Agentless Cloud-wide Monitoring of Virtual Disk State

Wolfgang Richter

CMU-CS-15-138
October 2015

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy*

This research was sponsored by the National Science Foundation under grant numbers CNS-0614679, CNS-0833882, IIS-1065336, DGE-0750271, and DGE-1252522, the Defense Advanced Research Projects Agency under grant number FA8721-5-C-0003, Intel ISTC-CC, IBM, and Vodafone. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Agentless, Agentless Monitoring, Agentless Cloud Monitoring, Cloud, Cloud Computing, Cloud Monitoring, Deduplication, Deduplicated Snapshotting, Distributed Streaming Virtual Machine Introspection, File Deduplication, File Snapshotting, File Monitoring, Incremental Hashing, Introspection, Optimistic File Snapshotting, Retrospection, Searchable Backup, Snapshotting, Virtual Disk, Virtual Storage, Virtual Machine, VM, Virtual Machine Introspection, VMI

Abstract

This dissertation proposes a fundamentally different way of monitoring persistent storage. It introduces a monitoring platform based on the modern reality of software defined storage which enables the decoupling of policy from mechanism. The proposed platform is both *agentless*—meaning it operates external to and independent of the entities it monitors—and *scalable*—meaning it is designed to address many systems at once with a mixture of operating systems and applications. Concretely, this dissertation focuses on virtualized clouds, but the proposed monitoring platform generalizes to any form of persistent storage. The core mechanism this dissertation introduces is called Distributed Streaming Virtual Machine Introspection (DS-VMI), and it leverages two properties of modern clouds: virtualized servers managed by hypervisors enabling efficient introspection, and file-level duplication of data within cloud instances. We explore a new class of agentless monitoring applications via three interfaces with two different consistency models: `cloud-inotify` (strong consistency), `/cloud` (eventual consistency), and `/cloud-history` (strong consistency). `cloud-inotify` is a publish-subscribe interface to cloud-wide file-level updates and it supports event-based monitoring applications. `/cloud` is designed to support batch-based and legacy monitoring applications by providing a file system interface to cloud-wide file-level state. `/cloud-history` is designed to support efficient search and management of historic virtual disk state. It leverages new fast-to-access archival storage systems, and achieves tractable indexing of file-level history via whole-file deduplication using a novel application of an incremental hashing construction.

I dedicate this work to those that have helped me make my dreams become reality. First, I must thank my maternal grandparents who supported me academically and my, at times, expensive explorations with computers and electronics. Gratitude alone is not nearly enough. I will never forget your legacy. Second, I must thank my paternal grandparents for showing me how people can persevere through the worst of times the world has ever seen and thrive. Third, for my parents who fostered a home filled with the digital wonders of the modern world, and my siblings for always being there for me. Fourth, to Ma and Baba for support and guidance through the fog of stress. Finally, and most importantly, to Debjani Biswas my companion for life, my wife. Thanks for putting up with my long hours, wacky ideas, and going on adventures with me around the world.

Acknowledgments

The PhD process is a herculean effort, of which I am only one small part. Along the way I have been aided by the smartest people at the top of the field of Computer Science. My success is attributable to the great opportunities presented to me by the academic environment fostered at Carnegie Mellon University's School of Computer Science. I have been honored with public support through the NSF, and must thank the American people for supporting scientific efforts across the country. Of course, my interactions with many people along the way have shaped me, formed my research focus, and honed my skills. Here I thank many by name, but there are many more than I could possibly name. Thank you to everyone that interacted with me during my tenure as a graduate student. We all truly stand on the shoulders of giants.

