

3.7.2 Append-only Constraint

Log files serve as an important resource for intrusion analysis and statistics collection (e.g., Web logs). Operation level constraints implemented by SVSDS can be used to protect important log files from being overwritten or deleted by intruders. Once the administrator marks the log files to be append-only, SVSDS ensures that all operations on block belonging to these files are only appends.

For append-only blocks, SVSDS checks the difference between the original contents and the block being written to verify that data is only being appended, and not overwritten or deleted. In order to avoid reading the original contents of the block during a write operation for comparison, SVSDS caches append-only blocks when the blocks are read from the disk. During the comparison, it checks the cache for the block. If it is not present, SVSDS issues a read request and adds the block to the cache before processing the write request.

As inode blocks of append-only files have to be updated, SVSDS permits the first block of append-only files to be overwritten. It also checks if the same set of bytes is modified every time. Since SVSDS does not have sufficient semantic information about the file system, enforcing append-only constraints on directories does not work properly.

3.8 Issues

This section describes issues we have encountered while developing the SVSDS architecture. Section 3.8.1 describes consistency issues for file systems that use SVSDS. Section 3.8.2 describes the mechanisms for reverting back versions. Section 3.8.3 describes denial of service attacks.

3.8.1 Consistency

Because SVSDS does not know about file-systems semantics, reverting versions might leave the file system that runs on a SVSDS in an inconsistent state. A file system consistency checker (e.g., *fsck*) needs to be run after the disk is reverted back to a previous version. Since SVSDS internally uses pointers to track blocks, the consistency checker should also issue appropriate calls (namely *alloc_block*, *create_ptr*, and *delete_ptr*) to SVSDS to ensure that disk-level pointers are consistent with file system pointers.

3.8.2 Administrative Interfaces

To prevent unauthorized users from reverting versions inside the disk, an SVSDS should have a special hardware interface (or port) through which an administrator can revert back to previous versions. This port can also be used by the administrator to set the versioning frequency.

3.8.3 DoS Attacks

SVSDS are also vulnerable to denial of service attacks. There are three issues to be handled: (1) an intruder could mark arbitrary files for versioning or set the operation-level constraints on them; (2) blocks that are marked for versioning could be repeatedly overwritten; and (3) lots of bogus file could be created to delete old versions. In order to prevent a malicious user from adding files for versioning or setting operation-level constraints, SVSDS permits these operations only through the admin interface. As with most of the denial of service attacks there is no perfect solution to attack of type 3. To counter attacks of type 2, in our current prototype, SVSDS throttles the access to the group of file that is versioned. An alternative solution to this problem would be to exponentially increase the versioning interval of the particular file / directory (i.e., a groups of blocks) that is being constantly overwritten.

3.9 Limitations

Since SVSDS operate at the granularity of blocks, it cannot selectively version individual files in an Ext2TSD file system. This is because in Ext2TSD multiple inodes share the same block on a disk. Hence, other inodes that share the inode block would also be versioned when one of them is versioned. Also, in the current implementation of SVSDS, users do not have access to the previous versions of files.

Chapter 4

Implementation

We implemented a prototype SVSDS as a pseudo-device driver in Linux kernel 2.6.15 that stacks on top of an existing disk block driver. Figure 4.1 shows the pseudo device driver implementation of SVSDS. SVSDS has $7,487$ lines of kernel code out of which $3,060$ were reused from an existing TSD prototype. The SVSDS layer receives all block requests, and re-maps and redirects the common read and write requests to the lower-level device driver. The additional primitives required for operations such as block allocation and pointer management are implemented as driver `ioctl`s.

When a program reads or writes to a block, the requests comes through syscall, and then it passed through VFS and to the file system (e.g.,ext2tsd). From the file system it reaches SVSDS, which with the help of SVL find the corresponding physical blocks, redirects the request to the underlying disk through the generic block driver.

