Design and Implementation of Views: Isolated Perspectives of a File System

Matthew W. Pagano Zachary N. J. Peterson
The Johns Hopkins University
Baltimore, Maryland, USA
{mpagano,zachary}@cs.jhu.edu

Abstract

We present Views, a file system architecture that provides isolation between system components for the purposes of access control, regulatory compliance, and sandboxing. Views allows for discrete I/O entities, such as users, groups, or processes, to have a logically complete yet fully isolated perspective (*view*) of the file system. This ensures that each entity's file system activities only modify that entity's view of the file system, but in a transparent fashion that does not limit or restrict the entity's functionality. Views can therefore be used to monitor system activity based on user accounts for access control (as required by federal regulations such as HIPAA), provide a reliable sandbox for arbitrary applications without inducing any noticeable loss in performance, and enable traditional snapshotting functionality by manipulating and transplanting views as snapshots in time. Views' architecture is designed to be file system independent, extremely easy to use and manage, and flexible in defining isolation and sharing polices. Our implementation of Views is built on ext3cow, which additionally provides versioning capabilities to all entities. Benchmarking results show that the performance of Views is nearly identical to other traditional file systems such as ext3.

1 Introduction

The need to reliably isolate, analyze, and potentially snapshot various file system states exists in many environments today. One particularly salient example is that of medical security, in which there has been a recent impetus to convert highly sensitive paper medical records into electronic format. The Health Insurance Portability and Accountability Act of 1996 (HIPAA) was written to develop standards for the normalization of individual health records and to encourage the use of electronic records pursuant to these goals. To protect the sensitive data these health records can hold from unauthorized modifications, HIPAA includes provisions that address the way patient medical records are accessed and stored. Specifically, HIPAA mandates the use of access control mechanisms designed to monitor the activity of individual system users, thereby protecting the privacy of patient information [12]. Fulfilling this mandate requires the ability to efficiently and reliably isolate and identify the actions of each individual user on a multi-purpose, multi-user workstation or server.

Similarly, the need for isolation between system components is paramount when testing applications or processes. Researchers often need to monitor the behavior of applications running on their systems such that: 1) they are guaranteed that they can record all of the application's behavior; 2) they can prevent any undesirable data modification; and 3) they can completely undo all of the application's actions without the threat of missing some covert or obscure behavior. This scenario is often found in research pertaining to malware, virtualization, honeypots and honeynets, software quality assurance, and system benchmarking and performance testing. In each, the researcher requires concrete assurance that she can dependably monitor all modifications made by a particular user or application. It is worth noting that each of these examples has its own set of requirements. In fields such as malware and honeypot research, the threat model describes an adversary who is actively endeavoring to subvert the safeguards implemented by the researcher. Any such solution must therefore be robustly and securely implemented at a low level to thwart such tampering, especially if the testing system is connected to legitimate systems. In software quality assurance and benchmarking, system speed and performance is a high priority in order to provide a realistic testing environment, so isolation strategies such as virtualization might not be acceptable.

Once isolation between system components is achieved, the researcher may need to save the state of a given component for development, offline analysis, future use/restarting from some point in time, or trans-

This work was supported in part by Independent Security Evaluators, Baltimore, Maryland, USA.

plant to a different environment. This is often the case in application and process checkpoint and recovery solutions, virtual machine analysis and migrations, and database management (especially if databases are to be operated on and modified in parallel). Each scenario requires some means of isolating and snapshotting a certain system component in a comprehensive, portable manner.

Regardless of the environment and chosen solution, what is needed in each case is a method of reliably monitoring and controlling file system access, as well as preventing and undoing unauthorized data modification. In traditional file systems, such as ext3 or NTFS, unauthorized modifications to data are commonly prevented through kernel-enforced permissions or access control lists. This is usually implemented by storing metadata within inodes that lists the users who may access the data. However, even in cases where modifications are authorized, unforeseen problems may arise. This is because all users, groups, and processes share the same namespace; that is, they have the same *view* of the data. For most file system applications, this is preferential because it enables data sharing (*e.g.* applications and configuration files) and avoids wasting disk space with many copies of the same data. The ease with which data can be shared, though, can be detrimental when dealing with sensitive information, (*e.g.* accidentally copying a health record to the `/tmp` directory may make it accessible to all users). Another obvious drawback to this architecture is that any changes to application data, be they benign or malevolent, are seen by all users. For example, a trojan horse unintentionally installed by a single user could affect all who use that system.

We present the design and implementation of Views, a system for providing isolated yet logically complete perspectives of a file system. These isolated perspectives can be used to achieve the needs of the scenarios described thus far in a reliable, efficient manner. The primary design goal of Views is to provide every *entity* in the system with its own isolated *view* of the file system. We define an entity to be anything that can perform I/O within the file system. An entity can be a user, group, process, family of processes, *etc*. A view is a logically complete perspective of the file system; every entity sees the entire file system hierarchy. Any data modifications made by an entity are, by default, only viewable by that entity. In one extreme example, a user can perform an `rm -rf /` operation without affecting any other user. Views features a robust policy management mechanism, allowing namespaces to be customized for every entity. Providing data isolation at the file system level within the kernel allows for increased security (as opposed to implementing the isolation in user space), ease of use for the administrator and system users as all of the logic is pre-configured in the file system (as opposed to a revision control system that requires additional configuration and management), resource efficiency and low overhead as only the file system blocks that are modified between views are copied and rewritten, as well as a high degree of confidence in the isolation since it is performed deterministically at the file system level (as opposed to using heuristics that may not be 100% accurate). Furthermore, Views supports temporal policies that dictate how long a view is valid. For example, an entity that is suspected of misbehaving may be analyzed and all modifications it has made to the file system easily purged.

To minimize performance and storage overheads, Views uses a copy-on-write mechanism to store only the blocks that have been modified, although copy-on-write functionality is not a requirement for our implementation. Because Views is implemented in the kernel, it provides throughput performance that compares well to an unmodified file system, as we discuss in Section 5.

