Selective Versioning in a Secure Disk System

Swaminathan Sundararaman
Stony Brook University

Gopalan Sivathanu
Stony Brook University

Erez Zadok
Stony Brook University

Abstract

Making vital disk data recoverable even in the event of OS compromises has become a necessity, in view of the increased prevalence of OS vulnerability exploits over the recent years. We present the design and implementation of a secure disk system, SVSDS, that performs selective, flexible, and transparent versioning of stored data, at the disk-level. In addition to versioning, SVSDS actively enforces constraints to protect executables and system log files. Most existing versioning solutions that operate at the disk-level are unaware of the higher-level abstractions of data, and hence are not customizable. We evolve a hybrid solution that combines the advantages of disk-level and file-system—level versioning systems thereby ensuring security, while at the same time allowing flexible policies. We implemented and evaluated a software-level prototype of SVSDS in the Linux kernel and it shows that the space and performance overheads associated with selective versioning at the disk level are minimal.

1 Introduction

Protecting disk data against malicious damage is one of the key requirements in computer systems security. Stored data is one the most valuable assets for most organizations and damage to such data often results in irrecoverable loss of money and man power. In today's computer systems, vulnerabilities in the OS are not uncommon. OS attacks through root kits, buffer overflows, or malware cause serious threat to critical applications and data. In spite of this, security policies and mechanisms are built at the OS level in most of today's computer systems. This results in wide-scale system compromise when an OS vulnerability is exploited, making the entire disk data open to attack.

To protect disk data even in the event of OS compromises, security mechanisms have to exist at a layer below the OS, such as the disk firmware. These mechanisms must not be overridable even by the highest privileged OS user, so that even if a malicious attacker gains OS root privileges, disk data would be protected.

Building security mechanisms at the disk-level comes with a key problem: traditional disk systems lack higher-level semantic knowledge and hence cannot implement flexible policies. For example, today's disk systems cannot differentiate between data and meta-data blocks or even identify whether a particular disk block is being used or is free. Disks have no knowledge of higher-level abstractions such as files or directories and hence are constrained in providing customized policies. This general problem of lack of information at the lower layers of the system is commonly referred to as the "information-gap" in the storage stack. Several existing works aim at bridging this information-gap [4, 11, 16, 18].

In this paper, we present the design and implementation of SVSDS, a secure disk system that transparently performs selective versioning of key data at the disk-level. By preserving older versions of data, SVSDS provides a window of time where data damaged by malicious attacks can be recovered through a secure administrative interface. In addition to this, SVSDS enforces two key constraints: *read-only* and *append-only*, to protect executable files and system activity logs which are helpful for intrusion detection.

In SVSDS, we leverage the idea of Type-Safe Disks (TSD) [16] to obtain higher-level semantic knowledge at the disk-level with minimal modifications to storage software such as file systems. By instrumenting file systems to automatically communicate logical block pointers to the disk system, a TSD can obtain three key pieces of information that are vital for implementing flexible security policies. First, by identifying blocks that have outgoing pointers, a TSD differentiates between data and meta-data. Second, a TSD differentiates between used and unused blocks, by just identifying blocks that have no incoming pointers (and hence not reachable from any meta-data block). Third, a TSD knows higher abstractions such as files and directories by just enumerating blocks in a sub-tree of the pointer hierarchy. For example, the sub-tree of blocks starting from an inode block of an Ext2 file system belong to a collection of files.

Using this semantic knowledge, SVSDS aggressively versions all meta-data blocks, as meta-data impact the accessibility of normal data, and hence is more important. It also provides an interface through which administrators can choose specific files or directories for versioning, or for enforcing operation-based constraints (read-only or append-only). SVSDS uses its knowledge of free and used blocks to place older versions of meta-data and chosen data, and virtualizes the block address-space. Older versions of blocks are not accessible to higher layers, except through a secure administrative interface upon authentication using a capability.

We implemented a prototype of SVSDS in the Linux kernel as a pseudo-device driver and evaluated its correctness and performance. Our results show that the overheads of selective disk-level versioning is quite minimal. For a normal user workload SVSDS had a small overhead of 1% compared to regular disks.

The rest of the paper is organized as follows. Section 2 describe background. Section 3 discusses the threat model. Section 4 and Section 5 explain the design and implementation of our system respectively. In Section 6, we discuss the performance evaluation of our prototype implementation. Related work is discussed in Section 7 and we conclude in Section 8.

2 Background

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. In this section we first talk about versioning as a means for protecting data. We then give a brief description about TSDs to make the paper self-contained.

2.1 Data Versioning

Versioning data is a widely accepted solution to data protection especially for data recovery. Versioning has been implemented in different layers. It has been implemented above the operating system (in applications), inside the operating system (e.g., in file systems) and beneath the operating system (e.g., inside the disk firmware). We now discuss the advantages and disadvantages of versioning at the different layers.

Application-level versioning. Application-level versioning is primarily used for source code management [1, 2, 22]. The main advantage of these systems is that they provide the maximum flexibility as users can control everything from choosing the versioning application to creating new versions of files. The disadvantage with these systems is that they lack transparency and users can easily bypass the versioning mechanism. The versioned data is typically stored in a remote server and becomes vulnerable when the remote server's OS gets compromised.

File-system–level versioning. Several file systems support versioning [6, 10, 12, 15, 19]. These systems are mainly designed to allows users to access and revert back to previous versions of files. The older versions of files are typically stored under a hidden directory beneath its parent directory or on a separate partition. As these file systems maintain older versions of files, they can also be used for recovering individual files and directories in the event of an intrusion. Unlike application-level versioning systems, file-system–level versioning is usually transparent to higher layers. The main advantage of these versioning systems is that they can selectively version files and directories and can also support flexible versioning policies (e.g., users can choose different policies for each file or directory). Once a file is marked for versioning by the user, the file system automatically starts versioning the file data. The main problem with file-system–level versioning is that their security is closely tied to the security of the operating system. When the operating system is compromised, an intruder can bypass the security checks and change the data stored in the disk.

Disk-level versioning. The other alternative is to version blocks inside the disk [7, 20, 23]. The main advantage of this approach is that the versioning mechanism is totally decoupled from the operating system and hence can make data recoverable even when the operating system is compromised. The disadvantage with block-based disk-level versioning systems is that they cannot selectively version files as they lack semantic information about the data stored inside them. As a result, in most cases they end up versioning all the data inside the disk which causes them to have significant amount of space overheads in storing versions.

In summary, application-level versioning is weak in terms of security as can be easily bypassed by users. Also, the versioning mechanism is not transparent to users and can be easily disabled by intruders. File-system—level data-protection mechanisms provide transparency and also flexibility in terms of what data needs to be versioned but they do not protect the data in the event of an operating system compromise. Disk-level versioning systems provide better security than both application and file system level versioning but they do not provide any flexibility to the users to select the data that needs to be versioned. What we propose is a **hybrid solution**, i.e., combine the strong security that the disk-level data versioning provide, with the flexibility of file-system—level versioning systems.