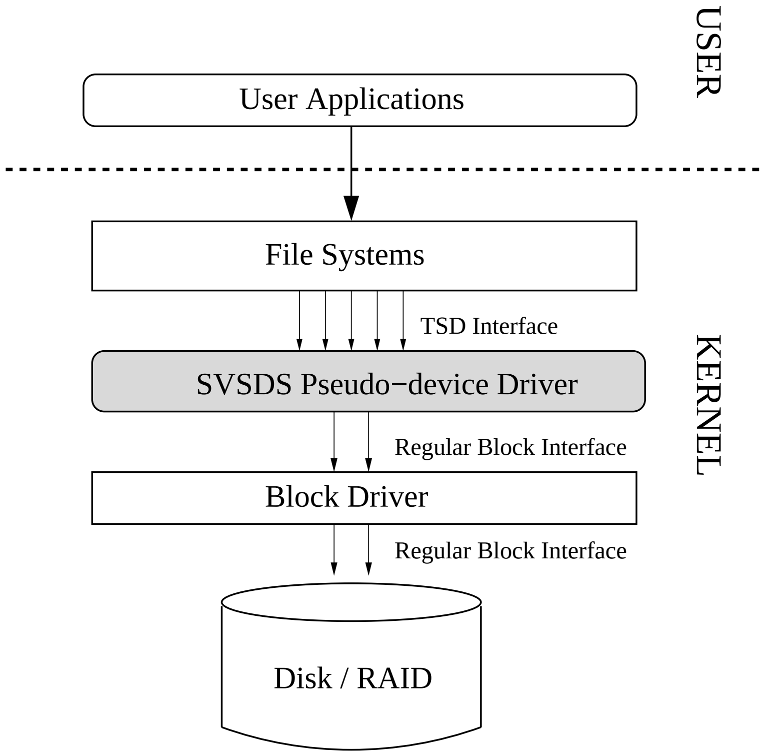

Figure 4.1: Prototype Implementation of SVSDS

Chapter 5

Evaluation

We evaluated the performance of our prototype SVSDS framework in the context of Ext2TSD [26]. We ran general-purpose workloads on our prototypes and compared them with unmodified Ext2 file system on a regular disk. This section is organized as follows: In Section 5.1 we talk about our test platform, configurations, and procedures. In Section 5.2 we analyze the performance of the SVSDS framework on typical user workloads. In Section 5.3 we analyze the performance on OpenSSH workloads. In Section 5.4 we analyze the performance on kernel compile workload.

5.1 Test infrastructure

We conducted all tests on a 2.8GHz Intel Xeon CPU with 1GB RAM, and a 74GB 10Krpm Ultra-320 SCSI disk. We used Fedora Core 6 running a vanilla Linux 2.6.15 kernel. To ensure a cold cache, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-t distribution. In each case, the half-widths of the intervals were less than 5% of the mean. Wait time is the difference between elapsed time and CPU time, and is affected by I/O and process scheduling. We recorded disk statistics from */proc/diskstats* for our test disk. We provide the following detailed disk-usage statistics: the number of read I/O requests (*rio*), number of write I/O requests (*wio*), number of sectors read (*rsect*), number of sectors written (*wsect*), number of read requests merged (*rmerge*), number of write requests merged (*wmerge*), total time taken for read requests (*ruse*), and total time taken for write requests (*wuse*).

Unless otherwise mentioned, the system time overheads were caused by the hash table lookups required during the CREATE_PTR and DELETE_PTR TSD calls. The hash table lookups (on the m-table) required by SVL during every request to the disk add a significant overhead to the system time. This CPU overhead is due to the fact that our prototype is implemented as a pseudo-device driver that runs on the same CPU as the file system. In a real SVSDS setting, the hash table lookups will be performed by the processor embedded in the disk and hence will not influence the overheads on the host system, but will add to wait time.

We have compared the overheads of SVSDS using Ext2TSD against Ext2 on a regular disk and Ext2TSD on a TSD. We denote Ext2TSD on a SVSDS using the name Ext2Ver. The letters M and A are used to denote selective versioning of meta-data and all data respectively.

5.2 Postmark

Postmark [13] simulates the operation of electronic mail and news servers. It does so by performing a series of file system operations such as appends, file reads, directory lookups, creations, and deletions. This benchmark uses little CPU but is I/O intensive. We configured Postmark to create 3,000 files, between 100–200 kilobytes, and perform 300,000 transactions. The results for the postmark benchmark is shown in Figure 5.1.

Figure 5.1 and Table 5.1 show the performance of Ex2TSD on SVSDS for Postmark with a versioning interval of 15 seconds. Postmark deletes all its files at the end of the benchmark, so no space is occupied at the end of the test. SVSDS transparently creates versions and thus, consumes storage space which is not visible to the file system.