Of course, a large part of my success is due to excellent guidance and tutelage under my advisor Mahadev Satyanarayanan. Over the many years, our group administrative assistants including Tracy Farbacher, Angela Miller, and Chase Klingensmith always made my day better and ensured academic life was smooth. Deborah Cavlovich is amazing taking away every possible headache that arose during graduate school. Without her, scheduling classes, staying on top of PhD administrative overhead, and practically anything else would have been a nightmare. Catherine Copetas is a friend and helper to all students at CMU, she enriched my life by offering me many opportunities to meet new and interesting visitors. Karen Lindenfelser and Joan Digney always helped organize events via the SDI and the PDL. Their efforts made our lives less stressful, and more fun. Of course, conversations and nights well spent with friends led to new ideas and filled my life with fun. Thank you especially Yoshihisa Abe, Leman Akoglu, Brandon Amos, Sivaraman Balakrishnan, Field Cady, Zhuo Chen, Jim Cipar, Bhavana Dalvi, Leigh-Ann DeLyser, Ruta Desai, Bin Fan, Kiryong Ha, Wenlu Hu, Shiva Kaul, Elie Krevat, Anvesh Komuravelli, Jayant Krishnamurthy, Meghana Kshirsagar, Hyeontaek Lim, Michelle Mazurek, Brendan Meeder, Iulian Moraru, Richard Peng, Amar Phanishayee, Kai Ren, Mehdi Samadi, Raja Sambasivan, Vyas Sekar, Vivek Seshadri, Jiri Simsa, Wittawat Tantisiriroj, Ekaterina Taralova, Alexey Tumanov, Vijay Vasudevan, Kevin Waugh, Gabriel Weisz, Lin Xiao, and Erik Zawadzki. For welcoming me to CMU, thank you Matthew Delaney, Jason Calaiaro, and Michael Mackin. Thank you all for the wonderful memories, they will stay with me for life.

Last, but not least, I must thank a lot of my mentors and close research peers. This includes Glenn Ammons, Hrishikesh Amur, David Andersen, Vasanth Bala, Nilton Bila, Sastry Duri, Canturk Isci, Michael Kaminsky, Todd Mummert, Padmanabhan Pillai, Darrell Reimer, and Srinivas Seshan. The research scientists in my group Benjamin Gilbert, Adam Goode, and Jan Harkes have each served as mentors and an inspiration as I developed my systems research skills. Thanks for keeping me technically honest, humble, and holding me to the highest of standards. Each and every one of you has a hand in my future career and success.

Contents

List of Figures

List of Tables

Chapter One

Introduction

The advent of cloud computing coupled with the increasing trend of accessing storage over the network provides us with an opportunity to rethink how we monitor persistent storage. Traditionally, storage devices were tightly coupled with the operating system. The operating system acted as a facilitator of interaction with hardware for userspace applications. This was done primarily for protection and performance. The kernel interacted with storage devices via a driver and enforced constraints which userspace had to obey. This enabled the creation of crash-consistent file systems and robust storage of data culminating with advanced file systems like ZFS [133]. Kernel mediated direct memory access ensured fast reads and writes to persistent storage. Though the storage landscape is entirely different today, we still work with monolithic systems which believe they are tightly coupled to their underlying storage. Especially in virtualized clouds, the virtual machine abstraction, by design, prevents deep understanding of where resources come from.

But this tight coupling is an illusion. Within modern clouds there is either a layer of virtualization between a guest kernel and its storage, or the storage is served over the network. In many cases, both a layer of virtualization *and* a network hop exist between an executing guest and its storage. This reality pervades the enterprise in the form of network attached storage arrays. In the home, media servers are being used to save and serve large files such as movies via network file systems. Thus, it is no longer true that protection and performance are guaranteed by a monolithic kernel. The new reality decouples storage from its executing environment. However, higher performance, better resilience, and stronger protection are achievable if we rethink storage with a modern lens.

Higher performance and better resilience to data loss induced failures are out of the scope of this dissertation. But it is trivial to imagine why they are achievable with modern cloud computing. Clouds horizontally scale services such as storage into large distributed systems. This means individual cloud instances writing into cloud storage can obtain bandwidth far exceeding the bandwidth of a single drive, or even that of a RAID array. By quickly replicating data across cloud datacenters, even large-scale disasters become possible to survive—impossible with the more monolithic, tightly coupled version of storage. Stronger protection is within scope, and the basis upon which a lot of this dissertation was completed. The decoupling of layers in the storage realm enables separation of mechanism, policy, and enforcement. This reflects lessons learned very early on in the networking and OS communities, but never truly applied to storage which traditionally remained tightly coupled from storage device to kernel driver and file system.

