Chapter Seven

Related Work

This dissertation addresses how we monitor persistent storage in the new reality which decouples storage from the operating systems and applications running on top of it. Whether this occurs due to a virtualizing hypervisor, or just because the storage is now network attached does not matter. The new proposed mechanism for monitoring virtual disk state, DS-VMI, is not limited by the type of operating system being monitored, the type of underlying storage technology, or where the writes pass through. It achieves all of this with a single simple set of requirements: practical support for duplicating storage writes to an introspection endpoint, and theoretical guarantees over the semantics of the file systems that it monitors. The described interfaces leveraging this proposed mechanism are designed to serve classes of monitoring workloads, while remaining application-agnostic. Introspection at this scale has never been described before, although it does parallel the efforts in the networking community for packet filtering. In addition, many bodies of research relate directly to this dissertation we described below.

This chapter references and describes the immense body of work that DS-VMI both incrementally improves upon and evolves. The core direct contribution of this dissertation is DS-VMI—an agentless, OS-agnostic technology exposing guest file system state via intermediate cooperation. `/cloud` and `cloud-inotify` are the first attempts, to our knowledge, at constructing unified guest-independent interfaces to live file system state. `/cloud-history` uniquely explores deduplication in a storage setting *not* to save space, but to save indexer computation time. In the context of `/cloud-history` we show that although bandwidth does matter to indexers, nothing trumps reducing their total workload.

The rest of this chapter explores many facets of the monitoring problem as it has been approached by related, but different, research communities. Section 7.1 describes the performant solutions [2, 45, 61, 72] that the storage community provides for snapshotting which could be polled for file-level updates. Section 7.2 discusses versioning file systems. Although practically a perfect fit for many of the properties this dissertation achieves, versioning file systems require guest support which is in direct opposition to a core goal—agentless monitoring. Section 7.3 discusses approaches in developing smart disks [4, 108] that implement semantic understanding of

file systems and file-level updates. The primary goal of the smart disk community was a traditional storage goal: increase performance via intelligent prefetching and reorganizing sector layout on disk. Section 7.4 discusses related techniques [14, 34, 47, 54, 55, 88, 131, 132] for externally understanding writes to virtual storage. Sections 7.5 and 7.6 describe the traditional agent-based file-level monitoring solutions, and nascent efforts towards agentless monitoring. Finally, we conclude our research in Section 7.7 by surveying a large amount of backup systems and comparing them within a framework we specifically developed for this purpose. The goal is to understand the gaps in the backup solution space, especially for searchable backup.

7.1 Snapshotting Derived History

The most efficient implementations of virtual disk snapshotting use block-level techniques to create snapshots representing only the changed blocks between different versions of a virtual disk. They can guarantee exact point-in-time versioning of virtual disks. From the beginning of execution, disk writes go into an overlay over a base disk image. Snapshotting amounts to creating a new overlay, redirecting writes to the new overlay, and copying the old overlay to a centralized store. The old overlay may then safely compact into the original disk image, while the new overlay absorbs writes waiting for the next snapshot event. Compaction ensures that the overlays do not infinitely accumulate, thereby guaranteeing that reads do not become prohibitively expensive by traversing too many layers.

To implement the functionality of the interfaces described in this dissertation, block-level snapshotting by itself requires re-indexing disk state at a file-level. But, block-level snapshotting has been made very efficient. Olive [2], Lithium [45], and Petal [61] create snapshots within hundreds of milliseconds and incur low I/O overhead during snapshot creation. Parallax [72] was explicitly designed to support "frequent, low-overhead snapshot[s] of virtual disks." For example, snapshotting at a high frequency of 100 times per second caused only 4% I/O overhead to the guest OS using the virtual disk served by Parallax. Thus, low overhead, high frequency, block-level snapshotting is possible, but it is not enough for efficient file-level interfaces, especially for event-driven workloads. The Elephant file system [103] efficiently stores versions of files, but does not perform file-level deduplication. Veeam's Backup & Replication [120] indexes files after block-level deduplicated snapshotting. We could speedup indexing by skipping portions of the file system tree with old timestamps, but then we lose security guarantees—nothing prevents tampering with timestamps.