For Ext2TSD, the elapsed time was observed to be 1.6% lesser, the system time 1.14 times, and wait time 8.12% lesser that of Ext2. The increase in the system time is because of the hash table lookups during CREATE_PTR and DELETE_PTR calls. The decrease in the wait time is because,

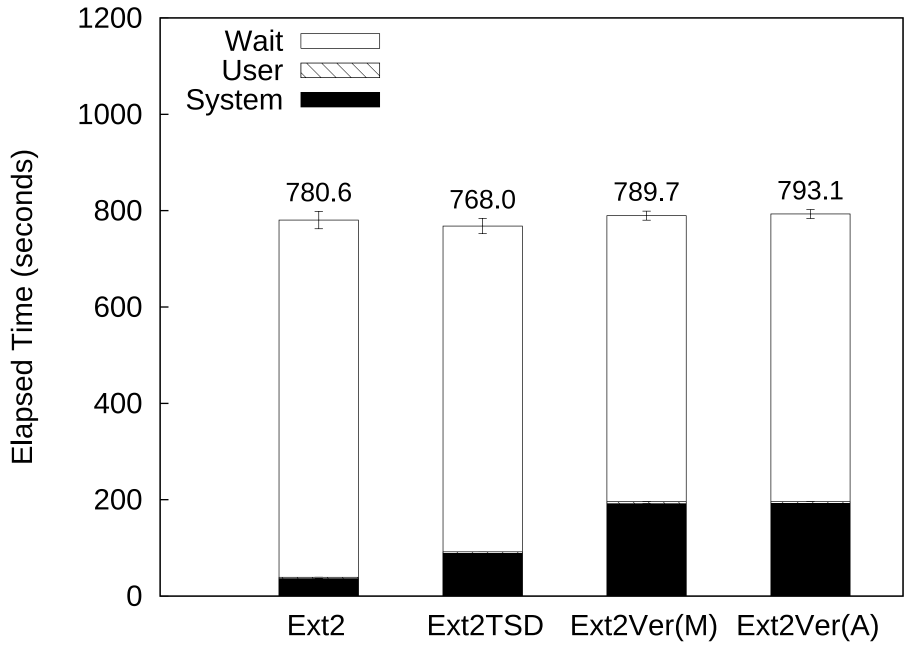

Figure 5.1: Postmark results for SVSDS

unlike Ext2, Ext2TSD does not take into account future growth of files while allocating space for files, this decrease in wait time caused it to perform slight better than ext2 file system on a regular disk, but would have had a more significant impact in a benchmark with files that grow.

For Ext2Ver(M), elapsed time was observed to be 1.17%, or statistically indistinguishable given our confidence interval, system time 4.28 times and wait time -19.91% that of Ext2. The increase in system time is due to the additional hash table lookups to locate entries in the m-table. The decrease in wait time is due to better spacial locality and increased number of requests being merged inside the disk. This is because random writes (i.e., writing inode block along with writing the newly allocated block) were converted to sequential writes due to versioning.

For Ext2Ver(A), elapsed time was observed to be statistically indistinguishable, system time 4.29 times and wait time 19.47% less that of Ext2.

	Ext2	**Ext2TSD**	**Ext2TSD(M)**	**Ext2TSD(A)**
Elapsed	780.5s	768.0s	789.7s	793.1s
System	36.28s	88.58s	191.71s	191.94s
Wait	741.42s	676.11s	593.80s	597.09s
Space o/h	0MB	0MB	443MB	1879MB
Performance Overhead over Ext2				
Elapsed	-	-1.60 %	1.17%	1.61%
System	-	1.44 ×	4.28 ×	4.29×
Wait	-	-8.12 %	-19.91%	-19.47%

Table 5.1: Postmark results.

5.3 OpenSSH Compile

To simulate a relatively heavy user workload, we compiled the OpenSSH source code. We used OpenSSH version 4.5, and analyzed the overheads of Ext2 on a regular disk, Ext2TSD on a TSD, and meta-data and data versioning Ext2TSD on SVSDS for the `untar`, `configure`, and `make` stages combined. These operations in combination constitute a significant amount of CPU activity and I/O operations. The results for the OpenSSH compilation are shown in Figure 5.2.