2.2 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 such as file creation, deletion, extension, renaming, etc. 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, 5], and constrains disk systems with respect to the range of functionality that they can provide.

Pointers are the primary mechanisms by which data is organized. Most importantly, pointers define reachability of blocks; i.e., a block that 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.

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 Type-Safe Disk (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.

TSDs widen the traditional block-based interface to enable the software layers to communicate pointer information to the disk. File systems that use TSDs should use the disk APIs (CREATE_PTR, DELETE_PTR, ALLOC_BLOCK, GETFREE) exported by TSDs to allocate blocks, create and delete pointers, and get free-space information from the disk.

The pointer manager in TSDs keeps track of the relationship among blocks stored inside the disk. The pointer operations supported by TSDs are CREATE_PTR and DELETE_PTR. Both operations take two arguments: source and destination block numbers. The pointer manager uses a P-TABLE (or pointer table) to maintain the relationship among blocks inside the disk. Entries are added to and deleted from the P-TABLE during CREATE_PTR and DELETE_PTR operations. When there are no incoming pointers to a block it is automatically garbage collected by the TSD.

One other important difference between a regular disk and a TSD is that the file systems no longer does free-space management (i.e., file systems no longer need to maintain bitmaps to manage free space). The free-space management is entirely moved to the disk. TSDs export ALLOC_BLOCK API to allow file systems to request new blocks from the disk. The ALLOC_BLOCK API takes a reference block number, a hint block number, and the number of blocks as arguments and allocates the requested number of file system blocks from the disk maintained free block list. After allocating the new blocks, TSD creates pointers from the reference block to each of the newly allocated blocks.

The garbage-collection process performed in TSDs is different from the traditional garbage-collection mechanism employed in most programming languages. A TSD reclaims back the deleted blocks in an online fashion as opposed to the traditional offline mechanism in most programming languages. TSDs maintain a reference count (or the number of incoming pointers) for each 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 it is the pointer information provided by TSD that allows the disk to track the liveness of blocks, which cannot be done in traditional disks [17].

3 Threat Model

Broadly, SVSDS provides a security boundary at the disk level and makes vital data recoverable even when an attacker obtains root privileges. In our threat model, applications and the OS are untrusted, and the storage subsystem comprising the firmware and magnetic media is trusted. The OS communicates with the disk through a narrow interface that does not expose the disk internal versioning data. Our model assumes that the disk system is physically secure, and the disk protects against attackers that compromise a computer system through the network. This scenario covers a major class of attacks inflicted on computer systems today.

Specifically, an SVSDS provides the following guarantees:

- All meta-data and chosen file data marked for protection will be recoverable to an arbitrary previous state even if an attacker maliciously deletes or overwrites the data, after compromising the OS. The depth of history available for recovery is solely dependent on the amount of free-space available on disk. Given the fact that disk space is cheap, this is an acceptable dependency.

- Data items explicitly marked as *read-only* is guaranteed to be intact against any malicious deletion or overwriting.

- Data items marked as *append-only* can never be deleted or overwritten by any OS attacker.

It is important to note that SVSDS is designed to protect the data stored on the disk and does not provide any guarantee on which binaries/files are actually executed by the OS (e.g., rootkits could change the binaries in memory). As files with operation-based constraints (specifically read-only constraints) cannot be modified inside SVSDS, upon a reboot, the system running on SVSDS would return to a safe state (provided the system executables and configuration files are marked as read-only).

4 Design

Our aim while designing SVSDS is to combine the security of disk-level versioning, with the flexibility of versioning at higher-layers such as the file system. By transparently versioning data at the disk-level, we make data recoverable even in the event of OS compromises. However, today's disks lack information about higher-level abstractions of data (such as files and directories), and hence cannot support flexible versioning granularities. To solve this problem, we leverage Type-Safe Disks (TSDs) [16] and exploit higher-level data semantics at the disk-level.

Type-safe disks export an extended block-based interface to file systems. In addition to the regular block `read` and `write` primitives exported by traditional disks, TSDs support pointer management primitives that can be used by file systems to communicate pointer-relationships between disk blocks. For example, an Ext2 file system can communicate the relationships between an inode block of a file and its corresponding data blocks. Through this, logical abstractions of most file systems can be encoded and communicated to the disk system. Figure 1 shows the on-disk layout of Ext2. As seen from Figure 1, files and directories can be identified using pointers by just enumerating blocks of sub-trees with inode or directory blocks as root.

The overall goals of SVSDS are the following:

- Perform block versioning at the disk-level in a completely transparent manner such that higher-level software (such as file systems or user applications) cannot bypass it. System administrators or users can set up versioning policies or revert and delete versions through an offline privileged channel after a capability-based authentication process enforced by the disk system.

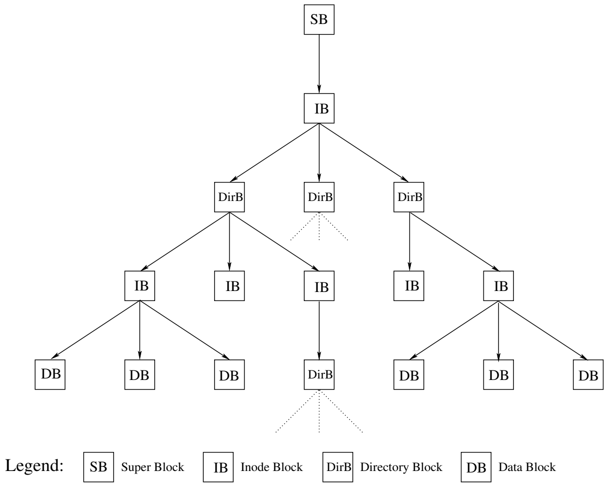


Figure 1: Pointer relationship inside an FFS-like file system

- Aggressively version all meta-data (e.g., Ext2 inode blocks) and chosen data as per the policies set up by administrators or users. In the perspective of a file system, versioning policies must be at granularities of individual files or directories.

- Enforce basic constraints at the disk-level, such as *read-only* and *append-only*. Users must be able to choose specific files or directories to be protected by these constraints.

Figure 2 shows the overall architecture of SVSDS. The three major components in SVSDS are, (1) Storage virtualization Layer (SVL), (2) The Version Manager, and (3) The Constraint Manager. The SVL virtualizes the block address space and manages physical space on the device. The version manager automatically versions meta-data and user-selected files and directories. It also provides an interface to revert back the disk state to previous versions. The constraint manager enforces read-only and append-only operation-level constraints on files and directories inside the disk.