Early on in networking the value of separating mechanism and policy, along with a need for gatekeeping network packets, was clearly recognized. Mogul et al. describe implementing user-level packet filtering and inspection [79, 80]. Their design separated the mechanism—kernel-level hooks—of capturing packets, and the implementation—unprivileged, userspace applications—of policy. The importance of separating policy and mechanism was first described in the context of the Hydra research project by Levin et al. [63]. This early work led to the development of deep packet inspecting (DPI) firewalls in networking. DPI firewalls have three attractive properties. First, they are decoupled from the hosts that they monitor except for a network connection. Thus, they have the luxury of being immune to individual host misconfiguration or compromise. In other words, the firewall gatekeepers continue operating even with multiple misbehaving network hosts. In addition, they can not be turned off by the hosts, and the hosts can not affect the policy specified to the firewall. Importantly, these properties hold for both hosts within the firewall's protection boundary, and also for external hosts.

This dissertation explores adding functionality similar to deep packet inspection into the storage layer of cloud infrastructure. Although we use the word cloud, this rethinking of monitoring storage applies in general to storage whether or not it is virtualized. Access over the network technically makes this task easier to accomplish. In this dissertation, much like network-based firewalls, we propose implementing deep file-level monitoring via interpretation of storage writes. Whether in the absence of, or in addition to, a network hop, hypervisors act as a mediator between guest and storage in clouds. This unique position makes them the best location to implement our mechanism.

By placing file-level monitoring within a layer between kernel and storage we gain three properties which are impossible without such decoupling. First, the monitoring becomes immune to guest misconfiguration or compromise. Should a virus infect a guest, it can not hide its tracks if it touches persistent storage. When file-level monitoring is coupled into a monolithic system, viruses can hide. Rootkits which compromise the kernel have the capability of masking their presence entirely. Should an operator misconfigure a guest, file-level monitoring can fail. For example, the wrong permissions set on a log folder causes log analytics processes to fail and not perform their job. Second, guests become protected from bugs affecting file-level monitoring agents, and malicious monitors. No software is bug free, thus we must plan for bugs within file-level monitors—even those designed to protect such as virus scanners [83, 84]. The effect of such bugs is devastating because virus scanners often have root access to perform their job of finding viruses. Should a file-level monitoring process actually have malicious designs on the host it monitors, the decoupling of policy—what a monitoring agent has access to at a file-level—and mechanism—the method of capturing file-level state—provides protection. In a monolithic system, one has no choice but to run monitoring agents at the privilege level they request with direct access to resources. Third, file-level monitoring implementations become easier to scale across different operating systems and applications. They are easier to implement because they do not have to operate within the same environment as the monolithic system producing the writes. The file-level monitoring occurs decoupled from the environment generating the writes.

Thus, we believe the time to rethink storage is now upon us. We also believe that the strong guarantees along with the easier implementation of file-level monitoring make decoupling policy and mechanism paramount for scaling management of storage in the future. Monitoring subsumes a large portion of the required upkeep of individual systems, and is a key problem facing administrators of any computing system [69]. Examples of monitoring include routine virus scanning [21], intrusion detection [118], log file analysis [112], and configuration auditing [119]. In the monolithic model, state-of-the-art monitoring is accomplished with generally third-party agents [51, 60] embedded within the system they monitor.

Embedding a potentially untrusted, third-party agent within the boundary of an enterprise increases its attack surface—the agent may be hijacked by an intruder and used maliciously [83, 84]. In addition, agents consume valuable resources, and are unpredictable in their resource consumption [60]. An agent using excessive memory causes memory pressure and potentially paging to disk, slowing down or even halting critical services. Finally, if a system is compromised, the agent may be tampered with or deliberately fed false information. In the case of a compromise or misconfiguration that is initially undetected, agents can not be trusted at all.

1.1 Problem Statement and Scope

This dissertation rethinks monitoring persistent state in the modern era of virtualized cloud computing, and network attached storage. Based off of the observation that storage is accessed either via a hypervisor or the network, this dissertation proposes deep inspection of disk writes for monitoring purposes. Similar to deep packet inspecting firewalls, this dissertation proposes decoupling mechanism and policy in the storage context. It seeks to answer the primary question of *how do we securely monitor file-level state as generally as possible*? In this context, security means guaranteed enforcement of policy both ways between a monitored system and the system performing monitoring. In other words, monitored systems should never fear the monitoring agents, and monitoring agents should never fear the systems they monitor. Generalizable in this context means generalizing across the combinatorial space of operating system types and applications.

