Chapter Four

/cloud: A Cloud Synthesized File System

In persistent storage there are three degrees of freshness. `cloud-inotify`, discussed in the previous chapter, is for low-latency monitoring the first degree of freshness—in-flight operations mutating persistent state. `/cloud` provides an interface to the second degree of freshness—live state stored in virtual disks across the cloud. This state is not as fresh as in-flight write operations, nor as old as state kept in historic archives. `/cloud` offers a read-only view into the file systems within virtual disks associated with running instances. The natural fit for implementing `/cloud` was a file system interface which provides familiar file-level access to files inside monitored file systems across the cloud. Thus, `/cloud` is implemented by a compact FUSE driver as a POSIX-compliant read-only file system with eventually consistent semantics for file data from remote virtual disks. Because `/cloud`'s implementation uses the familiar file system abstraction, it is directly usable by legacy applications without re-implementation or modification.

`/cloud` benefits from DS-VMI's normalization of file system data structures by only needing a single implementation to handle multiple different types of VM file systems. We envision many applications for `/cloud` including querying old log data, scanning for new vulnerabilities, or checking for misconfigurations. Each of these tasks is impossible in a purely event-driven architecture such as `cloud-inotify`. Accessing old log data cannot happen with `cloud-inotify` because there is no method of going back in time within a file-level update stream. Although, we will describe a system of storing these update streams for the purpose of going back in time in Chapter 5. Scanning through files for vulnerabilities does not make sense with events because many files might never receive writes in which case they are invisible in `cloud-inotify`. Scanning for a misconfiguration has the same exact problem: even if we could travel backwards in a file-level update stream, files which never mutate are not under the purview of DS-VMI.

The rest of this chapter is organized as follows. Section 4.1 provides an overview of the design ideas and implementation technologies of `/cloud`. The core implementation (200 lines of C) is kept small by leveraging abstractions provided by the DS-VMI mechanism. We describe the `/cloud` FUSE driver and normalized format in Section 4.2. Section 4.3 describes `/cloud`'s consistency model with metadata versioning. Metadata versioning guarantees metadata consistency, but file data is

eventually consistent. Section 4.4 discusses two applications of `/cloud` in more detail. The final Section 4.5 demonstrates the usefulness of a legacy toolchain when layered on top of the file-system-like `/cloud`.

4.1 Design and Implementation of `/cloud`

Batch-style workloads typically operate by scanning through a large corpus of data, computing some intermediate state, and producing a final answer. Thus, we expect them to be long-running processes which read large portions of virtual file systems at a time. A large amount of legacy tooling and frameworks already exist for processing batch-style workloads such as the MapReduce [31] distributed runtime. These properties guided our design of `/cloud`.

`/cloud` needed to implement an interface which did not require rewriting entire toolchains and frameworks. The pre-existing codebases are battle-hardened from decades of testing, familiar to engineers, and represent millions of man-hours time worth of development. This led us down the path of implementing a lowest common denominator interface—the POSIX file system interface—for accessing live virtual disk state. Because these jobs are long running, and potentially touching every instance in a cloud, we deemed it not appropriate to snapshot every virtual disk for every batch job. Thus, we took the design decision to read data directly from live virtual disks. Ideally, snapshots would enable freezing large swaths of virtual disk state at once for batch workload consumption. But, taking frequent snapshots of entire virtual disks may incur too much overhead.

`/cloud` is implemented as a FUSE file system within a Linux host which translates between host system calls into the virtual disk space of a cloud instance by leveraging DS-VMI maintained datastructures. Exposing `/cloud` to other operating systems is as simple as layering an SMB or NFS server on top because `/cloud` appears as a simple mount within its host. This means legacy tools can reside in their operating environment of choice, and still access files within cloud instances for monitoring. FUSE does not fully support the local Linux `inotify` interface. If it did, a single FUSE file system could serve as a shared implementation for both `cloud-inotify` and `/cloud`.

`/cloud` is exportable over the network via Samba or NFS. Administrators can use this interface to rapidly query log files and configuration state across multiple instances in a cloud with legacy tooling. For example, consider organizing VM file systems within a hierarchical directory scheme: `/cloud/host/vm/fs/path`. Administrators can leverage familiar legacy tools such as `grep` to quickly search subsets of VMs. They could also use standard log monitoring applications such as Splunk [112] to monitor log files within `/var/log/` across VMs without executing agents inside those VMs. `/cloud` only transmits data when requested via file system operations.