The rest of this section is organized as follows. Section 4.1 describe how transparent versioning is performed inside SVSDS. Section 4.2 talks about the versioning mechanism. Section 4.4 describes our recovery mechanism and how an administrator recovers after detecting an OS intrusion. Section 4.5 describes how SVSDS enforces operation based constraints on files and

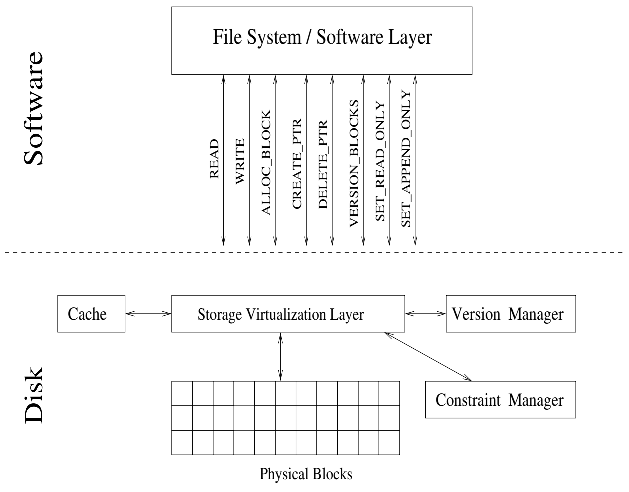


Figure 2: Architecture of SVSDS

directories. Finally, in Section 4.6, we discuss some of the issues with SVSDS.

4.1 Transparent Versioning

Transparent versioning is an important requirement, as SVSDS has to ensure that the versioning mechanism is not bypassed by higher layers. To provide transparent versioning, the storage virtualization layer (SVL) virtualizes the disk address space. The SVL splits the disk address space into two: logical and physical, and internally maintains the mapping between them. The logical address space is exposed to file systems and the SVL translates logical addresses to physical ones for every disk request. This enables SVL to transparently change the underlying physical block mappings when required, and applications are completely oblivious to the exact physical location of a logical block.

SVSDS maintains T-TABLE (or translation table), to store the relationship between logical and physical blocks. There is a one-to-one relationship between each logical and physical block in the T-TABLE. A version number field is also added to each entry of T-TABLE to denote the last version in which a particular block was modified. Also, a status flag is added to each T-TABLE entry to indicate the type (meta-data or data), and status (versioned or non-versioned) of each block. The T-TABLE is indexed by the logical block number and every allocated block has an entry in the T-TABLE. When applications read (or write) blocks, the SVL looks up the T-TABLE for the logical block and redirects the request to the corresponding physical block stored in the T-TABLE entry.

Free-Space Management SVSDS has two different address spaces, whereas the regular TSDs only have one. Hence, SVSDS cannot reuse the existing block allocation mechanism of regular TSDs. To manage both address spaces, the SVL uses two different bitmaps: logical block bitmaps (LBITMAPS) in addition to the existing physical block bitmaps (PBITMAPS). SVSDS uses a two-phased block allocation process. During the first phase, the SVL allocates the requested number of physical blocks from PBITMAPS. The allocation request need not always succeed as some of the physical blocks are used for storing the previous versions of blocks. If the physical block allocation request succeeds, it proceeds to the next phase. In the second phase, the SVL allocates an equal number of logical blocks from LBITMAPS. It then associates each of the newly allocated logical block with a physical block and adds an entry in the T-TABLE for each pair. The flags for these new entries are copied from the reference block passed to the ALLOC_BLOCK call and the version number is copied from the disk maintained version number. This ensures that all blocks that are added later to a file inherit the same attributes (or flags) as their parent block.

4.2 Creating versions

The version manager is responsible for creating new versions and maintaining previous versions of data on the disk. The version manager provides the flexibility of file-system–level versioning while operating inside the disk. By default, it versions all meta-data blocks. In addition, it can also selectively version user-selected files and directories. The version manager automatically checkpoints the meta-data and chosen data blocks at regular intervals of time, and performs copy-on-write upon subsequent modifications to the data. The version manager maintains a global version number and increments it after every checkpoint interval. The checkpoint interval is the time interval after which the version number is automatically incremented by the disk. SVSDS allows an administrator to specify the checkpoint interval through its administrative interface.

The version manager maintains a table, V-TABLE (or version table), to keep track of previous versions of blocks. For each version, the V-TABLE has a separate list of logical-to-physical block mappings for modified blocks.

Once the current version is checkpointed, any subsequent write to a versioned block creates a new version for that block. During this write, the version manager also backs up the existing logical to physical mapping in the V-TABLE. To create a new version of a block, the version manger allocates a new physical block through the SVL, changes the corresponding logical block entry in the T-

TABLE to point to the newly allocated physical block, and updates the version number of this entry to the current version. Figure 3 shows a V-TABLE with a few entries in the mapping list for the first three versions. Let's take a simple example to show how entries are added to the V-TABLE. If block 3 is overwritten in version 2, the entry in the T-TABLE for block 3 is added to the mapping list of the previous version (i.e., version 1).

Versioning TSD Pointer Structures TSDs maintains their own pointer structures inside the disk to track block relationships. The pointer management in TSDs was explained in Section 2.2. The pointers refers to the disk-level pointers inside TSDs, unless otherwise mentioned in the paper. As pointers are used to track block liveness information inside TSDs, the disk needs to keep its pointer structures up to date at all times. When the disk is reverted back to the previous version, the pointer operations performed in the current version have to be undone for the disk to reclaim back the space used by the current version.

To undo the pointer operations, SVSDS logs all pointer operations to the pointer operation list of the current version in the V-TABLE. For example, in Figure 3 the first entry in the pointer operation list for version 1 shows that a pointer was created between logical blocks 3 and 8. This create pointer operation has to be undone when the disk is reverted back from version 1 to 0. Similarly, the first entry in the pointer operation list for version 3 denotes that a pointer was deleted between logical blocks 3 and 8. This operation has to be undone when the disk is reverted back from version 3 to version 2.

To reduce the space required to store the pointer operations, SVSDS does not store pointer operations on blocks created and deleted (or deleted and created) within the same version. When a CREATE_PTR is issued with source a and destination b in version x. During the lifetime of the version x, if a DELETE_PTR operation is called with the same source a and destination b, then the version manager removes the entry from the pointer operation list for that version in the V-TABLE. We can safely remove these pointer operations because CREATE_PTR and DELETE_PTR operations are the inverse of each other and would cancel out their changes when they occur withing the same version. The recovery manager maintains a hash table indexed on the source and destination pair for efficient retrieval of entries from the V-TABLE.

4.3 Selective Versioning

Current block-based disk systems lack semantic information about the data being stored inside. As a result, disk-level versioning systems [7, 23] version all blocks. But versioning all blocks inside the disk can quickly consume all available free space on the disk. Also, versioning all blocks is not efficient for the following two reasons: (1) short lived temporary data (e.g., data in the */tmp* folder and installation programs) need not be versioned, and (2) persistent data blocks have varying levels of importance. For example, in FFS-like file systems, versioning the super block, inode blocks, or indirect blocks is more important than versioning data blocks as the former affects the reachability of other blocks stored inside the disk. Hence, SVSDS selectively versions meta-data and user-selected files and directories to provide deeper version histories.