Our overall design is largely file system independent in that it relies only upon Linux's built-in Extended Attribute interface [2]. As this is currently the only requirement of Views, Views can be implemented on a wide variety of file systems. Views does not modify any kernel interfaces, requires no special software, and is completely transparent to the user. Since Views is implemented in the kernel, it obtains an additional protection mechanism when compared to a user space implementation. Our design is indiscriminate of backup and restore techniques and allows for easy integration into information life-cycle management systems.

We have implemented Views in the ext3cow file system [7]. Ext3cow is a freely-available, open-source versioning file system (available at: `www.ext3cow.com`) that is designed to meet other regulatory compliance requirements, such as fine-grained data authenticity and encryption.

2 Related Work

Views is designed to incorporate three prevailing techniques in current operating systems research: kernel modification to improve security, isolation between I/O entities via distinct namespaces, and versioning of files to achieve regulatory compliance. There has been significant research in each of these three areas. Security-Enhanced Linux (SELinux) [4] is a system created by the National Security Agency to introduce a flexible mandatory access control (MAC) architecture to Linux. As with Views, SELinux modifies the Linux kernel by implementing the native extended attribute structure to assign labels to the entities of a file system. Using these labels, SELinux can determine what are permissible interactions between entities based on customizable security policies.

Mounts [3] is a system designed by Al Viro that uses the `mount` operation to create isolated file system namespaces for each user. Because the isolation in the original design was too restrictive, Mounts has been further developed to include shared mounts when data needs to be shared in a two-way mode, and slave mounts when data needs to be shared in a one-way mode. Users can define the location and nature of each mounted file system using pluggable authentication modules and scripts for system startup and user creation. Lastly, versioning file systems have been developed that provide simple procedures and effective means of revision control [5, 9, 10] and regulatory compliance [7]. These types of systems generally support versioning mechanisms that allow users to easily create and review versions of a given file.

SELinux is effective at preventing unwarranted exposure of data within a system, but its vast policy configurations can be labor-intensive to administer. This can potentially impede vital functionality if it is mismanaged. Furthermore, the system still has only one file system, so no namespace isolation is achieved. If a policy is misconfigured, data can be left exposed as all entities still operate within the same file system. Views can be used to extend and simplify the functionality of an SELinux-enabled kernel by creating distinct yet complete file systems for each entity. This will guarantee by default that an entity cannot access the file system of another entity, but that each entity retains full access to the data it needs to perform its functions. This level of protection is provided automatically and with no further configuration needed as soon as the file system is created.

Unlike SELinux, Mounts does achieve a level of file system isolation. However, current documentation on Mounts focuses mainly on isolation based on user accounts. Moreover, the isolation in Mounts is created by calling the mount operation from user space. In contrast, Views is implemented on an open-ended framework that allows isolation for any entity that performs I/O on the system, such as users, groups, individual processes, families of processes, *etc*. Views also achieves a finer and more robust degree of isolation at a lower level by creating inode hierarchies in the file system on disk. By using defined policies, the administrator can define a customized view as well as set a time-to-live for any entity, so views are as temporal or permanent as the administrator chooses. Views is also designed to be independent of any versioning model and capable of being implemented in a wide variety of system designs.

In Ventana [8], Pfaff *et al.* propose a system that provides centralized management of virtual machines distributed across a network. Ventana allows users on separate workstations to set up private or shared branches of a file system, create virtual machines using a particular view of a file system, version these views, and govern the corresponding access controls. Views differs from Ventana in that the isolation and branching of separate views results from creating inode hierarchies at a low-level on disk, as opposed to using virtualized machines. This enables Views to offer almost native I/O speeds, minimize overhead, and provide fine-grained isolation of entities. In addition, because Views is implemented at the kernel level, there is no need for additional configuration or management once the file system is installed.

Finally, virtual machines have become a popular way to isolate I/O entities. A wide variety of virtual machine products are currently available for both high-end servers and end-user machines. Virtual machines have allowed administrators to provide both isolated disk images, which provide total data isolation, and copy-on-write functionality [13], which provides block-level isolation. Managing a virtual machine environment, however, is imperfect. Virtual machines can add complexity to system management and licensing, can add a layer of abstraction to devices that may limit features and performance, and can be difficult to scale.

3 Views Architecture

Views is built on top of the ext3cow file system [7]. Ext3cow is a variant of the ext3 file system [1] that provides file versioning accessed through a time-shifting interface, authenticated encryption, and other technologies designed to meet regulatory compliance. Ext3cow is an effective platform for a Views implementation as it provides versioning capabilities and supports Linux's Extended Attributes API (the only requirement of Views). Extended Attributes are a file system facility available in certain file systems such as ext3cow that provide a means to associate additional metadata with files and directories. Extended Attributes, often abbreviated as xattr, have been used for implementing access control lists [2] and security tags in SELinux [4].

Figure 1 shows an example of isolated views for two users. Each user has a logically complete view of the file system, so there is no loss of functionality for any of the system's entities. For example, both users currently share a view of the `bin/` directory, thus providing both users with all system utilities present in this folder. If one of the users (say the `matt` user) modifies a file in the `bin/` directory, that modified file becomes part of the view belonging to `matt`. However, this file modification is not made to the corresponding file in the `bin/` directory of the view belonging to the `zachary` user, as happens in traditional file systems that provide only one namespace for all users.

As depicted in Figure 1, suppose `matt` creates a directory `foo/`. This directory then exists only in his view. `zachary` is completely unaware that the `foo/` folder

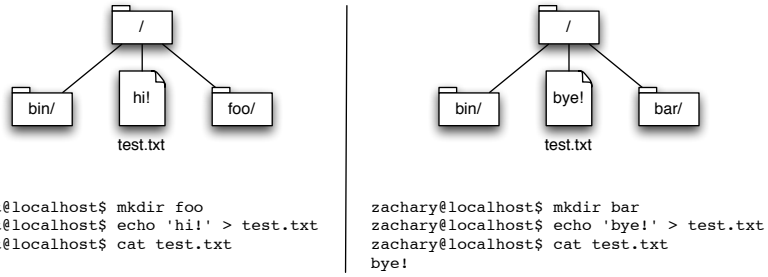

Figure 1: An example of Views' isolation model.

exists in the file system. Next, `zachary` creates a directory named `bar/`. Once again, this directory exists only in the view of `zachary` and is fully unknown to `matt`. Lastly, both users create a file with the name `test.txt`, but while the same name appears in both views, they represent different files with different contents.