For Ext2TSD, we recorded a insignificant increase in elapsed time, a an insignificant increase in system time and a 108% increase in the wait time over Ext2. Since the results are statistically indistinguishable, it is difficult to quantify for the increase in wait time.

For Ext2Ver(M), we recorded a 7.2% increase in elapsed time, and a 41% increase in system time over Ext2. Ext2Ver(M) consumed 1.1 MB of additional space to store the versions. The increase in system time overhead is due to the additional hash table lookups by SVL to remap the read and write requests. Ext2Ver(M) consumes 496 KB of additional space to store the versions.

For Ext2Ver(A), we recorded a 7.21% increase in elapsed time, and a 39% increase in system time over Ext2. Ext2Ver(A) consumes 15MB of additional space to store the versions.

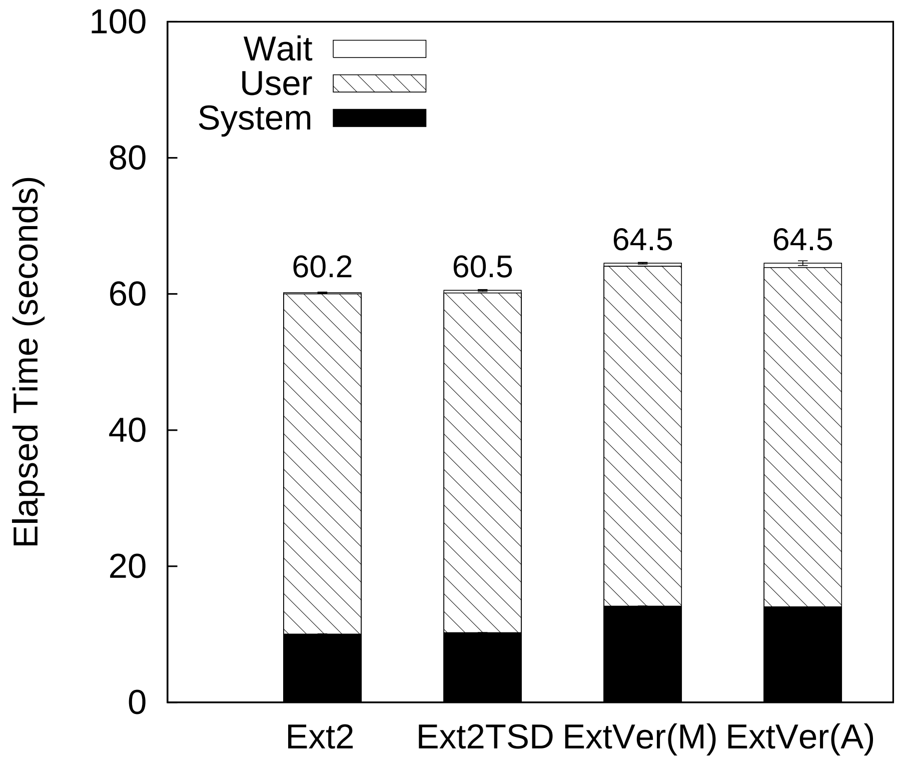

Figure 5.2: OpenSSH Compile Results for SVSDS

	Ext2	**Ext2TSD**	**Ext2TSD(M)**	**Ext2TSD(A)**
Elapsed	60.186s	60.532s	64.520s	64.546s
System	10.027s	10.231s	14.147s	14.025s
Wait	0.187s	0.390s	0.454s	0.634s
Space o/h	0MB	0MB	496KB	15.14MB
Performance Overhead over Ext2				
Elapsed	-	0.57 %	7.20%	7.21%
System	-	2 %	41 %	39%
Wait	-	108 %	142%	238%

Table 5.2: OpenSSH results.

5.4 Kernel Compile

To simulate a CPU-intensive user workload, we compiled the Linux kernel source code. We used a vanilla Linux 2.6.15 kernel and analyzed the overheads of Ext2TSD on a TSD and Ext2TSD on SVSDS with versioning of all blocks and selective versioning of meta-data blocks against regular Ext2, for the `untar`, `make oldconfig`, and `make` operations combined. The results are shown in Figure 5.3.