Versioning meta-data. Meta-data blocks have to be versioned inside the disk for two reasons. First, reachability: meta-data blocks affects the reachability of data blocks that it points to (e.g., the data blocks can only be reached through the inode or the indirect block). Second, recovery of user-selected files: we need to preserve all versions of the entire file system directory-structure inside the disk to revert back files and directories.

To selectively version meta-data blocks, SVSDS uses the pointer information available inside the TSDs. SVSDS identifies a meta-data block during the first CREATE_PTR operation the block passed as the source is identified as a meta-data block. For all source block passed to the CREATE_PTR operation, SVSDS marks it as meta-data in the T-TABLE.

SVSDS defers reallocation of deleted data blocks until there are no free blocks available inside the disk. This ensures that for a period of time the deleted data blocks will still be valid and can be restored back when their corresponding meta-data blocks are reverted back during recovery.

To version files and directories, applications issue an ioctl to the file system that uses SVSDS. The file system in turn locates the logical block number of the file's inode block, and calls the VERSION_BLOCKS disk primitive. VERSION_BLOCKS is a new primitive added to the existing disk interface for applications to communicate the files for versioning (see Table 1). After the blocks of the file are marked for versioning, the disk automatically versions the marked blocks at regular intervals.

Versioning user-selected data. Versioning meta-data blocks alone does not make the disk system more secure. Users still want the disk to automatically version certain files and directories. To selectively version files and directories, applications and file systems only have to pass the starting block (or the root of the subtree) under which all the blocks needs to be versioned. For example, in Ext2 only the inode block of the file or the directory needs to be passed for versioning. SVSDS does

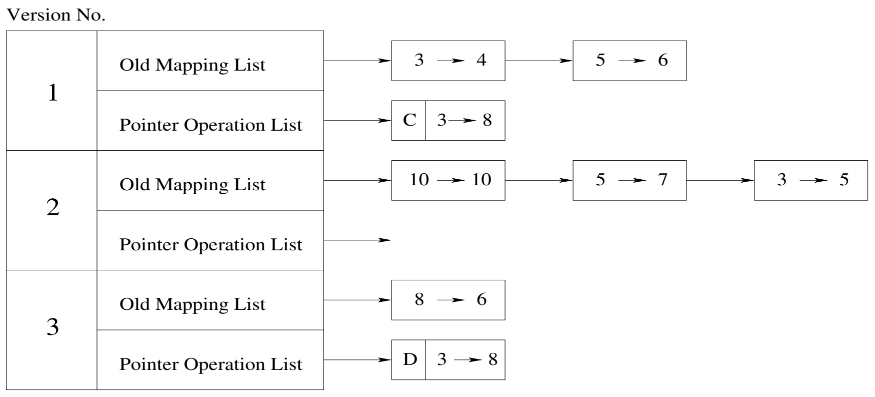


Figure 3: **The v-table data structure.** *A simplified v-table state is shown for first three versions in SVSDS. Each entry in the old mapping list corresponds to logical and physical block pair. C & D in the pointer operation list represent Create pointer and Delete pointer operations, respectively.*

a Breadth First Search (BFS) on the P-TABLE, starting from the root of the subtree. All the blocks traversed during the BFS are marked for versioning in the T-TABLE.

One common issue in performing BFS is that there could potentially be many cycles in the graph that is being traversed. For example, in the Ext2TSD [16] file system, there is a pointer from the inode of the directory block, to the inode of the sub-directory block and vice versa. Symbolic links are yet another source of cycles. SVSDS detects cycles by maintaining a hash table (D-TABLE) for blocks that have been visited during the BFS. During each stage of the BFS, the version manager checks to see if the currently visited node is present in the D-TABLE before traversing the blocks pointed to by this block. If the block is already present in the D-TABLE, SVSDS skips the block as it was already marked for versioning. If not, SVSDS adds the currently visited block to the D-TABLE before continuing with the BFS.

To identify blocks that are subsequently added to versioned files or directories, SVSDS checks the flags present in the T-TABLE of the source block during the CREATE_PTR operations. This is because when file systems want to get a free block from SVSDS, they issue an ALLOC_BLOCK call with a reference block and the number of required blocks as arguments. This ALLOC_BLOCK call is internally translated to a CREATE_PTR operation with the reference block and the newly allocated block as its arguments. If the reference block is marked to be versioned, then the destination block that it points to is also marked for versioning. File systems normally pass the inode or the indirect block as the reference block.

4.4 Reverting Versions

In the event of an intrusion or an operating system compromise, an administrator would want to undo the changes done by an intruder or a malicious application by reverting back to a previous safe state of the disk. We define reverting back to a previous versions as restoring the disk state from time t to the disk state at time t - tv, where tv is the checkpoint interval.

Even though SVSDS can access any previous version's data, we require reverting only one version at a time. This is because SVSDS internally maintains state about block relationships through pointers, and it requires that the pointer information be properly updated inside the disk to garbage-collect deleted blocks. To illustrate the problem with reverting back to an arbitrary version, let's revert the disk state from version f to version a by skipping reverting of the versions between f and a. Reverting back the V-TABLE entries for version a alone would not suffice. As we directly jump to version a, the blocks that were allocated, and pointers that were created or deleted between versions f and a, are not reverted back. The blocks present during version a does not contain information about blocks created after version a. As a result, blocks allocated after version a becomes unreachable by applications but according to pointer information in the P-TABLE they are still reachable. As a result, the disk will not reclaim back these block and the we will be leaking disk space. Hence, SVSDS allows an administrator to revert back only one version at a time.

SVSDS also allows an administrator to revert back the disk state to a arbitrary point in time by reverting back one version at a time until the largest version whose start time is less than or equal to the time mentioned by the administrator is found. REVERT_TO_PREVIOUS_VERSION and REVERT_TO_TIME

Disk Primitives	Description
VERSION_BLOCKS(BNo)	Marks all blocks in the subtree starting from block BNo to be versioned. The data blocks present in the subtree will be versioned along with the reference (or meta-data) blocks.
REVERT_TO_PREVIOUS_VERSION	Reverts back the disk state from current version to the previous version.
REVERT_TO_TIME(t)	Reverts back the disk state one version at a time till it finds a version v with start time less than or equal to t.
MARK_READ_ONLY(BNo)	Marks all blocks in the sub-tree starting from block BNo as read-only.
MARK_APPEND_ONLY(BNo)	Marks all blocks in the sub-tree starting from block BNo as append-only. BNo itself will not be an append-only block as it could be a meta-data block, with non-sequential updates.

Table 1: *Additional Disk APIs in SVSDS*

are the additional primitives added to the existing disk interface to revert back versions by the administrator (see Table 1).

While reverting back to a previous version, SVSDS recovers the data by reverting back the following: (1) *Pointers*: the pointer operation that happened in the current version are reverted back; (2) *Meta-data*: all meta-data changes that happened in the current version are reverted back; (3) *Data-blocks*: all versioned data blocks and some (or all) of the non-versioned deleted data-blocks are reverted back (i.e., the non-versioned data blocks that have been garbage collected cannot be reverted back); and (4) *Bitmaps*: both logical and physical block bitmap changes that happened during the current version are reverted.