3.1 File Views

In most file systems, every file has a single inode that contains that file's metadata, including modification time, permissions, and pointers to the file's data blocks. If the file system supports Extended Attributes, the inode also contains space or a block pointer in which to store Extended Attribute information – typically a name-value pair. This Extended Attributes feature is the only requirement for a Views implementation, although in theory the data stored in the Extended Attributes for Views can be stored elsewhere in the inode if there is sufficient free space.

When a Views file system is created by the root user, every original file is represented by a single inode. We call this base file system the *master view* and each inode a *master inode*. The master inode may be changed over time, for system maintenance for example (see Section 3.3), but it represents the basis from which all additional views are created.

When an entity makes an initial modification to a file in the master view, a new view for that entity is created. To create a new view, a new *entity inode* is allocated, the metadata from the master inode is copied into the new entity inode, and the entity inode is linked to the master inode with an extended attribute (see Figure 2). Views for a file are represented by entity-inode pairs stored as extended attributes. When the file is modified by an entity, all modifications are recorded in the corresponding entity inode, isolating the changes to only that entity's view.

As shown in Figure 2, suppose that a Views file system has been created (thus a master view is in place). Further suppose that `matt` and `zachary` are two users on this system. Both `matt` and `zachary` access a master file whose metadata is captured in a master inode of inode number (`i_ino`) 211. If `matt` or `zachary` accesses this file in a read-only fashion (checked using the flags of the `nameidata` data structure associated with the lookup of that file), they are given the master inode. No entity inode is created for them in read-only accesses to conserve inodes in use and to avoid unnecessary system overhead (*e.g.* a Views file system creating entity inodes for each inode in a large directory every time an entity runs an `ls` command). Moreover, the corresponding directory entry (dentry) structures that are created for `matt` and `zachary` linking them to the master inode are assigned a read-only metadata character. This is done to prevent future write accesses made to this file by these (non-root) entities from being saved to the master inode. Further discussion on how we modify the dentry system is given in Section 3.2.

Now suppose that `matt` requests write access to the master file corresponding to master inode 211. An extended attribute name-value pair is then created for `matt` in master inode 211. The name in the name-value pair is the user ID of `matt`, which as shown in Figure 2 is 1001. The value in the name-value pair is the inode number of the entity inode that is created for `matt` for this file (721). If the entity inode of this file in `matt`'s entity view needs to be retrieved later, the names of the extended attribute name-value pairs of master inode 211 can be searched for `matt`'s user ID (1001). If this user ID is found, the value of that name-value pair yields the inode number of the proper entity inode. A similar situation transpires if `zachary` modifies the master file corresponding to master inode 211: an extended attribute name-value pair is added to master inode 211 in which the name is `zachary`'s user ID (1002) and the value is the inode number of the entity inode created for `zachary` (429).

We improve upon ext3cow's copy-on-write model to achieve Views. All data modifications made to an entity inode are copy-on-written: a new block is allocated

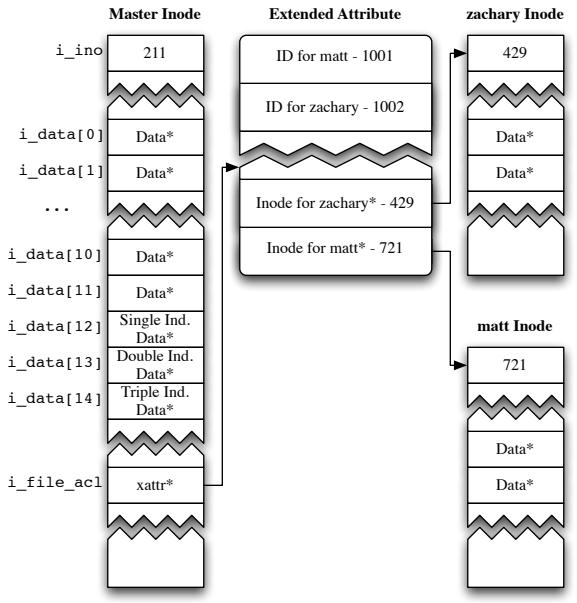

Figure 2: Architecture of Views using extended attributes.

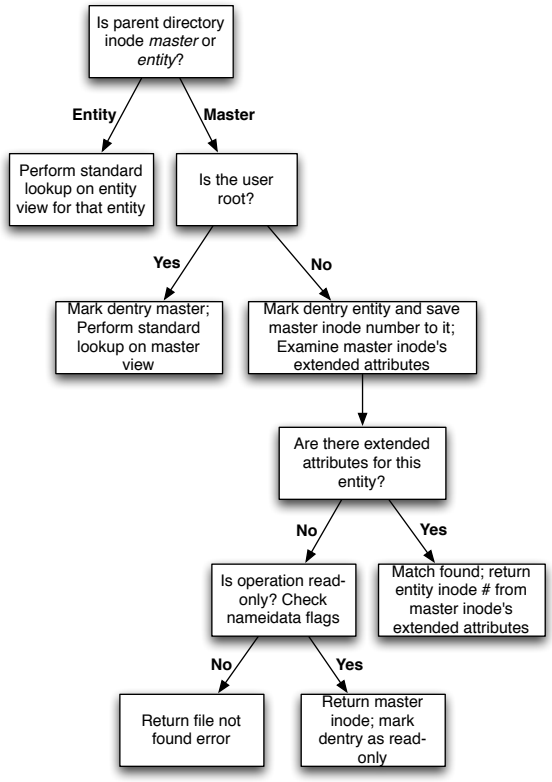

Figure 3: The lookup process on a Views file system.

for the modification and the old blocks are preserved as part of the master view. When an entity modifies a data block, the new block is stored only in that entity's inode, and therefore viewable only by that entity. Because new blocks are allocated only for the individual blocks that change between master and entity views, Views is highly efficient with respect to storage and performance, as demonstrated in Section 5.