Virtual machine time travel [115] is the most closely related work to `/cloud-history`. They implement copying of the block-level write stream to a continuous data protection (CDP) storage server. By keeping a log of virtual disk writes, they are able to recreate virtual disk state at any point in the past. They use CDP in combination with memory checkpointing to let an administrator rollback any VM to a prior execution state. They do not interpret the block-level stream at a file-level, unlike DS-VMI. For example, file-level deduplication is not efficient because it would require re-indexing each snapshot created via CDP—potentially re-indexing upon every block-level modification. The storage community has also proposed using incremental hashing, but in the

context of achieving secure properties with cheap, highly distributed state for network-attached secure disks [44].

Generality: Virtual disk snapshots require no in-VM support. If the virtual disk is implemented as a file within a host file system, snapshotting that file does not require hypervisor support either—an underlying file system or storage technology could implement snapshotting. Their block-level granularity means they work for any type of guest OS, file system, and mix of applications. This wide applicability makes virtual disk snapshotting an attractive mechanism as it generalizes to all possible VMs. When implemented as files, they are also easily copied between hosts in a cloud.

Isolation: The mechanism implementing virtual disk snapshotting, whether inside the hypervisor, or in the host environment, is completely decoupled from the environment within the VM. This means that any compromise, misconfiguration, or failure experienced by the VM can not affect virtual disk snapshotting. In other words, virtual disk snapshotting maintains the integrity of persistent storage.

Cost: The cost of virtual disk snapshotting is fundamentally tied to its strengths. Its generality and isolation leaves snapshots as a series of point-in-time samples. This is not indexer-friendly: indexing a snapshot requires applying it to a base image, mounting file systems, and crawling those file systems for changes. In addition, the discreteness of these sampled views limits history to those events that persist between them. An alternative is storing not just the overlay containing changed blocks, but all of the blocks of the VM in every snapshot. Although more expensive, this snapshot design is more indexer-friendly. However, it only skips the first step of applying overlays—one must still mount file systems and crawl them for file-level changes. Thus, although a potential for implementing `/cloud` and `/cloud-history`, the time to index with snapshotting would greatly increase, and is therefore not a good choice as a storage format for implementing latency-sensitive interfaces such as `cloud-inotify`.

7.2 Versioning File Systems

Versioning file systems, such as HAMMER [33], NILFS [82], or Elephant [103], directly expose file-level history which makes them very index-friendly. Unlike virtual disk snapshots, versioning file systems require no costly application of overlays, no mounting of file systems, and, depending on their implementation, no crawling to discover changes at a file-level. As an example, imagine Alice uses a log-structured file system that exposes historic state. Positions in the log represent different versions of the file system. Indexing the file-level changes since the last version requires crawling the log entries since the last version. The log forms a compact representation of the file-level changes, similar to how a snapshot overlay forms a compact representation of block-level changes.

Fine Granularity: A file system, by definition, directly knows the files it stores and the operations performed on those files—potentially all operations. A file system directly exposes file-level state. A versioning file system has the opportunity to retain per-file versions, and define the retention policies over those versions. Users are not stuck to a single set of policies for entire VMs. For

example, Alice may not want to keep certain private files' history retained, or versioning of large binary artifacts derived, and easily regenerated, from source files.

Index-friendly: A versioning file system directly exposes the history of files. With no intermediate costly steps between the capture of history, and its file-level interpretation, a versioning file system is index-friendly. Indexing a log of file-level changes is much cheaper than scanning entire file systems looking for changes.

Cost: The cost of a versioning file system is abandonment of generality and isolation. We lose generality by requiring support inside the VM of the versioning mechanism—the versioning file system. In this case, the cloud requires all VMs to install, configure, and maintain a specific versioning file system. This is not possible for all types of VMs. Specifically, VMs with closed source kernels may never support the required versioning file system. We lose isolation by requiring in-VM configuration and maintenance. Non-malicious, accidental misconfiguration of the versioning file system leads to lost history for any individual VM. Malicious attacks compromising the VM violates the integrity of the captured history—attackers have free reign to modify the reported history of a VM.