4.4.1 Reverting Mapping

SVSDS reverts back to its previous version from the current version in two phases. In the first phase, it restores all the T-TABLE entries stored in the mapping list of the previous version in the V-TABLE. While restoring back the T-TABLE entries of the previous version, there are two cases that need to be handled. (1) An entry already exists in the T-TABLE for the logical block of the restored mapping. (2) An entry does not exist. When an entry exists in the T-TABLE, the current mapping is replaced with the old physical block from the mapping list in the V-TABLE. The current physical block is freed by clearing the bit corresponding to the physical block number in the PBITMAPS. If an entry does not exist in the T-TABLE, it implies that the block was deleted in the current version and the mapping was backed up in the V-TABLE. SVSDS restores the mapping as a new entry in the T-TABLE and the logical block is marked as used in the LBITMAPS. The physical block need not be marked as used as it is already alive. At the end of the first phase, SVSDS restores back all the versioned data that got modified or deleted in the current version.

4.4.2 Reverting Pointer Operations

In the second phase of the recovery process, SVSDS reverts back the pointer operations performed in the current version by applying the inverse of the pointer operations. The inverse of the CREATE_PTR operation is a DELETE_PTR operation and vice versa. The pointer operations are reverted back to free up the space used by blocks created in the current version and also for restoring pointers deleted in the current version.

Reverting back CREATE_PTR operations are straight forward. SVSDS issues the corresponding DELETE_PTR operations. If there are no incoming pointers to the destination blocks of the DELETE_PTR operations, the disk automatically garbage collects the destination blocks.

While reverting the DELETE_PTR operations, SVSDS checks if the destination blocks are present in the T-TABLE. If yes, SVSDS executes the corresponding CREATE_PTR operations. If the destination blocks is not present in the T-TABLE, it implies that the DELETE_PTR operations were performed on non-versioned blocks. If the destination blocks are present in the deleted block list, SVSDS restores the backed up T-TABLE entries from the deleted block list and issues the corresponding CREATE_PTR operations.

While reverting back to a previous version, the inverse pointer operations have to be replayed in the reverse order. If not, SVSDS would prematurely garbage collect these blocks. We illustrate this problem with a simple example. From Figure 4(a) we can see that block a has a pointer to block b and block b has pointers to blocks c and d. The pointers from b are first deleted and then the pointer from a to b is deleted. This is shown in Figs. 4(b) and 4(c). If the inverse pointer operations are applied in the same order, first a pointer would be is created from block b to d (assuming pointer from b to d is deleted first) but block b would be automatically garbage collected by SVSDS as there are no incoming pointers to block b. Replaying pointer operations in the reverse order avoids this problem. Figs 4(d), 4(e), and 4(f) show the sequence of

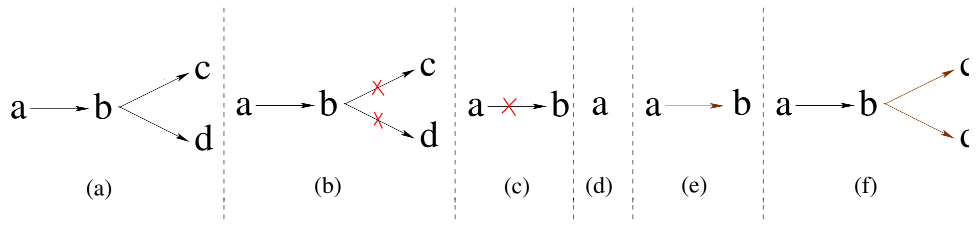


Figure 4: **Steps in reverting back delete pointer operations**

steps performed while reverting back the delete pointer operations in the reverse order. We can see that reverting back pointer operations in the reverse order correctly reestablishes the pointers in the correct sequence.

4.4.3 Reverting Meta-Data

SVSDS uses the mapping information in the V-TABLE to revert back changes to the meta-data blocks. There are three cases that need to be handled while reverting back meta-data blocks: (1) The meta-data block is modified in the new version, (2) The meta-data block is deleted in the new version, and (3) The meta-data block is first modified and then deleted in the new version. In the first case, the mappings that are backed up in the previous version for the modified block in the V-TABLE are restored. This is done to get back the previous contents of the meta-data blocks. For the second case, the delete pointer operations would have caused the T-TABLE entries to be backed up in the V-TABLE as they would be the last incoming pointer to the meta-data blocks. The T-TABLE entries will be restored back in the first phase of the recovery process and the deleted pointers are restored back in the second phase of the recovery process. Reverting meta-data blocks when they are first modified and then deleted is the same as in reverting meta-data blocks when they are deleted.

4.4.4 Reverting Data Blocks

When the recovery manager reverts back to a previous version, it cannot revert back to the exact disk state in most cases. To revert back to the exact disk state, the disk would need to revert mappings for all blocks, including the data blocks that are not versioned by default. In a typical TSD scenario, blocks are automatically garbage collected as soon as the last incoming pointer to them is deleted, making their recovery difficult if not impossible. The garbage collector in SVSDS tries to reclaim the deleted data blocks as late as possible. To do this, SVSDS maintains an LRU list of deleted non-versioned blocks (also known as the deleted block list).

When the delete-pointer operations are reverted back, SVSDS issues the corresponding create-pointer operations only if the deleted data blocks are still present in the deleted block list. This policy of lazy garbage collection allows users to recover the deleted data blocks that have not yet been garbage collected yet.

Lazy garbage collection is also useful when a user reverts back the disk state after inadvertently deleting a directory. If all data blocks that belong to the directory are not garbage collected, then the user can get back the entire directory along with the files stored under it. If some of the blocks are already reclaimed by the disk, the user would get back the deleted directory with data missing in some files. Even though SVSDS does not version all data block, it still tries to restore back all deleted data blocks when disk is revert back to its previous version.

4.4.5 Reverting Bitmaps

When data blocks are added or reclaimed back during the recovery process the bitmaps have to be adjusted to keep track of free blocks. The PBITMAPS need not be restored back as they are never deleted. The physical blocks are backed up either in the deleted block list or in the old mapping lists in the V-TABLE. The physical blocks that are added in the current version are freed during the first and second phases of the recovery process. During the first phase, the previous version's data is restored from mapping list in the V-TABLE. At this time the physical blocks of the newer version are marked free in the PBITMAPS. When the pointers created in the current version are reverted back by deleting them in the second phase, the garbage collector frees both the physical and the logical blocks, only if it is the last incoming pointer to the destination block.

The LBITMAPS only have to be restored back for versioned blocks that have been deleted in the current version. While restoring the backed up mappings from the V-TABLE, SVSDS checks if the logical block is allocated in the LBITMAPS. If it is not allocated, SVSDS reallocates the deleted logical block by setting the corresponding bit in the LBITMAPS. The deleted non-versioned blocks need not be restored back. Previously, these blocks were moved to the deleted block list and were added back to the T-TABLE during the second phase of the recovery process.