When an entity creates a file (thus the file does not currently exist in the master view), only a single inode is allocated. Because the name and inode are not accessible by any other entity and the file is not part of the master view, no master inode or extended attribute is created or needed.

3.2 Directories and Naming

In addition to having isolated data modifications, entities may also modify the file system namespace in isolation. This is easily accomplished in Views as directories are simply inodes whose data are directory entries. Name modifications (additions, removals, or changes to a name) by an entity to a directory that has a master inode (*i.e.* part of the master view) creates an entity inode and extended attribute entry, similar to what is shown in Figure 2. Likewise, any name modification to a directory created by an entity (thus that directory exists only in that entity's view) results in only a single directory inode.

To perform a lookup of a name in Views, the file system reads the inode of the parent directory. The inode may be a master inode if the parent directory is part of the master view, or an entity inode if the parent directory is part of an entity view. If the parent directory is an entity inode, a normal lookup is performed on the entity view for that entity. If the parent directory is a master inode and the entity performing the lookup is the root user, a normal lookup is performed on the master view for the root user. If the parent directory is a master inode and the entity performing the lookup is not the root user, the file system searches the extended attributes of the master inode of corresponding name for an identifier that matches that entity. If a matching identifier is found, the entity inode corresponding to that identifier in the master inode's extended attribute name-value pair is returned. If no identifier is found and the operation is **not** read-only (checked using the flags of the `nameidata` data structure associated with the lookup), the file system returns a file not found error. Conversely, if no identifier is found but the operations **is** read-only, the master inode of the corresponding name is returned for purposes of efficiency, as discussed in Section 3.1. In this case, the corresponding directory entry (dentry) structure that is created linking that entity to the master inode is marked as read-only, as shown in Figure 4. This is done to pre-

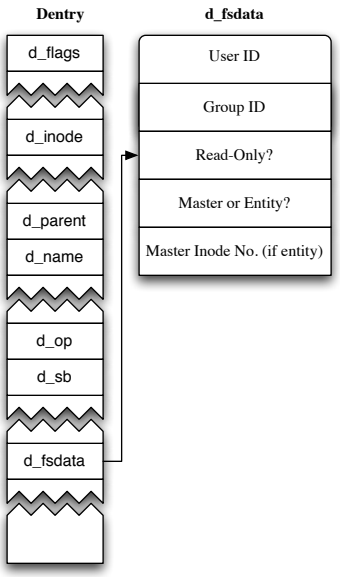

Figure 4: Visual representation of how the directory entry (dentry) structure is modified for Views during the lookup process. Specifically, the `d_fsdata` parameter is modified to include the information shown on the right. This information is added to the `d_fsdata` parameter as soon as the dentry structure associated with the lookup is passed to the `ext3cow_lookup` function for on-disk inode searches.

vent future write accesses made by that (non-root) entity to that file from being performed on the master inode. The dentry structure for this lookup is also marked as either master or entity depending upon whether the corresponding inode is in the master view or an entity view, respectively. This process is shown in Figure 3.

Because the same name can exist in different views but represent different data (as shown with the `test.txt` file in Figure 1), further modifications to the dentry system are necessary. This is achieved by writing the following items to the filesystem-specific data parameter, `d_fsdata`, of a given dentry structure as soon as the dentry is passed to the `ext3cow_lookup` function for on-disk inode searches, as shown in Figure 4.

- The user ID of the entity that owns the dentry.
- The group ID of the entity that owns the dentry.
- Whether the dentry is read-only or not.
- Whether the dentry points to a master or entity inode.
- If the dentry points to an entity inode, the inode number of the master inode to which that entity inode belongs.

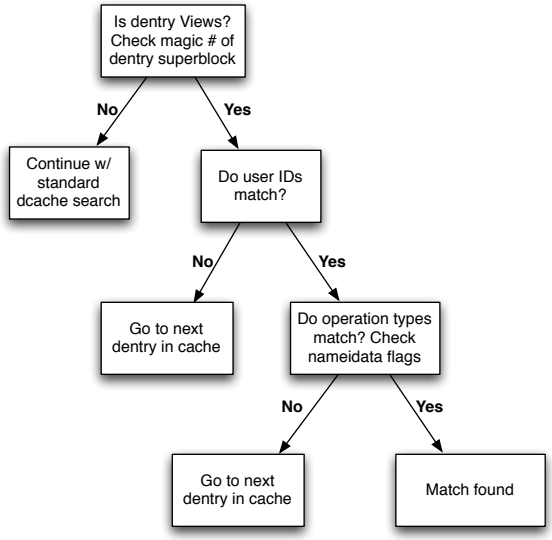

Figure 5: Our modified dentry cache (dcache) search process.

The dentry cache (dcache) search is then modified as follows. First, it is determined whether the cached dentry being analyzed belongs to a Views partition. This is checked using the magic number of the cached dentry's superblock. Next, a comparison is performed between the IDs of the entity searching the cache and the IDs stored in the `d_fsdata` parameter of the cached dentry. A comparison is also made between the operation type (*i.e.* read-only or not read-only) stored in in the `d_fsdata` parameter of the cached dentry and the operation type of that particular lookup (checked using the flags of the `nameidata` data structure associated with the lookup). If the IDs and the operation types both match, our modified dcache search returns the valid dentry. Otherwise, our search moves on to the next dentry in the cache in search of a match. This procedure is shown in Figure 5.