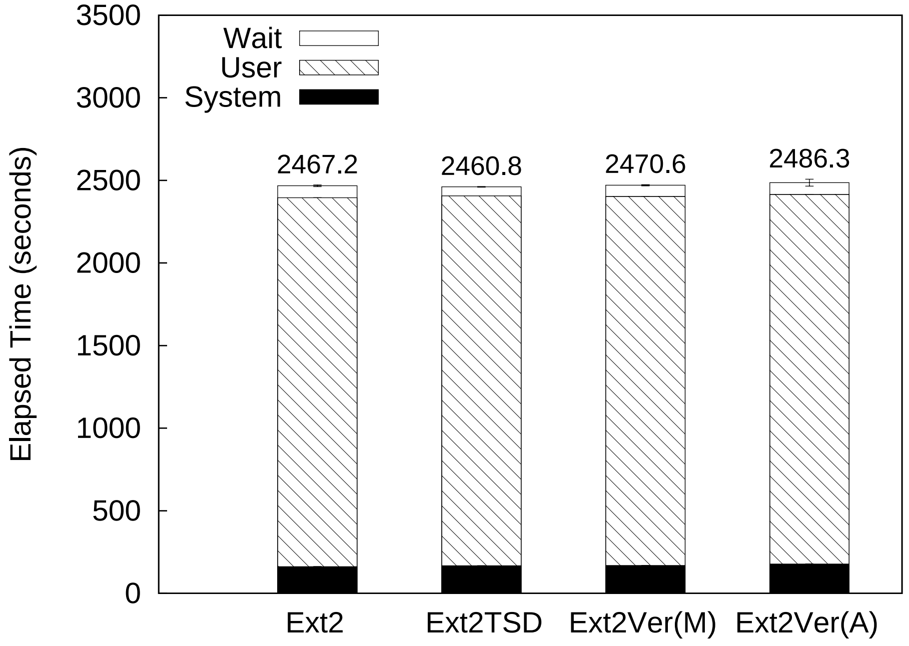

Figure 5.3: Kernel Compile Results

For Ext2TSD, elapsed time was observed to statistically indistinguishable, system time overhead was 3.6% lower and wait time lower by 24% than that of Ext2. The decrease in the wait time is because ext2TSD does not consider future growth of files while allocating new blocks. The decrease in wait time is due to better spatial locality in Ext2TSD.

For Ext2Ver(M), elapsed time was observed to be indistinguishable, system time overhead was 4.6% and wait time lower by 5.6% than that of Ext2. The increase in wait time in relation to ext2TSD is due to versioning meta-data blocks which affect the locality of the stored files. The space overhead of versioning meta-data blocks was 51 MB.

For Ext2Ver(A), elapsed time was observed to be indistinguishable, system time overhead was 10.1% higher and wait time lower by 0.78% than that of Ext2. The increase in wait time in relation to ext2TSD is due to versioning all blocks which worsens the locality of files. The increase in system time is due to the additional hash table lookups for required for storing old mapping information in the v-table. The space overhead of versioning all blocks was 181 MB.

	Ext2	**Ext2TSD**	**Ext2TSD(M)**	**Ext2TSD(A)**
Elapsed	2467s	2461s	2471s	2468s
System	162s	167s	169s	177s
Wait	72.1s	54.7s	68.0s	71.6s
Space o/h	0MB	0MB	51MB	181MB
Performance Overhead over Ext2				
Elapsed	-	-0.26 %	0.13%	0.77%
System	-	3.6%	4.7%	10%
Wait	-	-24%	-5.6%	-0.8%

Table 5.3: Kernel Compile results.

Chapter 6

Related Work

Data protection has been a major focus of systems research in the past decade. Inadvertent user errors, malicious intruders, and malware applications that exploit vulnerabilities in operating systems have exacerbated the need for stronger data protection mechanisms. Various data protection solutions have been proposed that use techniques such as encryption, versioning, snapshotting, and data backup to counter this problem and recover from errors or disasters.

The rest of the this section is organized as follows. Section 6.1 discusses encryption file systems. Section 6.2 discusses version control systems. Section 6.3 describes versioning file systems, their advantages and disadvantages. Section 6.4 talks about systems that take snapshots of the entire file system. Section 6.5 talks about systems that version blocks at the disk level. Finally Section 6.6 discusses systems that virtualize the available storage space.