This dissertation describes a mechanism and interfaces on top of that mechanism which enables separation of capturing state and the logic for monitoring that state. Thus, monitoring both online and offline persistent state is within the scope of this dissertation. Enforcement of policies is assumed to be a solved problem within cloud computing. Network quarantining and virtual machine suspension are both pre-existing techniques for isolating and stopping misbehaving cloud instances. Thus, enforcement is outside of the scope of this dissertation.

Online persistent state refers to live file systems as they mutate in real-time within virtual disks. Offline persistent state refers to backups or snapshots of file systems. As a guiding principle throughout this dissertation, mechanism and interface designs must operate without the explicit support of a monitored system. For the remainder of this dissertation, we consider the terms monitored system, guest, cloud instance, and virtual machine to be synonymous and interchangeable.

There are four key challenges enumerated below that this dissertation addresses in answering this fundamental question.

1.1.1 Bridging the Semantic and Temporal Gaps

The normal path for an application system call destined to write to a disk moves from a guest's userspace, to the guest's kernel, and then out to the hypervisor modifying the virtual disk. This path is leveraged by agents as they can hook system calls, register for disk state change notification, trace disk operations, and generally use the guest's kernel to directly accomplish their job. An agentless solution, by definition, does not rely on any guest support whatsoever. Even relying on guest hints makes an agentless method vulnerable to guest misconfiguration or compromise. Agentless methods receive black box operations at the lowest abstraction layer, and must reconstruct or infer higher-level meaning.

The challenge is to bridge this *semantic gap* between black box disk-level operations and their white box file-level interpretation with significantly less context than the guest kernel. In addition, layers often reorder operations for better throughput or latency. File systems generally update data first, metadata second. Thus, there is an additional *temporal gap*–the time between when operations flush to disk, and when they have semantic meaning within the guest file system. Atomic operations, for example, wait until their instructions finish before persisting that their operation finished. The temporal gap is the time between the start of the atomic operation, and the persistence of their completeness.

1.1.2 Bounding Overhead

Monitoring clearly comes at a cost to the system being monitored. We monitor systems for various reasons, but if the monitoring cost exceeds the value of the benefit derived from monitoring, then monitoring does not make sense. Thus, there must be *boundable overhead* for any monitoring solution—agent or agentless. Should monitoring require upgrading the hardware capabilities of all hosts in a production cloud, for example increasing their memory, this cost will more than likely exceed the value of monitoring. In addition, a key metric in virtualized cloud environments is consolidation. Monitoring overheads directly hurt the consolidation factor of a cloud environment by using resources which could have otherwise been devoted to valuable production workloads. In the extreme case, boundable overhead means the capability of completely eliminating overhead. This implies the capability of turning on and off monitoring at will.

1.1.3 Generality

Monolithic design places every component inside the blackbox boundary of the virtual machine. This is attractive because all of the configuration state, necessary libraries, and versions of key system components are all kept together for a specific application. Indeed, this model has proven wildly successful with the use of virtual appliances for deploying software. However, this means that monitoring agents must have implementations and logic for each operating environment they support. The interfaces to an agentless monitoring platform should not needlessly tie themselves to an OS family or specific environment. Agentless monitoring should provide a single set of interfaces to persistent state that not only scales in the *number* of monitored guests, but also in the *types* of file systems and OS's it can monitor. In other words, agentless monitoring must generalize across OS's and applications.

1.1.4 Storing and Indexing Historic Changes

As promised, this dissertation handles not only online persistent state, but also offline persistent state. Due to modern trends in backup architectures providing low-latency, high-bandwidth access to data [81], we believe a critical feature of modern backup should be search—especially with tunable indexes. Backup subsumes so many different systems that a single indexing scheme seems unlikely to fit all present and future search needs. Thus, a key feature of modern backup should be optimized indexing over stored objects. In this dissertation we rethink storage formats for file-level backups given the existence of our storage monitoring mechanism. In addition, we explore techniques for reducing the time to index in an application-agnostic manner.

1.2 Thesis Statement

This dissertation claims that *agentless monitoring of disk state provides stronger security and correctness guarantees than traditional agent-based approaches, is achievable with modest modifications to modern operating environments, and enables generalizability across the combinatorial space of operating systems, libraries, applications, and their versions.* The cost of realizing agentless monitoring of disk state lies directly with the effort needed to deeply interpret virtual disk write operations—specifically the on-disk layout and data structures of file systems. That is to say, it is a capital intensive endeavor, but one that pays off over time. The upfront costs of writing file-system-specific introspection logic are amortized over a long time period. This is because file systems rarely change their on-disk layouts.