3.3 Policies of Views

The root user is a special entity in Views. In particular, the root user controls the master view. By default, the root user sees only the master view, but has the ability to see all entity views through an additional API or by using the `su`, or `switch user`, command. By using `su` to switch user accounts to a different user, the root user's view easily and automatically changes to the entity view of that user. This is demonstrated in Figure 6, which shows the root user switching into the entity views

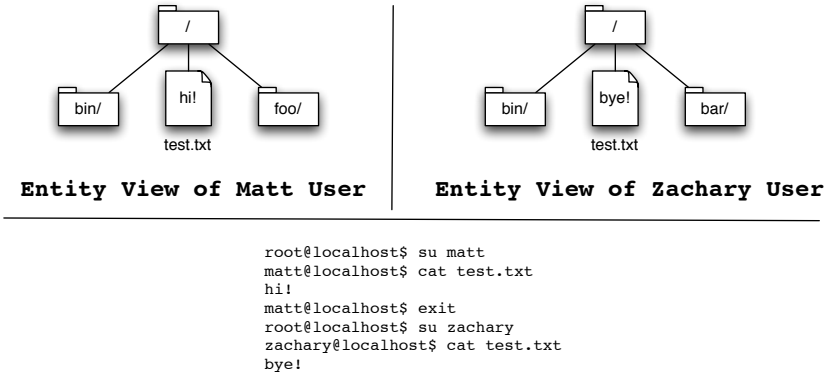

Figure 6: Demonstration of how the root user can easily switch into the entity views of each user by invoking the `su` command. This assumes the same file system as shown in Figure 1.

of the `matt` and `zachary` user accounts and viewing the data of each's entity inodes (assuming the file system described in Figure 1). This functionality can be used by the root user to update/upgrade/maintain the entity views of other users on the system. For example, if an application such as SSH is reported to contain a vulnerability for which a patch has been recently released, the root user can use `su` to switch into the entity views of all users on the system to install the patch on all views. This process can be automated with the use of a simple script. Future work for Views includes development on an additional API that can further automate this process.

When the root user adds a file or directory, it automatically becomes part of the master view for all future entities. This allows the root user, for example, to perform system-wide modifications and maintenance for the master view and for all future views. It is important to note that additions, removals, or changes to the master view do not by default affect an entity's pre-existing view. Said another way, if an entity has modified a view for a file and the master view changes above it, the entity's view is unchanged. Only future views see the master view changes to ensure view consistency for all entities.

3.4 Views and Versioning

In addition to access controls based on individual user activity, federal legislation such as HIPAA, Sarbanes-Oxley, and HITECH, requires an auditable trail of changes that are accessible in real time. This requirement mandates the use of versioning in a file system. Views fits well into ext3cow's versioning model, shown in Figure 7. In ext3cow, versioning is achieved by creating a new inode for every version, copying all the metadata information to the new inode (including data block pointers), and updating the fields only when data is changed in a copy-on-write fashion. By copying the data block pointers to the new inode, the new version continues to use the same data blocks until modifications are made. This is done to minimize overhead, as opposed to copying all data blocks for each new version regardless of whether the user actually makes any modifications to the file. If the user does make modifications, new data blocks are allocated in a copy-on-write fashion, but only for the data blocks that are actually modified in order to increase efficiency.

Modifications to an inode's data blocks are tracked via a copy-on-write bitmap named `i_cowbitmap`. Specifically, `i_cowbitmap` contains a bit for each data block belonging to the inode that states either that the block should be updated in place, or that a new block should be allocated in the event of an update. In the case of the latter, the old block is permanently retained for previous versions in the version chain. Versions of a file are implemented by chaining inodes together via a linked list, where each inode represents a version. Each inode contains a pointer named `i_nextinode` that points to the next inode in the versioning chain. Inodes are given a time-stamp named `i_epochnumber` that identifies the period of time to which that version belongs. `i_epochnumber` is updated if the data of that inode is modified.

To find a particular version, users perform a lookup on a file name at a given time. The file system traverses the inode chain for that file until it finds an inode that has the desired time-stamp. This in turn generates a point-in-time view of a file.

Implementing Views in ext3cow allows an entity to have an isolated view of a file as well as enjoy versioning capabilities. Every master and entity inode is capable of maintaining a version chain without affecting any other entity's view.

Entity inodes that do not have a corresponding master

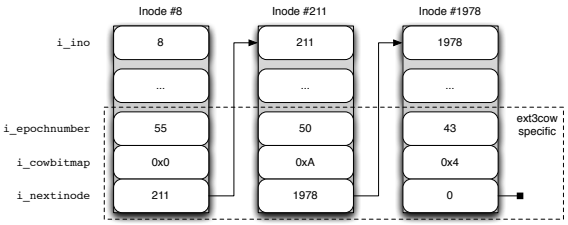

Figure 7: Visual representation of the inode structural changes made by ext3cow to support versioning. Because Views is built on ext3cow, Views also enjoys this level of versioning.

inode (*i.e.* files that are created by an entity that did not previously exist in the master view) are versioned as any file is versioned in ext3cow (See Peterson [7]). For files that are part of the master view (*i.e.* have a master inode), versioning becomes more interesting. All entities, including the root user, are entitled to versioning and no modification by any entity should affect another. To accomplish this, versioning within Views requires a parallel metadata structure. This is achieved because when new versions of a file are created, the inode metadata that is copied from the older inode to the new inode includes the extended attributes of the older inode. As a result, the new version of a master file becomes aware of all entity inodes belonging to the older version of that master file. This is necessary to ensure the continuity of entity views as version-snapshots are taken. Moreover, by examining a previous version of a master file at a particular time in the past, the root user can also observe all entity views of the master file at that time.

Figure 8 shows the metadata structure of a versioned view. We start with version 8 of the master inode, which has no entity views. At some point later, a version-snapshot is taken and the root user modifies the master inode, resulting in a new master inode (Version 9) chained to its previous versions. Additionally, Entity 1 creates a view of the file, resulting in her own entity inode (Entity 1 Version 1). Later, another version-snapshot is taken and the root user again modifies the master inode, leading to another inode in the version chain (Version 10). As discussed, the extended attributes of the master inode Version 9 are among the metadata copied into the master inode Version 10, so the master inode Version 10 becomes aware of all current entity views. In this example, this is entity inode Entity 1 Version 1. Independently, Entity 1 makes changes to her view of the file (Entity 1 Version 1), and because these changes occur after a version-snapshot, a new inode is allocated for her (Entity 1 Version 2). Lastly, a separate entity (Entity 2) creates a new view of master inode Version 10, and thus an entity inode is created for her (Entity 2 Version 1). It is important to remember that master inode Version 10 and entity inode Entity 1 Version 2 may store completely different data. Both versions exist because both the root user and Entity 1 performed modifications to their respective views after a version-snapshot.