7.3 Smart Disks

Semantically-smart disk systems (SDS) [108] interpret metadata and the type of a sector on disk as well as associations between sectors, but do not support distributed introspection across many disks. There are three classes of SDS systems. The first embeds knowledge of file systems into disk firmware. This is very similar to DS-VMI introspecting writes. Although the writes are received by a storage device, we still have a chance to introspect them before they are physically recorded. The obstacle facing this approach is hardware limitations such as very low memory. This limits the amount of buffering and introspection possible. The second assumes the writer exposes data structures to the SDS system. This requires guest support which sacrifices a core property provided by this dissertation. The third infers the data structures and on-disk layout of file systems based on external observations. Often, this third form is helped by a userspace probe process sending known file-level operations to the SDS system. Even with the probe, there are assumptions that must be made about the types of file systems being introspected, and the probe itself is a form of agent within the guest. We consider this third class a very promising research topic which needs more active exploration. The greatest barrier to introspection in general is the upfront cost of understanding complex data structures designed without support for external observation. If we could externally learn these complex data structures automatically on-the-fly, then there are no remaining obstacles for widespread implementation of introspection.

IDStor [4] introspects disk sector writes in iSCSI network packets with demonstrated support of the `ext3` file system. Their approach could replace the hypervisor hooks DS-VMI currently uses, especially if cloud instances boot from and use network-based volumes. Essentially, IDStor [4] is a research effort extending packet filtering technology to storage. Zhang et al. [132] describe an intrusion detection system placed inside a hypervisor using smart disks to understand guest VM

file systems. However, they focus on more than monitoring and placed significant logic on the critical I/O path—an entire intrusion detection system—which DS-VMI tries to avoid. All of these approaches could feed the interfaces envisioned within this dissertation: `/cloud`, `cloud-inotify`, and `/cloud-history`. Their key architectural difference from DS-VMI and its interfaces is that they do not consider operating on collections of disks, and instead focus on single disk systems.

7.4 Virtual Machine Introspection

Garfinkel and Rosenblum [42] coined the term *virtual machine introspection* and developed an architecture focusing on analyzing memory, although they introduced the possibility of disk-based introspection. Their landmark research recognized the power of external observation for varied tasks such as debugging, security, optimization, and monitoring. DS-VMI is a direct extension and evolution of their work into the modern world of software-defined storage. Pfoh et al. [89] developed a formal framework for analyzing introspection systems. DS-VMI, in their framework, would be classified as an *out-of-band* introspection method [89]. XenAccess [88] introspects both memory and disk, but only infers file creations and deletions. Zhang et al. [131] introspect disk, but for enforcing access control rules for a single virtual disk system in the critical I/O path. VMWatcher [55] interprets memory and disk operations, but requires kernel source. VMScope [54] captures events such as system calls, but does not interpret virtual disk writes. Virtuoso [34] automatically generates introspection tools, but not for disk operations. Hildebrand et al. [47] describe a method of performing disk introspection to the point of identifying disk sectors as metadata or data. Maitland [14] is a system that performs lightweight memory-based VMI for cloud computing via paravirtualization. All of these approaches provided insight for implementing DS-VMI, but none of the proposed VMI systems focuses on creating a robust, complete, and performant virtual disk-based introspection system. In addition, VMI literature, when it covers disks, typically deals with single disk systems ignoring the more prevalent cloud case involving collections of virtual disks.

This dissertation lifts introspection out of single-system research and thrusts it onto the modern stage of cloud computing. Thus, it was forced to take a look at what is needed to move introspection into a distributed, production focused environment. The VMI literature, while touching on all the necessary parts, has focused more on single-system introspection frameworks. The closest related work is Maitland [14], which does look at introspection in the cloud, but assumed paravirtualization and not full generality. The confluence of generality, agentless, and distributed requirements of this dissertation truly distinguish it from the VMI work of the past.

7.5 Agent-based File-level Monitoring

Agents [7, 21, 36, 40, 51, 66, 112, 113, 118, 119] are the current state-of-the-art method of monitoring file-level state within VMs. Agents run the gamut from a simple single-system virus scanner, to distributed monitoring across fleets of hosts numbering in the tens of thousands within world-

wide datacenters. ClamAV [21] is an example single-system virus scanner. It requires installation of the ClamAV scanning engine, and virus definition files. As we mentioned in the introduction, agents can introduce vulnerabilities into their host environments. Although designed for security applications, even tools such as virus scanners have the potential of introducing fatal holes into the armor of their hosts. ClamAV has many such vulnerabilities attributed to it, and a recent example is CVE-2015-2668 [84]. Symantec enterprise antivirus products also have exploitable vulnerabilities [83] letting attackers control what should be protected resources. Thus, placing such agents within the boundary of trusted systems creates inherent unavoidable risk.