4.2 Implementation with POSIX Read-only Semantics

Less than 200 lines of C code define `/cloud`'s implementation. This is due to DS-VMI's normalization of file system metadata into an intermediate format, and the power of the FUSE framework for

```
struct gray_inode
{
    uint64_t size;
    uint64_t mode;
    uint64_t uid;
    uint64_t gid;
    uint64_t atime;
    uint64_t mtime;
    uint64_t ctime;
};
```

Figure 4.1: Normalized metadata kept via DS-VMI for `/cloud` and other purposes. There is a list of blocks associated with the file not shown. And, in the case of directories, also a list of files within the directory.

implementing userspace file systems. Figure 4.1 shows the normalized metadata `/cloud` maintains via DS-VMI for populating necessary fields when looking up file attributes within the FUSE callbacks. We chose 64-bit unsigned integers to represent most data because (1) it is unsigned data, and (2) we felt this field was large enough to cover all of our existing use cases. However, it must be noted that file systems such as ZFS [78] which are based on 128-bit metadata attributes would require an expansion of the field sizes of our normalized format. This normalized format is a compromise down to a lowest common denominator of representing files. But, exactly like intermediate formats produced by compilers, it lets us create unified tooling around file data independent of the source file system.

Notably absent from this structure is a representation of the on-disk byte stream, or an extensible list of file attributes. We deliberately scope the FUSE interface to being simple and do not expose file-system specific features or attributes. The stream of bytes representing a file are stored in a separate list. Although this list is not needed for introspection, as only a reverse mapping lookup table is needed mapping blocks to files and not the converse, DS-VMI maintains it for `/cloud` on every metadata update in near-real-time.

The file system functions implemented by `/cloud` are listed in Table 4.1. `/cloud` implements a very minimal set of 5 virtual file system callbacks. The FUSE frameworks hides a lot of complexity, and a purely read-only file system completely avoids implementing trickier write functions. `/cloud` guards itself guaranteeing that any open file descriptor is read only by gating the `open` method to return `EROFS` if the flag `O_RDONLY` is not set. This ensures that callees to `/cloud` functions operate exclusively on read-only files. Of course, the virtual disk itself has some amount of churn independent of our external gating—thus, the files are generally writable by the guest.

Virtual File System Call	Implementation
`open`	ensure the pathname exists, and `O_RDONLY` is set
`read`	read from an arbitrary position in the file into a buffer
`getattr`	get a standard set of attributes about a file (eg size)
`readdir`	read the entries of a directory into a FUSE-specific list
`readlink`	place the pathname pointed to by a link into a buffer

Table 4.1: The customized virtual file system calls required to implement `/cloud`. `/cloud` returns `EROFS` on any attempt to open a file for writing.

4.2.1 Limitations of Normalization

The normalized format simplified the engineering necessary to create usable interfaces on top of DS-VMI. However, such normalization requires either a "lowest common denominator" format, or synthesis of values when a file system does not maintain metadata expected by our normalized format. It is possible that file systems exist where both issues occur. For example, our normalized format resembles a simplified UNIX-style inode. This is because our front end, FUSE, lives within a POSIX environment which expects inode-like functionality from underlying file systems. `NTFS` does not maintain all of the same metadata fields as `ext4`, thus some metadata is synthesized for `NTFS`. Mode bits are a great example of the need for synthesis with a normalized format. Mode bits do not exist within `NTFS` in the same format as in `ext4`. In addition, `NTFS` has a very verbose on-disk format and large amounts of metadata are currently ignored.

Although very useful, normalization may not work in all cases. The extended attributes of a file may end up mattering to a specialized application. However, none of the applications we studied so far for file-level monitoring workloads use extended attributes or file-system-specific attributes. This is because such infrastructure tools try to generalize across as many file systems and environment types as is feasible. There is a tension between using the features of a file system, and remaining generic. Currently, based on the limited set of applications studied in this dissertation, we believe most tools strive to remain generic. This generality lets them maximize their utility across as many environments as possible.

4.3 Metadata Versioning

To ensure correctness and a consistent view of guest file systems for legacy tools, we introduce the notion of *metadata versions*. A metadata version is a consistent snapshot of a file's metadata. A file has at most two metadata versions: its last known consistent state and its current, in-flux state. Legacy applications reading a file or its attributes are presented with its last known consistent metadata state. Reads of file data go to the original virtual disk, as shown in Figure 4.2. File versions only guarantee a consistent metadata view, but the data can change while it is being read from the

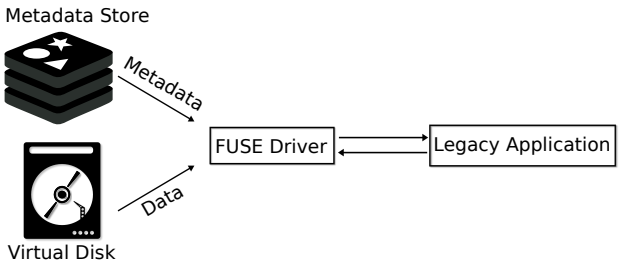

Figure 4.2: `/cloud` implementation

virtual disk. In other words, readers using our FUSE driver must occasionally handle stale data. For append-only style workloads, such as logging, this is rarely an issue because earlier data blocks are never overwritten unless log rotation occurs.