The addition, removal, and modification of names over time are also supported by Views. Names in ext3cow are scoped to times using additional metadata in the directory entry (dirent) structure. Every dirent contains birth and death time-stamps, defining a period of time for which that name is valid. This allows file names to be added, removed, and re-added over time while maintaining past data. Should an entity add or remove a name, a new entity inode is created and the corresponding dirent is modified. Because Views provides individual entity directory inodes, all names scope properly, even if they are used by other entities or re-used by the same entity.

4 Use Cases

We show the benefits and utility of Views in the following use cases.

4.1 Regulatory Compliance

As discussed in Section 1, Views has strong application within regulatory compliance standards such as HIPAA and HITECH. The National Institute of Standards and Technology (NIST) released a special publication in October 2008 that offered guidance on implementing the measures required by the HIPAA Security Rule [6]. Many of the technical safeguards described in this document are user-based access controls that must be deployed on all electronic devices that handle medical records. These controls, as well as how Views can be implemented to facilitate these controls, are detailed below.

Ensure that system activity can be traced to a specific user. Views completely isolates all file system modifications of each user into their own individual views that are accessible only by that user and the root user. Consequently, Views allows administrators with root-user privileges to easily monitor all file system activities of each user. Since Views is implemented within the kernel at the file system level, users of the system are unable to subvert the isolation mechanisms unless they can successfully rewrite the file system in the kernel. Future work for Views will allow an administrator with root-user credentials to easily print and organize a list of all file system changes and the associated metadata made by each user (see Section 6 for more details).

Specify requirements for access control that are both feasible and cost-effective for implementation.

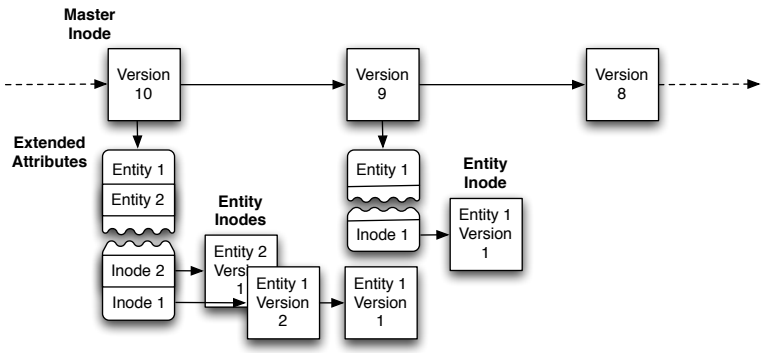

Figure 8: The metadata to support versioning of Views.

Views provides an immediate form of isolated yet logically complete views of a file system that can be readily implemented to achieve user-based access controls. Because these views are logically complete, no user or application is denied any of its functionality. In addition, Views is open-source and freely available. Installing Views is identical to installing the ext3 file system, and all of the protections of Views are provided immediately by default with no further action by the administrator. Section 5 shows that there is almost no penalty of performance in using Views over ext3.

Implement a mechanism to encrypt and decrypt EPHI [electronic protected health information]. Views is built on top of the ext3cow operating system and thus enjoys all of the benefits of ext3cow by default. As discussed in previous work on ext3cow [7], ext3cow offers full support for authenticated encryption.

Implement Audit/System Activity Review Process / Activate necessary audit system. Views retains all of the functionality of ext3cow. As discussed in Section 3.4, ext3cow (and thus Views) satisfies the auditing requirements of HIPAA through its versioning capabilities. In addition, Views provides administrators with a simple mechanism of isolating all changes made to the file system by each individual user. Auditing all file system modifications made by any user is therefore an easy and automatic process with Views.

Implement electronic mechanisms to corroborate that EPHI [electronic protected health information] has not been altered or destroyed in an unauthorized manner [*i.e.* integrity]. Views ensures that all changes made by a user are restricted to the file system view of that user only. All other versions of a file are unmodified in the views of all other users. As a result, users are unable to modify the master view, which is always available to the root user for review and for comparisons with each of the users' views to evaluate their changes. Important medical data that should not be erased can be saved to the master view or to a backup user account created by the administrator for this purpose. This ensures the integrity of all data stored on the system.

Beyond regulatory compliance, the ability of Views to isolate file system modifications by user account is also helpful with respect to system security. For example, malware that is installed (intentionally or unintentionally) by one user does not affect any other user on the system. In addition, that malware can be easily and completely removed from the system by deleting that user's view. No further trace of the malware remains on the system after doing so. This is in stark contrast to the high overhead and lack of guaranteed full protection of most anti-malware solutions, which often rely on heuristic-based approaches to detect and/or prevent suspicious software behavior. This is a limited approach because heuristics by nature are not guaranteed to deterministically detect all forms of suspicious activity. Furthermore, heuristic-based applications usually operate in user space, and are thus easier for an attacker to subvert.

4.2 Application Testing and Sandboxing

Researchers often need to test the behavior of software without permanently committing the software's actions to the system on which it is run. This can be the case with software that is still in development; software that requires multiple iterations of testing on clean systems that have never run the software before; malware that needs to be analyzed for signature generation, but obviously must be fully and reliably cleaned from the system after being run; software that a user is evaluating but is unsure if she wants to have permanently installed on her system; *etc*. In essence, software whose effects must be analyzed but not retained.

Solutions chosen to meet these goals must be reliable, secure, scalable, easy to use, and efficient with respect to

resources and overhead. While existing solutions are effective to a certain extent, there is room for improvement, as discussed in Section 2. For example, virtualization is a very common strategy in this realm. However, virtualization always taxes the system's resources and thus suffers from a performance loss. There is also overhead in configuring a virtual machine (VM) for use: the researcher must install virtualization software, install a guest OS, configure the guest OS for use, maintain the guest OS, manage multiple snapshots, *etc*. Moreover, virtualization often has scalability issues as the need for multiple VMs increases.

Another commonly used strategy is to use debuggers to monitor software behavior. While a debugger in theory can capture all of an application's actions, debuggers are highly complex in nature, often suffer from commonly used anti-debugging techniques, and offer no intrinsic means to undo the modifications of an application. Additionally, most debuggers operate in user space, and are thus easier for an attacker to subvert.