An illustrative example of a modern distributed monitoring system employing agents is Akamai's Query [24]. Query aggregates information from agent processes within every Akamai node across 60,000+ servers in 70 countries within 1,000 autonomous systems. Key to Query's success has been its SQL-like interface providing operators an efficient method of answering questions involving thousands of nodes. Query attempts to guarantee that the staleness of data never goes beyond 10 minutes. Not all of the metrics Query obtains could be obtained from file-level information, such as the process table which is normally kept only in-memory. Thus, some of Query's functionality must be retained by an in-guest agent; however, any file-level capabilities could be factored out by DS-VMI and its interfaces. Query bounds the end-to-end latency for a usable monitoring system to be on the order of 10 minutes, which is an easy bound to achieve with DS-VMI as reflected in Chapter 3. The interfaces layered on top of DS-VMI such as `/cloud`, are designed to be just as easy for operators to use as Query's SQL-like interface. A key lesson from Query is that interface design is critical for both scalability and usability. Sysman [10] provides a unified file-system interface to configuration state within Linux systems such as that represented within `/proc`. Sysman also has support for modifying configuration state inside systems via writing to special files. Sysman's interface design is similar to `/cloud`, although different in functionality—it permits writes which mutate guest state. Sysman provides evidence that such unified interfaces help scale management and reduce administrative overhead when handling thousands of systems.

7.6 Agentless File-level Monitoring

Agentless monitoring is a much less explored area of research, especially at the file-level [60]. Several works have developed agentless monitoring of memory [87, 121], but file-level agentless monitoring is much rarer. Most systems that claim to execute agentlessly in practice use interfaces already built-in to modern OSs such as Windows Management Instrumentation (WMI) [75] or the Simple Network Management Protocol (SNMP) [19]. An example of such a system is Ansible [6] for distributed system management. Ansible comes from the DevOps community and focuses on versioned management of system state and configuration across both Linux and Windows hosts. It describes itself as an "agentless" IT automation tool. For Ansible, agentless means managed hosts need not install a special agent. They instead must come pre-installed with a version of Python 2.4 or higher, and less than Python 3.0. A managed Linux host must have an ssh server installed, configured, and network-accessible. A managed Windows host must have the Windows Manage-

ment Framework along with PowerShell configured for remote access. Although greatly lowering the amount of configuration, there is still configuration of the monitored hosts. Generally, modern self-described "agentless" systems actually require some amount of configuration and support from the hosts they monitor or manage. Thus, most modern "agentless" systems fall back on preexisting mechanisms and are not truly agentless—they rely on guest support and resources. In this dissertation, we develop a truly agentless file-level monitoring framework based on robust virtual disk introspection.

7.7 Analyzing Backup Systems

We studied backup systems in search of one with the same mixture of qualities as `/cloud-history`. What we found is a gap in the backup solution space, which we fill with `/cloud-history`: an architecture designed for modern, quick-to-access backup storage that achieves low time to index. None of the backup systems surveyed was designed around rapid general-purpose indexing with rapid random access to backup objects. This is probably due to the historic nature of slow-to-access archival storage. Now, with the introduction of fast-access archival storage we have the opportunity to implement rapid indexing and, perhaps more importantly, re-indexing. Re-indexing enables changing and tweaking index algorithms for future demands. For example, the discovery of Heartbleed [23] benefits from an index of installed software, but also needs an index over user-compiled binaries and static binaries. Static binaries include libraries which are not necessarily installed system-wide, thus they will not appear in indexed lists of installed packages. Such indexes are prohibitively expensive to construct on-the-fly with the backup systems studied in this section, but would be much cheaper with `/cloud-history`.

While surveying backup systems, we developed a framework for classifying, comparing, and contrasting them. This framework helped us methodically catalog backup systems, and pose questions with more rigor than just a simple survey alone. For example, we can use our framework to ask the question, what gaps exist in the current design space of backup systems?

We developed nine axes for evaluating backup systems. We believe these axes capture the most relevant features of backup systems. As we describe each axis we provide examples from the research literature and industry. The axes which we developed are:

1. Granularity of state (Section 7.7.1)
2. Granularity of time (Section 7.7.2)
3. Complexity of supported queries (Section 7.7.3)
4. Level of consistency (Section 7.7.4)
5. Level of scale (Section 7.7.5)
6. Types of backup format (Section 7.7.6)

7. Types of storage targets (Section 7.7.7)
8. Types of input systems (Section 7.7.8)
9. Protection Radius (Section 7.7.9)

At the end, in Section 7.7.10, we compare backup systems using these nine axes.