4.5 Operation-based constraints

In addition to versioning data inside the disk, it is also important to protect certain blocks from being modified, overwritten, or deleted. SVSDS allows users to specify the types of operations that can be performed on a block, and the constraint manager enforces these constraints during block writes. SVSDS enforces two types of operation-based constraints: read-only and append-only.

The sequence of steps taken by the operation manager to mark a file as read-only or append-only is the same as marking a file to be versioned. The steps for marking a file to be versioned was described in Section 4.3. While marking a group of blocks, the first block (or the root block of the subtree) encountered in the breadth first search is treated differently to accommodate special file system updates. For example, file systems under UNIX support three timestamps: access time (atime), modification time (mtime), and creation time (ctime). When data from a file is read, its atime is updated in the file's inode. Similarly, when the file is modified, its mtime and ctime are updated in its inode. To accommodate atime, mtime, and ctime updates on the first block, the constraint manager distinguishes the first block by adding a special meta-data block flag in the T-TABLE for the block. SVSDS disallows deletion of blocks marked as read-only or append-only constraints. MARK_READ_ONLY and MARK_APPEND_ONLY are the two new APIs that have been added to the disk for applications to specify the operation-based constraints on blocks stored inside the disk. These APIs are described in Table 1.

Read-only constraint. The read-only operation-based 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 are not modified by an intruder or any other malware application. Since SVSDS does not have information about the file system data structures, atime updates cannot be distinguished from regular block writes using pointer information. SVSDS neglects (or disallows) the atime updates on read-only blocks, as they do not change the integrity of the file. Note that the read-only constraint can also be applied to files that are rarely updated (such as binaries). When such files have to be updated, the read-only constraint can be removed and set back again by the administrator through the secure disk interface.

Append-only constraint. Log files serve as an important resource for intrusion analysis and statistics collection. The results of the intrusion analysis is heavily dependent on the integrity of the log files. The operation-based constraints implemented by SVSDS can be used to protect log files from being overwritten or deleted by intruders.

SVSDS allows marking any subtree in the pointer chain as "append-only". During a write to a block in an append-only subtree, the operation manager allows it only if the modification is to change trailing zeroes to non-zeroes values. SVSDS checks the difference between the original and the new contents to verify that data is only being appended, and not overwritten. To improve the performance, the operation manager caches the append-only blocks when they are written to the disk to avoid reading the original contents of block from the disk during comparison. If a block is not present in the cache, the constraint manager reads the block and adds it to the cache before processing the write request. To speed up comparisons, the operation manager also stores the offsets of end of data inside the append-only blocks. The newly written data is compared with the cached data until the stored offsets.

When data is appended to the log file, the atime and the mtime are also updated in the inode block of the file by the file system. As a result, the first block of the append-only block is overwritten with every update to the file. As mentioned earlier, SVSDS does not have the information about the file system data structures. Hence, SVSDS permits the first block of the append-only files to be overwritten by the file system.

SVSDS does not have information about how file systems organize its directory data. Hence, enforcing append-only constraints on directories will only work iff the new directory entries are added after the existing entries. This also ensures that files in directories marked as append-only cannot be deleted. This would help in preventing malicious users from deleting a file and creating a symlink to a new file (for example, an attacker can no longer unlink a critical file like */etc/passwd*, and then just creates a new file in its place).

4.6 Issues

In this section, we talk about some of the issues with SVSDS. First we talk about the file system consistency after reverting back to a previous version inside the disk. We then talk about the need for a special port on the disk to provide secure communication. Finally, we talk about Denial of Service (DoS) attacks and possible solutions to overcome them.

Consistency Although TSDs understand a limited amount of file system semantics through pointers, they are still oblivious to the exact format of file system-specific meta-data and hence it cannot revert the state that is consistent in the viewpoint of specific file systems. 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 to SVSDS to ensure that disk-level pointers are consistent with file system pointers.

Administrative Interfaces To prevent unauthorized users from reverting versions inside the disk, SVSDS should have a special hardware interface through which an administrator can log in and revert back versions. This port can also be used for setting the checkpoint frequency.

Supporting Encryption File Systems Encryption File systems (EFS) can run on top of SVSDS with minimal modifications. SVSDS only requires EFS to use TSD's API for block allocation and notifying pointer relationship to the disk. The append-only operation-based constraint would not work for EFS as end of block cannot be detected if blocks are encrypted. If encryption keys are changed across versions and if the administrator reverts back to a previous version, the decryption of the file would no longer work. One possible solution is to change the encryption keys of files after a capability based authentication upon which SVSDS would decrypt all the older versions and re-encrypt them with the newly provided keys. The disadvantage with this approach is that the versioned blocks need to be decrypted and re-encrypted when the keys are changed.

DoS Attacks SVSDS is vulnerable to denial of service attacks. There are three issues to be handled: (1) blocks that are marked for versioning could be repeatedly overwritten; (2) lots of bogus files could be created to delete old versions, and (3) versioned files could be deleted and recreated again preventing subsequent modifications to files from being versioned inside the disk. To counter attacks of type 1, SVSDS can throttle writes to files that are versioned very frequently. An alternative solution to this problem would be to exponentially increase the versioning interval of the particular file / directory that is being constantly overwritten resulting in fewer number of versions for the file. As with most of the denial of service attacks there is no perfect solution to attack of type 2. One possible solution would be to stop further writes to the disk, until some of the space used up by older versions, are freed up by the administrator through the administrative interface. The downside of this approach is that the disk effectively becomes read-only till the administrator frees up some space. Type 3 attacks are not that serious as versioned files are always backed up when they are deleted. One possible solution to prevent versioned files from being deleted is to add *no-delete* flag on the inode block of the file. This flag would be checked by SVSDS along with other operation-based constraints before deleting/modifying the block. The downside of this approach is that normal users can no longer delete versioned files that have been marked as *no-delete*. The administrator has to explicitly delete this flag on the *no-delete* files.

5 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 5 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 from the file system, 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.

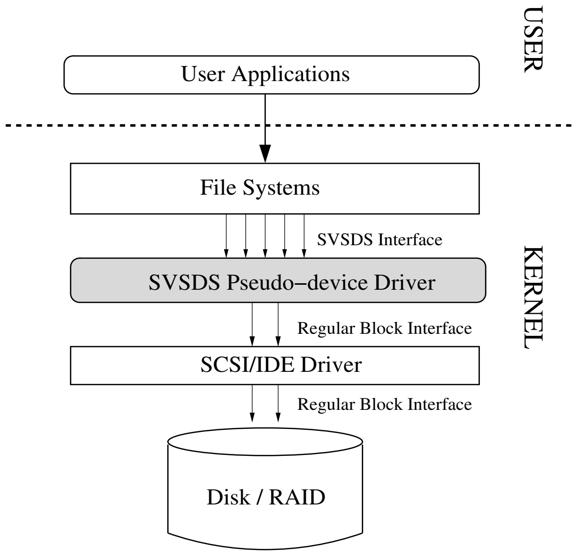


Figure 5: Prototype Implementation of SVSDS

In the current implementation we maintain all hash tables (V-TABLE, T-TABLE, P-TABLE, and D-TABLE) as in-memory data structures. As these hash tables only have small space requirements, they can be persistently stored in a portion of the NVRAM inside the disk. This helps SVSDS to avoid disk I/O for reading these tables.