Views offers an alternative to these strategies that satisfies the aforementioned requirements by providing isolation between user accounts. Each user automatically receives her own isolated yet logically complete view of the file system. Because this isolation is implemented at the file system level within the kernel, it is both efficient (only the blocks that change between views are copied and rewritten) and secure (*i.e.* from user space attacks). Even if an adversary can subvert the kernel, she still needs to successfully rewrite the Views file system to undo the isolation, yet still satisfy all of the standard requirements of a file system. Otherwise the system would crash and she would be unable to access the data stored on disk.

This being the case, Views can support software testing and sandboxing in an optimal fashion. Suppose a researcher needs to monitor a given application. The researcher can configure that application to always run with a given user account. When the application modifies a master file, an entity view of that file is created for that user account. The only change made to the master inode is that its extended attribute now contains a pointer to the entity inode belonging to that user ID. Otherwise, the master inode is left completely in tact as it was before the application was run. Most importantly, the data blocks belonging to the master inode are not changed by the application. All of the entity inode's blocks are copy-on-written from the master inode's blocks, which efficiently preserves the original values of the master inode's blocks. Moreover, the application's modifications do not affect and are not visible to any of the system's other user accounts. Since the entity view belonging to the application is logically complete, the application is able to read any system files to which it has the proper permissions, so the application is not denied any functionality. Because the application runs on the native OS and all isolation is performed at the file system level, there is virtually no performance loss while running the application in Views, as shown in Section 5.

After the researcher has finished testing the application, she can decide whether to retain or remove the application's modifications. Eliminating these modifications is both easy and guaranteed. Recall that all of the application's file system changes have been "quarantined" into their own entity inodes, which prevents them from disturbing the master inodes or any other entity inodes on the system. Consequently, these changes can be easily deleted by removing all entity inodes belonging to that entity view. This process is guaranteed to remove all file system modifications because an entity's changes are made to entity inodes only – the master inode's blocks are never modified in place. The only way that an adversary can circumvent this protection is to subvert the kernel and successfully rewrite the file system to undo the isolation inherent in Views.

Views also allows the researcher to retain the application's modifications if desired. She can continue to run the application indefinitely with the same user account, or alternatively she can merge the entity view created by the application with another entity view on the system, as described in Section 6.

The ability to test an application within a native OS without having to commit the application's changes is somewhat similar to the work done by Sun *et al.* [11]. However, because Views transparently provides each entity with its own fully isolated yet complete view of the file system, we avoid errors that might surface from transferring the properties of the isolated safe execution environment (SEE) to the host system. Views still provides the same degree of low-level isolation at minimal overhead, but Views can continue the file system isolation permanently if desired. This allows a user to permanently use two different versions of `gcc`, for example, without conflicting with any other entity view on the system.

4.3 Application Snapshot / Checkpoint and Recovery

The ability of Views to offer isolation of individual applications as described in Section 4.2 lends itself well to scenarios that require snapshots or checkpoint and recovery solutions. Section 6 discusses additional measures for enabling applications and processes as supported entities, partly in order to facilitate this type of functionality.

By setting up the application as an entity within Views, all of the application's file system modifications are saved to that entity's view. This entity view can optionally be saved and exported to a different view or system at a later

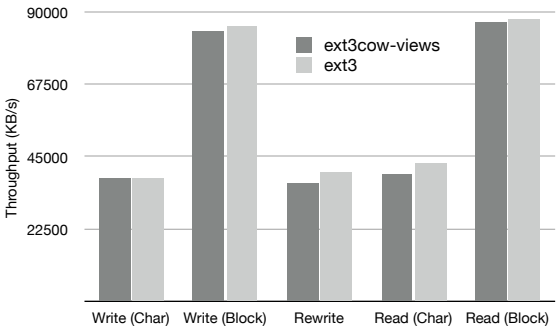

Figure 9: The throughput results from the Bonnie++ benchmark suite.

date. Allowing each entity's view to be saved, exported, and run on other environments enables the researcher to perform the following tasks:

- Examine the state and data of the application at any given time.
- Create snapshots of the application without the performance loss induced by virtualization solutions.
- Transplant a database, application, *etc*. from one system to another, potentially operating and/or modifying them in parallel. Because Views allows each entity view to be saved and exported, the transplanted view can be run at the present time or at a later date. This is especially useful when engaged in application testing and development.
- Restart the application on the original system in the event of a failure, or on a backup system if the original system cannot be repaired. This prevents downtime and loss in business continuity due to system malfunctions.
- Analyze the behavior of an application that runs on a production system without having to take the latter offline.

Fully transferring views from one system to another has not been thoroughly tested at this point. However, because Views stores all critical information in a clear format at the inode level, we believe this functionality is wholly supported by the Views infrastructure and should only require some additional development time.

5 Benchmarking Results

We measure the impact Views has on file system performance by comparing the I/O throughput of Views with that of ext3. We use Bonnie++, a benchmarking suite used to quantify five aspects of file system performance based on observed I/O bottlenecks in a UNIX-based file system. Bonnie++ performs I/O on large files (for our experiment, two 1GB files) to ensure I/O requests are not served out of the disk's cache. The five tests Bonnie++ performs are the following:

1. Each file is written sequentially by character.
2. Each file is written sequentially by block.
3. Each file is sequentially read and rewritten.
4. Each file is read sequentially by character.
5. Each file is read sequentially by block.

Figure 9 presents the throughput results for each Bonnie++ test, which show that our implementation of Views has little to no impact on the throughput performance of the file system when compared to that of ext3.

6 Future Work

Entities. Only users and groups are supported entities for Views at this time. We hope to develop additional classes of entities. Processes pose a unique challenge as they are difficult to accurately identify over time. Process numbers change with every execution and even the name of the binary may change. One possible solution is to identify a process or group of processes by a cryptographic hash of their contents or portion of their contents. Once processes are a supported entity, Views can be extended to provide isolation for any application without having to run the application under its own user account.