The vision of this thesis for cloud computing is that by slightly weakening the strong isolation between virtual machines and their underlying cloud infrastructure, we can provide a more secure environment by taking advantage of the tension between cloud customer, cloud operator, and monitoring vendors. Cloud customers wish to bound the information that monitoring vendors access. Monitoring vendors wish to maintain correct configurations and views of monitored state. Cloud operators wish to increase revenue by offering valuable services to their customers. Thus, the cloud operator is perfectly poised as a mediator guarding paying customers from monitoring vendors, and providing information independent of customer misconfiguration or compromise to the monitoring vendor. Monitoring vendors benefit with greater name recognition via the cloud marketplace, and interfaces which ease their generalization across cloud customer operating environments.

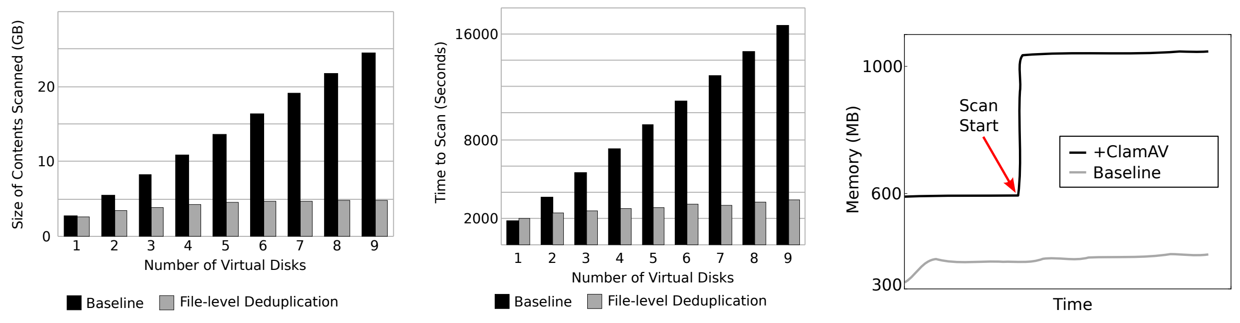
The research questions this dissertation addresses are: (1) What is the minimal set of extensions to modern environments needed for agentless persistent state monitoring? (2) How does agentless monitoring of persistent state change the implementation of file-level monitoring? (3) How does agentless monitoring of persistent state change the implementation of snapshotting? (4) How can backup systems minimize the time to index all stored objects and their versions?

1.3 Contributions

This dissertation challenges the status quo monolithic system design which embeds monitoring agents into the systems they monitor, with an agentless vision separating the mechanism of monitoring from the policies which act upon monitored information. In so doing, this dissertation disrupts a billion dollar industry [40]. We show that the long path towards an agentless world, with all of its benefits, is in fact tractable. It is feasible with most modern hypervisors to pursue today, in some cases with minimal disruption to existing deployments and operations. Should the mechanism described in this dissertation become widely available, switching to agentless monitoring is very difficult to argue against.

Agents can not fundamentally offer the guarantees of an agentless approach. No agent-based solution can guarantee isolation from monitored system misconfiguration or compromise. By implementing policy with mechanism inside the monitored system, corruption of either can directly affect the other. No agent-based solution can guarantee that its bugs will not affect the monitored system. In fact, guaranteeing bug freeness is undecidable without constraints. No agent-based solution easily generalizes across operating environments. Because of the deep coupling of agent and the monitored system, agents must needlessly cater to their environment rather than just their task. Thus, if the technology espoused in this dissertation is widely implemented, it becomes very difficult to justify the further use or development of agent-based monitoring solutions for persistent state. Agentless solutions also enable optimizations which are very difficult to implement with agent-based solutions. Agentless monitoring has the capability of leveraging global knowledge about collections of systems and resource allocations. For example, agentless monitoring can deduplicate across collections of virtual disks and schedule monitoring tasks efficiently with production VM workloads without mixing resource allocations.