6.1 Encryption file systems

Encryption file systems [9, 18, 25, 33, 35] are designed to provide data confidentiality. These file systems encrypt data using user-provided keys. Since the keys are only known to the users who created them or have access to the corresponding data, it is not possible for others to decrypt and read the file contents. Though encryption file systems protect the confidentiality of user‘s data, they do not protect malicious users from modifying them. In other words, these systems do not protect the integrity of the stored data. Systems such as SFSRO [6] help detect data modifications by unauthorized users, but not protect against unauthorized modification or help revert back to the original data.

6.2 Version control Systems

CVS [1], Subversion [2], and RCS [32] are a few popular applications that are used for source code management. They do not transparently version data and users have to execute commands to create and access versions. Rational ClearCase [11] is another application that is used for source code management. The disadvantage with this system is that it requires an administrator to manage it and it is expensive.

6.3 Versioning File Systems

Several file systems support versioning. VMS [17] allows previous versions of the files to be accessed via a hidden directory which is located in the same directory as the file whose version is being accessed.

The Elephant file system [24] transparently creates newer versions of the file when the last file handle that points to it is closed. Elephant also provides users with four retention policies: “keep one” performs no versioning, “keep all” retains every version of a file, “keep safe” keeps versions for a specific period of time but does not retain the long-term history of the file, and “keep landmark” retains only important versions in a file’s history. A user can mark a version as a landmark, or the system can use heuristics to mark versions as landmarks. Elephant has its own low-level disk format and cannot be used with other systems. It also lacks the ability to provide an extension list to be included or excluded from versioning. Additionally, user-level applications have to be modified to access old versions of a file.

In the Cedar File System [8], versions are stored on a remote server. Files are copied to the local disk for editing, and transferred to the remote server to create a new immutable version.

VersionFS [19] is a lightweight user-oriented versioning file system that is implemented as a stackable file system and supports user-specified configuration policies. LBFS [20] uses semantic block boundaries to exploit the similarity between files, and versions of the same file and coalesce groups of files to save network bandwidth. The Comprehensive Versioning File System CVFS [28] maintains older versions of files to allow for security rollbacks in the event of an intrusion.

The main problem with all of these file systems is that their security model is closely tied to the operating system. Once the operating system is compromised, an intruder can bypass security checks and change the data stored in the disk except for Cedar. Other problems with versioning at the file-system level are that file systems must be ported to different operating systems. Versioning at the block layer rather than the file system level can overcome this limitation.

6.4 Snapshotting File systems

Snapshotting or check-pointing is an alternative approach to versioning. Some of the popular file systems that implement snapshotting are WAFL [10], 3DFS [14], SnapMirror [22], Ext3cow [23], and ZFS [30].

WAFL creates read-only snapshots of the entire file system. WAFL uses a copy-on-write mechanism to create snapshots of files. Users can access older versions of files through a hidden directory present inside every directory.

In 3DFS, a network file server uses an optical disk jukebox to store snapshots of the file system. A user program locates all recently modified files and sends them to the server. The server writes these files to an optical disk. 3DFS provides read-only access to the previous versions of the files. One noticeable disadvantage of this system is the increased access time due to switching of disks inside the jukebox.

SnapMirror was designed to protect data in the event of a disaster. SnapMirror reduces bandwidth utilization by storing batches of writes, and asynchronously mirroring them to a remote server. The disadvantage with this system is that is tied to Network Appliance filers that require additional disks to store the snapshot data.

Ext3cow extends the popular Ext3 file system to support versioning by changing the meta-data structures of Ext3. Ext3cow extends the inode structure of Ext3 to store the inode number of the previous version. It traverses through the chain to locate previous versions of the files.

ZFS from Sun Microsystems abstracts the physical disks by proving a storage pool to the file system. ZFS never overwrites live blocks, and continuously takes snapshots of the system using a copy-on-write mechanism. These are read-only point-in-time copies of a file system. Since ZFS security is tied to the operating system it is possible for a intruder to read and modify snapshots.

Snapshotting file systems have the same problems as versioning file systems, discussed in Section 6.3.

6.5 Disk Level Versioning

The other alternative to performing versioning at the file system is to version blocks inside the disk. An advantage of this approach is that it is totally decoupled from the operating systems. Some of these systems do not have portability issues with multiple operating system. S4 [29], Clotho [5], TRAP [34] and Peabody [12] implement versioning at the disk level.