7.7.1 Granularity of State

Granularity of state refers to the level of abstraction at which capturing or indexing data for backup occurs. We have from lowest level to highest level:

1. **Record-level:** The granularity is sub-file level and means the backup system deeply understands the applications it monitors and their on-disk file formats. An example is Veeam [120] for Microsoft Sharepoint.
2. **File-fragments:** The granularity is at the level of file-fragments, or chunks, which are either fixed- or variable- length. Two examples are bup (variable-length) [16], and Venti (fixed-length) [93].
3. **File-level:** The granularity is file-level meaning the backup system understands the file systems it monitors, or has agents which collect files inside monitored systems. An example is the Elephant [103] file system.
4. **File-system level:** The granularity is file-system level meaning the file-system itself has the ability to create snapshots, or the backup system has a mechanism to snapshot individual file systems. An example is btrfs [99] or zfs [133] snapshotting.
5. **Block-level:** The granularity is block-level meaning an entire block device is snapshotted at once as just a bunch of bytes. Although the easiest to implement for a backup system, it provides the least utility to end users. An example is QEMU virtual disk snapshotting [91], or LVM snapshotting [3].

Although the most useful to end users, record-level granularity is the most complex to implement. Block-level granularity, one of the most common forms of backup, is the simplest to implement.

7.7.2 Granularity of Time

Granularity of time refers to the nature of the regularity and completeness of versioning in capturing the changes occurring to monitored systems. There are two forms of backup which affect the completeness of versions:

1. **Discrete:** Snapshots or versions are kept at distinct points in time normally following a schedule such as hourly, daily, or weekly. An example system that snapshots at scheduled intervals is Déjà Dup [32].

2. **Continuous:** Snapshots or versions are kept at every single recorded modification. An example file system that can continuously version is NILFS [82].

Clearly, continuous results in a finer granularity for queries. However, continuously recording modifications could introduce undesirable overhead and might be overwhelming to a user when intermediate modifications are meaningless.

7.7.3 Complexity of Supported Queries

The complexity of query refers to the expressiveness of the backup system's query interface. We identify four styles of expressiveness:

1. **Point-in-time:** All backup systems support retrieving a version of an object based on a notion of time. An example is Microsoft Windows System Restore [76] which lets one revert a system to a prior state in time.

2. **Metadata-based:** Some backup systems further index metadata from the objects or files that they store. An example backup system that allows metadata queries is Carbonite [18].

3. **File-based:** This style of query can leverage indexes created over file data. This requires the capability of indexing at a record-level inside files. An example of a backup system that can do this is Apple's Time Machine [7].

4. **General Queries:** This style of query is the most rich and expressive. It lets users search not only over indexes, but to perform arbitrary computation over stored objects as part of their search. An example of such a system, although not designed for backup, is ZeroCloud [129].

The most basic search query, and simplest, is point-in-time. It is well suited for most basic backup queries such as, "what was the latest version of this document?" This type of query is also ideal for recovery from accidental deletion, or when a known good configuration exists. No known backup system supports efficient deep queries over unindexed data. If one did, it could answer questions such as, "which binary executables contained a newly discovered zero-day vulnerability?"

7.7.4 Level of Consistency

There are different forms of consistency that a backup system can provide. In addition, with many complex services depending on each other, the capability of creating cross-system consistent snapshots at discrete points-in-time may emerge as important in the near future.

1. **No Consistency:** The backup system does nothing special to ensure consistency of data. For example, copying a block device without snapshotting. With writes ongoing during the copy operation, the back up will not have consistency with any point in time.

2. **Point-in-Time Consistency:** Care is taken to ensure that backups are consistent with a specific point in time. Copy-on-write, point-in-time snapshotting is an example of this category. MagFS [68] is an example system supporting this type of consistency.

3. **Multi-system Point-in-time Consistency:** This is a special form of consistency which applies across multiple systems. Coda [104] is an example system that provides this level of consistency for replicated read-write volumes.

These range from simple to complex. The most commonly implemented form of consistency is point-in-time consistency. Often this is done via a snapshot mechanism implemented on top of copy-on-write technology. Backup systems that provide no consistency create impossible to reason about versions of data. Multi-system point-in-time consistent backups are very rarely implemented.