Figure 1.1 shows the wins with an agentless model that leverages global knowledge—file-level duplication across VMs—for a file system scanning workload. In this example, traditional scanning agents execute independently inside each VM. It would be non-trivial and also inefficient for each type of scanning agent to implement file-level deduplication independently. As reflected in Figure 1.1(a), a virus scanning agent reads duplicated files multiple times wasting significant read-bandwidth. Processing time is also wasted scanning the same files over and over again as reflected in Figure 1.1(b). Simply running an agent uses resources of the VM—in this case significant amounts of memory as shown in Figure 1.1(c). Instead of wasting this memory n times for VMs which might all have the same files, an agentless implementation could bound the memory usage devoted to virus scanning across the entire cloud. These wins are not limited to file system scanning workloads. First, the cloud can separately bound global resources devoted to *any* monitoring workload by decoupling its execution from individual VM execution. Second, deduplication also benefits streaming workloads. For example, agents monitoring changes to configuration files could receive coalesced updates from multiple VMs across the cloud rather than processing duplicate file-level changes individually. These optimizations enhance the scalability of future clouds, and directly lead to lower total cost of ownership and higher revenue for all parties involved.



(a) Aggregated read bandwidth. (b) Aggregated processing time. (c) Memory of a single VM.

Figure 1.1: ClamAV's [21], a file system scanning agent, memory overhead and aggregated resource usage (Figures 1.1(a) and 1.1(b) reconstituted from [124]).

The contributions of this thesis, in addition to answering the aforementioned research questions, are:

- Demonstrates feasible, efficient, generalizable agentless monitoring of disk state (Chapter 2)

  Full reference implementation released under the Apache v2.0 License

  Support for the `NTFS`, `ext4`, and `FAT32` file systems

  API integration into the OpenStack cloud software

- Demonstrates three generalizable interfaces to disk state (Chapters 3, 4, and 5)

  `cloud-inotify` – for real-time agents (publish-subscribe interface, live writes)

  `/cloud` – for scanning/batch agents (legacy applications, file system interface, on-disk)

  `/cloud-history` – for historic data access (file-level deduplicated snapshots, archived)

- A backup study of a research group with 58 unique hosts over 1 year (Chapter 5.2)

  3,268 file system snapshots

  1.676 billion referenced files

  146 TiB of crawled file state

- A log-structured format for `/cloud-history` based on agentless monitoring (Chapter 5.5)
  - Heuristic time-based snapshotting of files
  - Garbage collection of log state
  - Application-agnostic indexing speedup via whole-file deduplication

Chapter Two

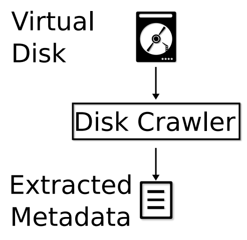
Distributed Streaming Virtual Machine Introspection (DS-VMI)

In this chapter, we explain the design and implementation of our mechanism that exposes live virtual disk state in near-real-time to monitors. The mechanism was designed to operate without any support from within the monitored system—the guest operating system in a VM. This means that it operates without paravirtualization, guest modifications, or specific guest configurations. We call this mechanism *distributed streaming VM introspection* (DS-VMI). DS-VMI infers file system modifications from sector-level disk updates in near-real-time and efficiently streams them to distributed or centralized monitors.

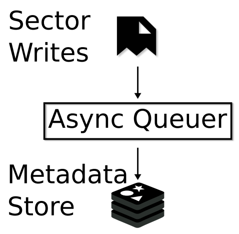
Our approach is based on the fact that virtual disks are emulated hardware. Hence, every disk sector write already passes through at least the hypervisor system. Remember that it may pass through many hosts depending on whether or not the backing storage is across the network. We transparently clone this stream to a userspace process on the hypervisor host, or any of the intermediate hosts. This minimally interferes with the running VM instances—generally they occasionally see higher latency writes. Only guest-flushed updates result in sector writes. Thus, we only handle file system updates that are flushed from the VM instances. Updates that have not been flushed, and therefore represent dirty state in guest memory, are outside the scope of this dissertation.

DS-VMI resembles classic VMI [42], with two crucial differences. First, we support streaming introspection from instances distributed throughout the cloud. The design of DS-VMI and its interfaces directly stems from this distributed setting. Second, instead of performing introspection synchronously, we always perform it asynchronously. We are able to minimize stalling of the VM during introspection because our goal is not intrusion detection: we are only monitoring guest actions, rather than trying to prevent tainted ones by enforcing policy. As stated in Chapter 1, enforcement of policy is outside the scope of this dissertation.