The Self-Securing Storage System (S4) internally audits all requests that arrive at the disk, and protects data in compromised systems by combining log-structuring with journal-based metadata versioning in order to prevent intruders from tampering or permanently deleting the data stored on the disk. The guarantees provided by S4 hold true only during the window of time in which it versions the data. When the disk runs out of storage space, S4 stops versioning data until the cleaner thread can free up space for versioning to continue. S4 versions all requests that arrive at the disk, whereas SVSDS versions metadata blocks and user selected file and directory blocks. Additionally, SVSDS enables operation-level constraints that help ensure that certain files and directories marked by an administrator as read-only (e.g., binaries, libraries) can never be modified or deleted. The append-only operational-level constraint in SVSDS protects log files from being over written.

Clotho is a storage block abstraction layer that automatically versions data at the block level. Clotho also provides a storage virtualization layer that is similar to SVSDS to transparently version data blocks. SVSDS differs from Clotho in the following ways: (1) SVSDS does selective replication of important blocks, and most importantly (2) it can version data at the granularity of files and directories.

Timely Recovery to any Point-in-time (TRAP) is a disk array architecture that provides data recovery in different modes. This system has three modes: TRAP-1 takes snapshots at periodic time intervals; TRAP-3 provides timely recovery to any point in time at the block device level (this mode is popularly known as Continuous Data Protection in storage); TRAP-4 is similar to RAID-5, where a log of the parities is kept for each block write. They cannot provide TRAP-2 (data protection at the file-level) as their block-based disk lacks semantic information about the blocks. TRAP-1 is similar to our current implementation where an administrator can choose a particular interval to version blocks. We have implemented TRAP-2, or file-level, versioning inside the disk as SVSDS has semantic information about blocks stored on the disk through pointers. TRAP-3 is similar to the mode in SVSDS where the time between creating versions is set to zero. Since SVSDS runs on a local disk, it cannot implement the TRAP-4 level of versioning. One other difference between TRAP and SVSDS is that SVSDS can also selectively version blocks at the disk level, while TRAP versions all blocks.

Peabody virtualizes the disk space to provide the illusion of a single large disk to the clients. It maintains a centralized repository of sectors and tries to reduce the space utilization by coalescing blocks across multiple virtual disks that contain the same data.

6.6 Storage Virtualization

Storage virtualization has helped systems provide a unified view of disks and partitions to the upper layers. Logical Disks [3], RAID systems [21], (LVM) [16, 31], and Petal [15] are some popular systems that virtualize the available storage space.

The Logical disk is an attempt to separate the file-system implementation from the disk characteristics by providing a logical view of the block device. The storage virtualization layer in SVSDS is analogous to their logical disk layer. The Logical Disk does not have mechanisms to roll back or provide recovery inside the disk.

RAID systems hide physical blocks from multiple disks and export a logical block layer to file systems. RAID systems are designed to provide data redundancy and do not store versions of blocks inside them.

There have been a variety of Logical Volume Management (LVM) implementations. LVMs also hide the physical blocks by exporting a list logical blocks which allows improved management of logical block devices. Some of them provide the ability to take snapshots of a running system using a copy-on-write mechanism and block replication.

Petal exports a virtual block device interface that uses many servers in a distributed fashion. Petal virtual disks provide an explicit snapshotting capability even when the blocks are being written. The disadvantage of Petal is that it is implemented as virtual disk service on the Digital Unix OS, thereby making it specific to a particular operating system.

Chapter 7

Future Work

In the current prototype, SVSDS uses the same versioning interval for reference blocks, and selected file and directory blocks. As individual files and directories have varying levels of importance, users would want to have the option of configuring versioning interval for individual files. Hence, we plan to explore on the feasibility of implementing flexible versioning policy for each file or directory.

In its present form, an administrator can only revert back to a previous version at the granularity of time. To make our system more usable, we plan to provide support for reverting back individual files.

SVSDS already has the components to build a storage intrusion detection system. It transparently versions blocks, has inbuilt support through operation-level constraints to protect libraries, executable files and log files. Hence, the next logical step would be to build an storage IDS on top of SVSDS.