Unfortunately, many backup systems do not provide any form of consistency. A backup for these tools consists of scanning the file system or objects of interest. This type of scanning means objects can mutate during backup execution. Examples include UNIX-based open source backup tools such as Amanda [135], BackupPC [29], bup [16], dump [114], rsync [73], and Déjà Dup [32]. Proprietary tools such as Dropbox [64], tarsnap [26], Backblaze [9], and Crashplan [22] also walk or monitor a file system when creating backups.

7.7.5 Level of Scale

Backup systems are designed with different levels of scale in mind. If one exceeds the scale of a backup system, performance often degrades. For example, backups may become slower if the number of backed up objects exceeds the design of the system. Or, retrieval of backed up data may become practically impossible once the scale reaches a certain size because it could become incredibly slow.

1. **Ingest Bandwidth:** Backup systems have to scale to an ingest bandwidth that allows a full backup to complete within a backup window. Often, in datacenters, full backups are performed over weekends when the overall system is less loaded. Thus, a general backup window will be less than 48 hours. An example backup system with very high ingest bandwidth is Sepaton [107], which can ingest up to 80 TB/hour.

2. **Number of Objects:** Depending on the implementation of a backup system, it may only scale to a certain amount of stored objects. For example, BackupPC can not handle millions of copies of a single file. This is because it deduplicates by hard linking to a file. Typically, the maximum number of hard links is a constant from an underlying file system. This limits the scalability of BackupPC.

3. **Space Requirements:** If a backup system does not leverage compression, delta encoding, deduplication, or other space-saving techniques, it may not scale well when the sizes of backed up systems is large. An example backup solution that employs deduplication, delta encoding, and compression is ZBackup [59].

The most scalable backup solutions leverage compression and deduplication to save space, and various tricks to maximize ingest bandwidth. An example of such a system from the research literature is Data Domain's deduplicating file system [134]. Handling large numbers of objects is a metadata problem, often solved by partitioning the metadata space. An example scalable system which partitions metadata across multiple metadata servers is Druva [37]. Sepaton performs no optimizations on the ingest stream, and thus fully utilizes parallel disk write bandwidth to maximize its ingest rate up to 80 TB/hour. Within 48 hours, Sepaton can ingest up to 4 PB of data.

7.7.6 Backup Format

Backup format refers to how backup data is organized and stored. Although guided by the granularity of state and time, there is flexibility in the format of backups. This format directly affects the run time of queries over historic data acting as a fixed access cost. Also affecting format includes techniques such as deduplication and compression, but those are discussed in the scalability section above (Section 7.7.5).

1. **Full:** A full backup consists of the entire object or set of objects being backed up. All backup systems support taking at least an initial full backup, and then they normally switch to a more efficient backup format. Some backup systems continue to periodically take full backups.

2. **Differential:** A differential backup consists of the changes to objects since the last full backup. Thus, they continuously grow in size since the last full backup. An example backup system supporting differential backup is Acronis True Image [1].

3. **Incremental:** This form of backup consists of only the changes since the last successful backup. They can be incremental at various levels of granularity such as byte- or block-level. An example backup system supporting incremental backups is Deltaic [13].

The choice of backup format impacts three critical resources: (1) bandwidth from the backup client to backup storage, (2) backup storage space, and (3) object retrieval time. Full backups provide immediate access to data, but use the most bandwidth and backup storage space. Differential backups only need one full backup and one differential backup to retrieve data, and they reduce the required bandwidth and backup storage space. Incremental backups minimize the required bandwidth and backup storage space, but require more steps to retrieve backup data.

7.7.7 Types of Storage Targets

The type of storage target has implications for the expressiveness of query that a search system may support. If backups are stored on tape, than quick, random access is impossible. This practically rules out deep search techniques that do not leverage indexes. Today, this is not an issue as most backup systems now use disks [123]. In addition, the type of storage target dictates the most efficient way of storing backups. Fast backup storage might encourage frequent deep, expressive queries over historic data.

1. **Tape:** Tape has been traditionally used for backups. The tar [39] program was designed to archive data to tape.

2. **Disk:** Commonly used by modern cloud services such as Backblaze. Magnetic, spinning disks have the property that sequential access is the most efficient. Thus, large, streaming backups and restores would be desirable similar to tape. However, the quicker random access times enables richer querying, and efficient implementation of deduplication.