Our metadata versioning is per-file rather than per-directory. Per-directory metadata versioning implies transitive closure over the metadata versions of the files within the directory as well. Such a need for recursive versioning quickly turns into whole-file-system versioning when workloads involve scanning the root directory which, although conceivable, is not efficient with the format of the metadata lookup tables for introspection purposes. Thus, we bounded our metadata versions to be per-file rather than per-directory or per-file-system. This is also similar to the design decision within the `git` [43] version control software to track versions of files rather than versions of directories.

4.4 Applications

In the Google example from Section 3.2.1, we assumed operators could be notified nearly instantaneously about misconfigured permission bits. Of course, this can only occur if they are already being monitored. If one discovered such a misconfiguration after the fact, adding it to a rule base that checks future streamed updates would not detect instances that already contain the misconfiguration. Using familiar commands such as `find`, one can check permissions across the cloud: `find /cloud/*/*/lib -maxdepth 0 -not -perm 755`.

In the log monitoring example from Section 3.2.2, we assumed the insights we want from log files are already known. But, if we come up with a new metric and want to know its historical value we can leverage `/cloud`. For example, perhaps unsuccessful `ssh` logins were never monitored before. It would be useful to be aware of how many of these occurred in the past, as they may represent malicious attempts. Using `/cloud` and `grep` we can quickly scan recent logs across all instances: `grep "Failed password" /cloud/*/*/var/log/auth.log`.

```
~ ./demo.sh
+ sudo kpartx -av /home/wolf/VM/vm_ext4_test/vm_ext4_test.raw
add map loop0p1 (252:2): 0 41938944 linear /dev/loop0 2048
+ sudo mount /dev/mapper/loop0p1 /mnt/linux-ext4/
+ sudo gammaray/bin/gray-fs /mnt/gray-fs/ /home/wolf/VM/vm_ext4_test/vm_ext4_test.raw
+ read

+ sudo ls /mnt/linux-ext4
bin   dev  home        initrd.img.old  lib64       media  opt   root  sbin     srv  tmp  var      vmlinuz.old
boot  etc  initrd.img  lib             lost+found  mnt    proc  run   selinux  sys  usr  vmlinuz
+ sudo ls /mnt/gray-fs
bin   dev  home        initrd.img.old  lib64       media  opt   root  sbin     srv  tmp  var      vmlinuz.old
boot  etc  initrd.img  lib             lost+found  mnt    proc  run   selinux  sys  usr  vmlinuz
+ sudo find /mnt -type d -name wolf
/mnt/linux-ext4/home/wolf
/mnt/linux-ext4/var/lib/sudo/wolf
/mnt/gray-fs/var/lib/sudo/wolf
/mnt/gray-fs/home/wolf
+ read

+ sudo diff -r /mnt/linux-ext4 /mnt/gray-fs
```

Figure 4.3: Command-line interaction with `/cloud`. In this demonstration, we mount an `ext4` file system using the normal kernel module via the `mount` command, and `/cloud` via the `gray-fs` command. We then use normal, legacy tools such as `ls`, `find`, and `diff` to show that `/cloud` has the same coverage as the Linux kernel's `ext4` module for this file system. The file system being introspected was an `ext4` file system residing within a 20 GiB virtual disk of an Ubuntu 12.04 Server 64-bit guest.

4.5 Exploring a `/cloud` Mount

In this section we briefly explore a `/cloud` mounted file system and demonstrate legacy tool support. In addition, we show complete parity for the introspected file system with the Linux kernel's `ext4` implementation. Figure 4.3 shows mounting the file system into two paths on our host machine. The first path, `/mnt/linux-ext4` is mounted using the canonical Linux `ext4` module. The second mounted path, `/mnt/gray-fs` is mounted using our introspection implementation of `ext4`.

We first explored the mount points using the very simple directory listing command, `ls`. We see that at least for the root folder, both `ext4` implementations report the same list of files and folders. Next, we ran `find` looking for any folder with the name `wolf`. We found exactly two folders in both mounts with the name `wolf`. Finally, we ran the deep comparison tool `diff` recursively on both mounts. `diff` did not find any differences between the two except for device files. `/cloud` currently does not recreate device files in exactly the same way as the Linux kernel's implementation of `ext4`. However, for the rest of the 100,000+ normal files, not a single one differed.

This example exploration of a `/cloud` mount shows the power of a simple file-system-like abstraction. Legacy tools work without any change, and this makes testing introspection code much easier because it is possible to compare with pre-existing tooling. If there is more than

one virtual machine, a tree of mount points is formed using the `gray-fs` command. These file systems are mountable over the network as long as the originating virtual disk, or a snapshot of the disk, is made available over the network. The in-memory metadata kept fresh by DS-VMI is automatically network available. In our prototype implementation, network availability is provided by the Redis [101] key-value store.