The read and write requests from file systems reach SVSDS through the Block IO (BIO) layer in the Linux kernel. The BIO layer issues I/O requests with the destination block number, callback function (BI_END_IO), and the buffers for data transfer, embedded inside the BIO data structure. To redirect the block requests from SVSDS to the underlying disk, we add a new data structure (BACKUP_BIO_DATA). This structure stores the destination block number, BI_END_IO, and BI_PRIVATE of the BIO data structure. The BI_PRIVATE field is used by the owner of the BIO request to store private information. As I/O request are by default asynchronous in the Linux kernel, we stored the original contents of the BIO data structures by replacing the value stored inside BI_PRIVATE to point to our BACKUP_BIO_DATA data structure. When I/O requests reach SVSDS, we replace the destination block number, BI_END_IO, and BI_PRIVATE in the BIO data structure with the mapped physical block from the T-TABLE, our callback function (SVSDS_END_IO), and the BACKUP_BIO_DATA respectively. Once the I/O request is completed, the control reaches our SVSDS_END_IO function. In this function, we restore back the original block number and BI_PRIVATE information from the BACKUP_BIO_DATA data structure. We then call the BI_END_IO function stored in the BACKUP_BIO_DATA data structure, to notify the BIO layer that the I/O request is now complete.

We did not make any design changes to the existing Ext2TSD file system to support SVSDS. The Ext2TSD is a modified version of the Ext2 file system that notifies the pointer relationship to the file system through the TSD disk APIs. To enable users to select files and directories for versioning or enforcing operation-based constraints, we have added three ioctls namely: VERSION_FILE, MARK_FILE_READONLY, and MARK_FILE_APPENDONLY to the Ext2TSD file system. All three ioctls take a file descriptor as their argument, and gets the inode number from the in-memory inode data structure. Once the Ext2TSD file system has the inode number of the file, it finds the the logical block number that correspond to inode number of the file. Finally, we call the the corresponding disk primitive from the file system ioctl with logical block number of the inode as the argument. Inside the disk primitive we mark the file's blocks for versioning or enforcing operation-based constraint by performing a breadth first search on the P-TABLE.

6 Evaluation

We evaluated the performance of our prototype SVSDS using the Ext2TSD file system [16]. We ran general-purpose workloads on our prototype and compared them with unmodified Ext2 file system on a regular disk. This section is organized as follows: In Section 6.1, we talk about our test platform, configurations, and procedures. Section 6.2 analyzes the performance of the SVSDS framework for an I/O-intensive workload, Postmark [8]. In Sections 6.3 and 6.4 we analyze the performance on OpenSSH and kernel compile workloads respectively.

6.1 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.

Unless otherwise mentioned, the system time overheads were mainly caused by the hash table lookups on T-TABLE during the read and write operations and also due to P-TABLE lookups during CREATE_PTR and DELETE_PTR operations. 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 the wait time.

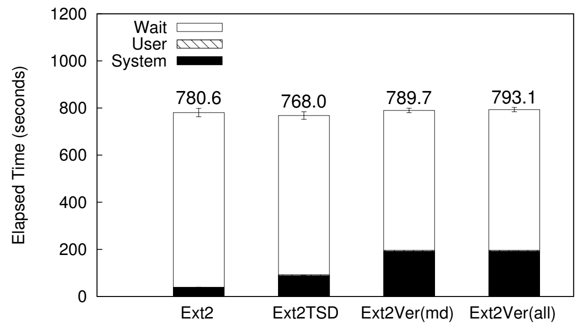
We have compared the overheads of SVSDS using Ext2TSD against Ext2 on a regular disk. We denote Ext2TSD on a SVSDS using the name Ext2Ver. The letters md and all are used to denote selective versioning of meta-data and all data respectively.

6.2 Postmark

Postmark [8] 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.

Figure 6 show the performance of Ex2TSD on SVSDS for Postmark with a versioning interval of 30 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. The average number of versions created during this benchmark is 27.

For Ext2TSD, system time is observed to be 1.1 times more, and wait time is 8% lesser that of Ext2. The



	Ext2	Ext2TSD	Ext2Ver(md)	Ext2Ver(all)
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%

Figure 6: Postmark results for SVSDS

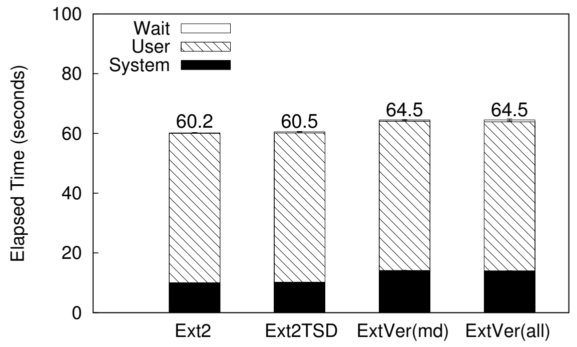
increase in the system time is because of the hash table lookups during CREATE_PTR and DELETE_PTR calls. The decrease in the wait time is because, Ext2TSD does not take into account future growth of files while allocating space for files. This decrease in wait time allowed Ext2TSD to perform slight better 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(md), elapsed time is observed to have no overhead, system time is 4 times more and wait time is 20% less than that of Ext2. The increase in system time is due to the additional hash table lookups to locate entries in the T-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 the random writes (i.e., writing inode block along with writing the newly allocated block) were converted to sequential writes due to copy-on-write in versioning.

For Ext2Ver(all), The system time is 4 times more and wait time is 20% less that of Ext2. The wait time in Ext2Ver(all) does not have any observable overhead over the wait time in Ext2Ver(md). Hence, it is not possible to explain for the slight increase in the wait time.

6.3 OpenSSH Compile

To show the space overheads of a typical program installer, 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 metadata and all data versioning in Ext2TSD on SVSDS for the `untar`, `configure`, and `make` stages combined. Since the entire benchmark completed in 60–65 seconds, we used a 2 second versioning interval to create more versions of blocks. On an average, 10 versions were created. This is because the pdflush deamon starts writing the modified file system blocks to disk after 30 seconds. As a result, the disk does not get any write request for blocks during the first 30 seconds of the OpenSSH Compile benchmark. The amount of data generated by this benchmark was 16MB. The results for the OpenSSH compilation are shown in Figure 7.