3. **Optical:** Optical implies rare access to backups, and would be stored and organized similar to tape. If it is not re-writable, than an implicit write-once semantic for backups is in effect.

4. **Solid State:** Fast, random access and high performance I/O make this option the most attractive for backup systems that need to support frequent deep queries. But, high cost usually rules out this type of storage for backup systems

5. **Cloud-based:** Cloud backup solutions must contend with WAN network bandwidth and availability. They must architect to begin and finish backup jobs asynchronously, and expect disconnections and disruptions during the backup process. An example cloud-based backup system is tarsnap. Often these backups are encrypted because they go to untrusted off-site locations.

The choice of backing storage for backup determines a lot about the strategy and method of backup, and supports or stymies the ability of the backup system to perform frequent deep queries which examine the contents of backup data.

7.7.8 Types of Input Systems

There are different methods of obtaining streams of data for ingest into a backup system. The most widely used methods employ backup agents or services that act from within the system they backup. However, cloud computing has popularized agentless backup of virtual disks via hypervisor-mediated snapshotting.

1. **In-band Agent-based:** This is the predominant form of backup today. One runs a backup program or service inside the system to be backed up. Apple's Time Machine, Microsoft Windows System Restore, and other services all use this model.

2. **Out-of-band Agentless:** This is a newly emerging paradigm for backup, used widely in cloud computing. Because cloud instances are usually virtual machines, and their storage is virtualized, it is easy to snapshot them with hypervisor support. Thus, the most common form of cloud backup actually captures data via agentless snapshotting.

In-band methods have complete visibility into the system they backup, and can more easily integrate into their environments. For example, Apple Time Machine lets users browse through versions of directories and files precisely because it collects and indexes file-level data from within its vantage point inside OS X. Out-of-band methods do not generally have complete visibility into the system they backup. Thus, their backups are generally more opaque and occur at a lower level such as the block-level.

7.7.9 Protection Radius

Protection radius refers to the extent of resilience against different types of failures. It is a function of the operation of a backup system and its architecture including human processes.

1. **Corruption:** Examples include an accidentally deleted file, or accidental mangling of a configuration file. Most backup solutions protect against accidental deletion. Dropbox [64], and btrfs [99] are two example systems that protect against accidental deletion. In Dropbox's case, one can go to a web application to restore deleted files and browse file-level history. btrfs enables the restore and examination of points-in-time over an entire file system, including currently deleted files.

2. **Fault Tolerant:** General hardware faults such as corrupt memory, misbehaving CPUs, or faulty storage layers. Ceph is an example storage system that has built-in fault tolerance, but it is not Byzantine Fault Tolerant [20]. Apple's Time Machine can protect against a locally failing hard drive, whereas btrfs local file system snapshots can not.

3. **Byzantine Faults:** This category includes malicious attacks including infection by malware. Malware could destroy or tamper with backups attached to computing systems. An example type of backup storage which is immune to tampering by malware is tape backups which generally rest in tape libraries without direct access by a computing system.

Backup solutions generally protect from corruption of future data by keeping historic backups. Should the backup system itself experience faults, it becomes increasingly more and more difficult to protect data. This is evidenced by almost no modern backup solutions fully implementing Byzantine Fault Tolerance. Tape backup systems come close because they maintain state offline. Depending on their overall architecture, they may be Byzantine Fault Tolerant.