Chapter 8

Conclusion

The main contribution of this work are as follows: (1) we have demonstrated that it is practical to selectively version meta-data blocks inside the disk system, (2) it is possible and useful to enforce operation-level constraints on block access, to protect immutable and log files from being deleted and (3) versioning users-selected files and directories inside the disk is possible in such a disk. Since versioning in done inside the disk, it is difficult for a intruder to bypass the versioning mechanism even after compromising the operating system. Compared to existing systems, SVSDS provides deeper histories by selectively versioning important blocks. SVSDS provides all this functionality with 1.2% overhead for typical user-like workloads.

SVSDS is vulnerable to denial of service attacks. While we provides a partial solution to this problem, we have shown that stronger data protection systems can be implemented with negligible performance overheads.

Bibliography

[1] B. Berliner and J. Polk. Concurrent Versions System (CVS). `www.cvshome.org`, 2001.

[2] CollabNet, Inc. Subversion. `http://subversion.tigris.org`, 2004.

[3] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03),* Bolton Landing, NY, October 2003. ACM SIGOPS.

[4] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proceedings of the Annual USENIX Technical Conference*, pages 177–190, Monterey, CA, June 2002. USENIX Association.

[5] M. D. Flouris and A. Bilas. Clotho: Transparent Data Versioning at the Block I/O Level. In *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)*, pages 315–328, College Park, Maryland, April 2004.

[6] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and Secure Distributed Read-Only File System. In *Proceedings of the 4th Usenix Symposium on Operating System Design and Implementation (OSDI '00),* pages 181–196, San Diego, CA, October 2000. USENIX Association.

[7] G. R. Ganger. Blurring the Line Between OSes and Storage Devices. Technical Report CMU-CS-01-166, CMU, December 2001.

[8] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, 1988.

[9] M. Halcrow. eCryptfs: a stacked cryptographic filesystem. *Linux Journal*, (156):54–58, April 2007.

[10] D. Hitz, J. Lau, and M. Malcom. File System Design for an NFS File Server Appliance. `www.netapp.com/library/tr/3002.pdf`, 2000.

[11] IBM Rational. Rational ClearCase. `www.rational.com/products/clearcase/index.jsp`.

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

[13] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[14] D. G. Korn and E. Krell. The 3-D File System. In *Proceedings of the USENIX Summer Conference*, pages 147–156, Baltimore, MD, Summer 1989.

[15] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-7)*, pages 84–92, Cambridge, MA, 1996.

[16] G. Lehey. The vinum volume manager. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 57–68, Monterey, CA, June 1999. USENIX Association.

[17] K. McCoy. *VMS File System Internals*. Digital Press, 1990.

[18] Microsoft Research. Encrypting file system for windows 2000. Technical report, Microsoft Corporation, July 1999. `www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp`.

[19] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March/April 2004. USENIX Association.

[20] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001. ACM.

[21] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD*, pages 109–116, June 1988.

[22] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleinman, and S. Owara. SnapMirror: File System Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 117–129, Monterey, CA, January 2002. USENIX Association.

[23] Z. N. J. Peterson and R. C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadat for a Time-Shifting File System. Technical Report HSSL-2003-03, Computer Science Department, The Johns Hopkins University, 2003. `http://hssl.cs.jhu.edu/papers/peterson-ext3cow03.pdf`.

[24] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: The File System that Never Forgets. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 2–7, Rio Rica, AZ, March 1999.

[25] D. P. Shanahan. CryptosFS: Fast cryptographic secure nfs. Master's thesis, University of Dublin, Dublin, Ireland, 2000.

[26] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 15–28, Seattle, WA, November 2006. ACM SIGOPS.

[27] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block-Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 379–394, San Francisco, CA, December 2004. ACM SIGOPS.

[28] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata Efficiency in Versioning File Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58, San Francisco, CA, March 2003. USENIX Association.

[29] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the 4th Usenix Symposium on Operating System Design and Implementation (OSDI '00)*, pages 165–180, San Diego, CA, October 2000. USENIX Association.

[30] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. `www.sun.com/software/solaris/ds/zfs.jsp`.

[31] D. Teigland and H. Mauelshagen. Volume managers in linux. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 185–197, Boston, MA, June 2001. USENIX Association.

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

[33] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.

[34] Q. Yang, W. Xiao, and J. Ren. TRAP-array: A disk array architecture providing timely recovery to any point-in-time. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*, pages 289–301. IEEE Computer Society, 2006.

[35] E. Zadok, I. Bădulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998. `www.cs.columbia.edu/~library`.