	Ext2	Ext2TSD	Ext2Ver(md)	Ext2Ver(all)
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%

Figure 7: OpenSSH Compile Results for SVSDS

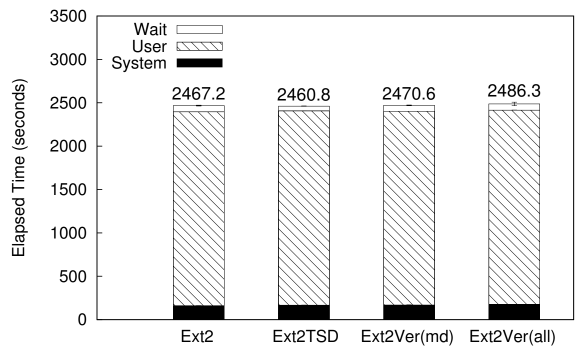
For Ext2TSD, we recorded a insignificant increase in elapsed time and system time, and a 108% increase in the wait time over Ext2. Since the elapsed and system times are similar, it is not possible to quantify for the increase in wait time.

For Ext2Ver(md), we recorded a 7% increase in elapsed time, and a 41% increase in system time over Ext2. The increase in system time overhead is due to the additional hash table lookups by SVL to remap the read and write requests. Ext2Ver(md) consumed 496KB of additional disk space to store the versions.

For Ext2Ver(all), we recorded a 7% increase in elapsed time, and a 39% increase in system time over Ext2. Ext2Ver(all) consumes 15MB of additional space to store the versions. The overhead of storing versions is 95%. From this benchmark, we can clearly see that the versioning all data inside the disk is not very useful, especially for program installers.

6.4 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. We used 30 second versioning interval and 78 versions were created during this benchmark. The results are shown in Figure 8.



	Ext2	Ext2TSD	Ext2Ver(md)	Ext2Ver(all)
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%

Figure 8: Kernel Compile results for SVSDS.

For Ext2TSD, elapsed time is observed to be the same, system time overhead is 4% lower and wait time is 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.

For Ext2Ver(md), elapsed time is observed to be the same, system time overhead is 5%, and wait time is lower by 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 is 51 MB.

For Ext2Ver(all), elapsed time is observed to be indistinguishable, system time overhead is 10% higher than that of Ext2. The increase in system time is due to the additional hash table lookups required for storing the mapping information in the V-TABLE. The space overhead of versioning all blocks is 181 MB.

7 Related Work

SVSDS borrows ideas from many of the previous works. The idea of versioning at the granularity of files has been explored in many file systems [6, 10, 12, 15, 19]. These file systems maintain previous versions of files primarily to help users to recover from their mistakes. The main advantage of SVSDS over these systems is that, it is decoupled from the client operating system. This helps in protecting the versioned data, even in the event of an intrusion or an operating system compromise. The virtualization of disk address space has been implemented in several systems [3, 7, 9, 13, 21]. For example, the Logical disk [3] separated 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 operation-based constraints in SVSDS is a scaled down version of access control mechanisms. We now compare and contrast SVSDS with other disk-level data protection systems: S4 [20], TRAP [23], and Peabody [7].

The Self-Securing Storage System (S4) is an object-based disk that internally audits all requests that arrive at the disk. It protects data in compromised systems by combining log-structuring with journal-based meta-data versioning to prevent intruders from tampering or permanently deleting the data stored on the disk. SVSDS on the other hand, is a block-based disk that protect data by transparently versioning blocks inside 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. As S4 is designed to aid in intrusion diagnosis and recovery, it does not provide any flexibility to users to version files (i.e, objects) inside the disk. In contrast, SVSDS allows users to select files and directories for versioning inside the disk. The disadvantage with S4 is that, it does not provide any protection mechanism to prevent modifications to stored data during intrusions and always depends on the versioned data to recover from intrusions. In contrast, SVSDS attempts to prevent modifications to stored data during intrusions by enforcing operation-based constraints on system and log files.

Timely Recovery to any Point-in-time (TRAP) is a disk array architecture that provides data recovery in three different modes. The three modes are: TRAP-1 that takes snapshots at periodic time intervals; TRAP-3 that 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. The disadvantage with this system is that, it cannot provide TRAP-2 (data protection at the file-level) as their block-based disk lacks semantic information about the data stored in the disk blocks. Hence, TRAP ends up versioning all 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.

Peabody is a network block storage device, that 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. This is done to improve the cache utilization and to reduce the total amount of storage space. Peabody versions data by maintaining write logs and transaction logs. The write logs stores the previous contents of blocks before they are overwritten, and the transaction logs contain information about when the block was written, location of the block, and the content hashes of the blocks. The disadvantage with this approach is that it cannot selectively versions blocks inside the disk.

8 Conclusions

Data protection against attackers with OS root privileges is fundamentally a hard problem. While there are numerous security mechanisms that can protect data under various threat scenarios, only very few of them can be effective when the OS is compromised. In view of the fact that it is virtually impossible to eliminate all vulnerabilities in the OS, it is useful to explore how best we can recover from damages once a vulnerability exploit has been detected. In this paper, we have taken this direction and explored how a disk-level recovery mechanism can be implemented, while still allowing flexible policies in tune with the higher-level abstractions of data. We have also shown how the disk system can enforce simple constraints that can effectively protect key executables and log files. Our solution that combines the advantages of a software and a hardware-level mechanism proves to be an effective choice against alternative methods. Our evaluation of our prototype implementation of SVSDS shows that performance overheads are negligible for normal user workloads.

Future Work . Our current design supports reverting the entire disk state to an older version. In future, we plan to work on supporting more fine-grained recovery policies to revert specific files or directories to their older versions. SVSDS in its current form, relies on the administrator to detect an intrusion and revert back to a previously known safe state. We plan to build a storage-based intrusion detection system [14] inside SVSDS. Our system would do better than the system developed by Pennington et al. [14] as we also have data dependencies conveyed through pointers. We also plan to explore more operation-based constraints that can be supported at the disk-level.

9 Acknowledgments

We like to thank the anonymous reviewers for their helpful comments. We thank Sean Callanan and Avishay Traeger for their feedback about the project. We would also like to thank the following people for their comments and suggestions on the work: Radu Sion, Rob Johnson, Radu Grosu, Alexander Mohr, and the members of our research group (File systems and Storage Lab at Stony Brook).

This work was partially made possible by NSF CAREER EIA-0133589 and NSF CCR-0310493 awards.

References

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

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

[7] 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.

[8] J. Katcher. PostMark: A new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[9] 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.

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

[11] M. Mesnier, G. R. Ganger, and E. Riedel. Object based storage. *IEEE Communications Magazine*, 41, August 2003. ieeexplore.ieee.org.

[12] 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.

[13] 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.

[14] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the 12th USENIX Security Symposium*, pages 137–152, Washington, DC, August 2003.

[15] 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.

[16] 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.

[17] 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.

[18] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 15–30, San Francisco, CA, March/April 2004. USENIX Association.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.