System	Granularity of Time	Granularity of State	Query	Level of Consistency	Scale	Type of Backup	Type of Storage	Input	PR
Acronis [1]	Discrete	Block	Time	Point-in-time	NS	Incremental	Cloud	Agentless	FT
Attic [56]	Discrete	Fragment	Time	None	NS	Incremental	Disk	Agent	C
Backblaze [9]	Continuous	File	File	None	NS	Incremental	Cloud	Agent	FT
BackupPC [29]	Discrete	File	Time	None	Limited	Incremental	Disk	Agent	C
bup [16]	Discrete	Fragment	Time	None	Limited	Incremental	Disk	Agent	C
btrfs [99]	Discrete	File-system	Time	Point-in-time	IN	Incremental	Disk	Agent	C
Carbonite [18]	Continuous	File	Metadata	None	Limited	Incremental	Cloud	Agent	FT
Ceph [53]	Discrete	Block	Time	Point-in-time	NS	Incremental	Disk	Agentless	FT
Coda [104]	Discrete	File-system	Time	Multi-system	Limited	Incremental	Disk	Agent	FT
Data Domain [134]	Discrete	Fragment	Time	Point-in-time	INS	Incremental	Disk	Flexible	FT
Déjà Dup [32]	Discrete	File	Time	None	Limited	Incremental	Disk	Agent	C
Deltaic [13]	Discrete	Block	Time	Flexible	Limited	Incremental	Disk	Flexible	C
Druva [37]	Discrete	File	Time	None	INS	Incremental	Cloud	Agent	FT
Elephant [103]	Continuous	File	Metadata	Point-in-time	IN	Incremental	Disk	Agentless	C
HAMMER [33]	Discrete	File-system	Time	Point-in-time	IN	Incremental	Disk	Agentless	C
MagFS [68]	Discrete	File-system	Time	Point-in-time	NS	Incremental	Cloud	Agent	FT
NILFS [82]	Discrete	File-system	Time	Point-in-time	IN	Incremental	Disk/SSD	Agentless	C
rsync [73]	Discrete	File	Time	None	Limited	Incremental	Disk	Agent	C
Sepaton [107]	Discrete	Block	Time	None	INS	Incremental	Disk	Agent	FT
Space Monkey [111]	Continuous	File	Time	None	NS	Incremental	Cloud	Agent	FT
System Restore [76]	Discrete	File-system	Time	Point-in-time	Limited	Full	Disk	Agent	C
Retrospect [95]	Discrete	File	Time	None	NS	Incremental	Disk	Agent	C
tar [39]	Discrete	File	Time	None	Limited	Incremental	Tape	Agent	C
tarsnap [26]	Discrete	File	Time	None	NS	Incremental	Cloud	Agent	FT
Time Machine [7]	Discrete	File	File	None	Limited	Incremental	Disk	Agent	C
Veeam [120]	Discrete	Block	File	Point-in-time	NS	Incremental	Disk	Agentless	FT
Venti [93]	Continuous	Fragment	Time	Point-in-time	IN	Incremental	Disk	Agent	FT
ZBackup [59]	Discrete	File	Time	None	NS	Incremental	Disk	Agent	C
zfs [86]	Discrete	File-system	Time	Point-in-time	IN	Incremental	Disk	Agent	C

Table 7.1: Table comparing modern backup systems. In the scale column, I stands for Ingest Bandwidth, N for Number of Objects, and S for Space Requirements. In the PR column, C stands for Corruption, and FT stands for Fault Tolerant.

7.7.10 Modern Backup System Implementations

Now that we have defined important features of backup system design, we catalog modern backup systems and place them within our nine dimensional space. Table 7.1 shows many different modern backup systems. We cataloged backup systems according to their documentation for open source solutions, and advertising materials for proprietary products.

Because of their choice of having cloud-only storage, Backblaze's customers take on average 2 weeks to complete a full backup [9]. This generally only happens the first time they install the Backblaze agent. Space Monkey [111] solves this problem by creating a P2P network between 1 TB backup hard drives hosted by their customers, augmented by the cloud.

For those systems supporting agentless backup, they are either based on snapshotting virtual

storage, or built-in OS-level technology. Acronis and Veeam leverage hypervisor support for point-in-time virtual disk snapshotting. Microsoft's System Restore, Apple's Time Machine, and backup-supporting file systems like Elephant are agentless, but require kernel-level support. Although Elephant is a research file system, there do exist production-ready file systems with backup capabilities such as NILFS, btrfs, zfs, and HAMMER.

Notably rare amongst modern backup systems are those which provide consistency guarantees, continuous granularity, agentless ingest, and rich query capability. Consistent backups are easier to reason about, especially across multiple systems, which is now the common case in distributed cloud computing. Agentless ingest implies zero configuration to maintain. This removes operator errors from corrupting or accidentally disabling backups. Rich query capabilities decrease time to find and retrieve the right backup data, or answer questions about history. By pairing DS-VMI with `/cloud-history`'s index time optimizations, we achieve all of these difficult to implement features except cross-system consistent backups, while not sacrificing flexibility in any other axis. Multi-system consistent backups is a part of the design space not served well by either `/cloud-history`, or any of the other studied backup solutions.