Policies. The Views framework was designed to be flexible for implementing and enforcing varying access control policies. Views' current policies and policy management system are rudimentary. The general policy that Views enforces is: the master view is read-only accessible by all entities; any additional data created by an entity is accessible only by that entity and the root user.

Future work for Views will include an API and configuration mechanisms to more granularly define the properties of each view, merge multiple views together, resolve any conflicts resulting from merging views based on pre-defined criteria, *etc*. For example, the administrator might wish to generate an entity view that is only valid for a certain period of time, after which point the view should be merged with another view or discarded altogether. Similarly, an administrator might wish to merge two views, and in the event of a duplicate file, keep the more recently modified file. Lastly, an administrator might wish to define which views, if any, can modify the master view such that all future views have access to those modifications.

We intend to provide administrators with this level of granular and comprehensive control by developing an intuitive API. Suppose that an administrator needs to create a new view for a user named `peter` that is a merge of the views belonging to users `matt` and `zachary`. This should apply to all files and directories within the Views file system. In the event that `matt` and `zachary` have a different view of the same file (*e.g.* each has his own `test.txt` file as shown in Figure 1), the new view for `peter` should assume the view of `zachary`. In the event that either `matt` or `zachary` creates multiple versions of a file using Views' versioning capabilities, the new view for `peter` should assume the earliest version that `matt` creates or the latest version that `zachary` creates. After 90 days, the `peter` view should be deleted. Our API call in Views will be able to represent this complex scenario using the following sample expression:

```
  root@localhost$ views
--new-user peter --merge matt,zachary
--recursive-path "/"
--view-conflict zachary
--version-conflict matt,oldest
--version-conflict zachary,latest
--expiration 90
```

The Views API then traverses all files and subdirectories within the directory specified in the "recursive-path" parameter of the API call (possibly leveraging or building upon existing tools such as `find`, `locate`, or `grep`). As the API encounters each inode in this path traversal, the API can use Linux's existing extended attribute API to query the inode's extended attributes for Views metadata. If such metadata is found, the Views API can read the metadata and apply the parameters specified by the administrator's API call accordingly. After determining what operations should be applied on each inode, the Views API can use the standard Linux system calls for reading, writing, creating, appending, deleting, *etc*., as Views does not modify any kernel interfaces.

For example, suppose an administrator makes a Views API call to delete all entity inodes in the `matt` entity view that have not been modified in the past 60 days. Using utilities similar to `find`, `locate`, or `grep`, each inode in the file system is examined. If the inode is a master inode, its extended attributes are searched for a name-value pair in which the name is `matt`'s user ID. If a match is found, the entity inode whose inode number is that of the value in the name-value pair is retrieved. The metadata from that entity inode is then read. If the data belonging to that entity inode has not been modified in the past 60 days, the entity inode and the corresponding extended attribute name-value pair in the master inode are both deleted. This process continues for all inodes in the file system.

Moreover, future work will include configurations within the Views API to allow data to be shared between entity views, as well as the access controls to govern this shared data. Suppose that an administrator wants a user `matt` to share the entity view of a user `zachary` in a folder named `/work` within the master view. The administrator can run the following Views API call:

```
  root@localhost$ views --share-view
-recursive-path "zachary@/work"
-user matt
```

The Views API then traverses all of the files and subdirectories within `/work`. For each master inode in this path traversal, the extended attributes are searched for name-value pairs in which the name is `zachary`'s user ID. If a match is found, the value from that name-value pair is extracted. This value is the inode number of `zachary`'s entity inode for that file. A name-value pair is then added to the master inode's extended attributes in which the name is `matt`'s user ID and the value is the inode number of `zachary`'s entity inode. From that point forward, anytime that either `matt` or `zachary` accesses that file, they will both be accessing the same entity inode, and will thus be accessing the same file and data blocks. This process is continued for all inodes in the `/work` folder.

Note that this solution applies only to pre-existing entity views. Future work will also include how to share data for entity views that have yet to be created.

7 Conclusions

We have introduced Views, an open-source file system extension that uses Linux's Extended Attribute API to provide isolated, complete, and configurable views of a file system namespace. Applications of Views includes access control compliance, sandboxing, and snapshotting. In this system, any entity that can perform I/O is able to modify the file system without affecting the file system views of the system's other entities. Each entity is therefore only able to see the changes it makes. Views is designed to support flexible isolation policies, allowing the permanence of file system modifications to be defined. Preliminary experimental results show that Views performs comparably to an unmodified ext3 file system. Our implementation of Views was built on and enjoys all of the functionality of ext3cow, an open-source versioning file system.

References

[1] CARD, R., TS'O, T. Y., AND TWEEDIE, S. Design and implementation of the second extended file system. In *Proceedings of the Amsterdam Linux Conference* (1994).

[2] GRÜNBACHER, A. POSIX access control lists on Linux. In *Proceedings of the USENIX Technical Conference, FREENIX Track* (June 2003), pp. 259–272.

[3] HALLYN, S. E., AND PAI, R. Applying mount namespaces. http://www.ibm.com/developerworks/linux/library/l-mount-namespaces.html, 2007.

[4] LOSCOCCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the USENIX Technical Conference, FREENIX Track* (June 2001), pp. 29–42.

[5] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. A versatile and user-oriented versioning file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2004), pp. 115–128.

[6] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). An Introductory Resource Guide for Implementing the Health Insurance Portability and Accountability Act (HIPAA) Security Rule. NIST Special Publication 800-66 Revision 1, 2008.

[7] PETERSON, Z., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage 1*, 2 (2005), 190–212.

[8] PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the Symposium of Networked Systems Design and Implementation* (2006), pp. 353–366.

[9] SANTRY, D. J., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)* (December 1999), pp. 110–123.

[10] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2003), pp. 43–58.

[11] SUN, W., LIANG, Z., SEKAR, R., AND VENKATAKRISHNAN, V. N. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of the Network and Distributed System Security Symposium* (2005), pp. 265–278.

[12] UNITED STATES CONGRESS. The Health Insurance Portability and Accountability Act (HIPAA) Security Rule. 45 CFR Parts 160, 162 and 164, 1996.

[13] VMWARE INC. VMware ESX Server. http://vmware.com/, 2009.