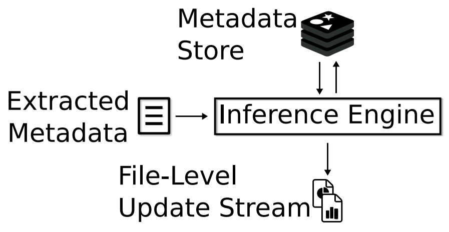
The rest of this chapter is organized as follows. Section 2.1 provides a very high-level view of the architecture of DS-VMI. Section 2.2 describes the requirements necessary from a hypervisor, file system, and guest kernel for DS-VMI. These requirements include theoretical underpinnings



(a) Disk Crawler: Offline or live, produces metadata used for inference.



(b) Async Queuer: Queues instance writes into an in-memory queue.



(c) Inference Engine: loads metadata and interprets queued writes.

Figure 2.1: Three-stage DS-VMI architecture.

grounded in a theory of file systems, and technical requirements such as guest kernel parameters. This section culminates with the description of an implementation of cloning writes using a modern hypervisor. Section 2.3 deals with bootstrapping metadata about file systems within a storage device, primarily a virtual disk, necessary for the DS-VMI runtime. It includes a discussion of the three file systems we support: `NTFS`, `ext4`, and `FAT32`. Section 2.4 describes the asynchronous architecture and an optimization to reduce overhead. Section 2.5 describes the DS-VMI runtime from receiving a sector-level write, to understanding its file-level implications. Section 2.6 describes how DS-VMI supports live attachment to an already running VM, and also detachment for when operators wish to drop overhead to zero. Section 2.7 describes extending the API of an existing cloud to integrate introspection. An evaluation of the overhead of this mechanism on four representative file-level workloads is provided in Section 2.8. We finish with a description of what DS-VMI is *not* designed for, and its fundamental limitations in Section 2.9.

2.1 Overview of the DS-VMI Prototype

We have built an experimental prototype of DS-VMI, for the KVM hypervisor [58] using QEMU [11] for disk emulation. Via custom introspection code, our prototype supports commonly-used file systems including `ext4`, `NTFS`, and `FAT32`. We show that our solution works out of the box with unmodified cloud images of Ubuntu Server 14.04 LTS as provided by Canonical [17]. Our prototype has a three-stage architecture. The first stage is an indexing step, performed once per unique virtual disk (not needed for clones), for initializing. The other two stages are specific to the runtime of each VM instance executing in a cloud, as shown in Figure 2.1. We summarize these stages below, with details in Sections 2.3 through 2.5:

1. **Crawling and indexing virtual disks. (Figure 2.1(a))** This stage generates indexes of file system data structures via a component called *Disk Crawler*. The indexes are generated live or loaded at instance launch from a central store such as the image library storing virtual disks.

2. **Capture and cloning of disk writes. (Figure 2.1(b))**
   This stage is implemented via a userspace helper process called *Async Queuer* that receives a stream of write events from a minimally modified QEMU. Normally, it runs at the hypervisor hosting the VM for low latency, but it can technically receive this stream over the network.

3. **Introspection and translation. (Figure 2.1(c))**
   In this stage, the *Inference Engine* interprets sector writes, reverse maps them to file system data structures, and produces a stream of file update events. It operates either at the hosting hypervisor, or across the network.

To ground our discussion, lets follow an example write originating within a guest VM, and to better understand how monitoring might work, imagine a monitor application interested in monitoring a file called `/home/monitorme/clock.jpg`. Imagine that the `clock.jpg` file gets updated every 5 seconds by a webcam pointed at a clock with a second hand. Thus, our monitor should see modifications to this file every 5 seconds. DS-VMI can freely discard the rest of the virtual disk I/O because no other registered monitor exists.

Let us examine what happens when the file is modified, and for brevity we follow a single virtual disk block write. First, an instruction executed by the guest VM traps into the KVM kernel module as shown in Figure 2.2 by the arrow moving out of the box labeled, "Execute natively in Guest Mode," into the box labeled, "Handle Exit." The KVM kernel module identifies whether or not the trapped instruction is for an I/O operation. Because it is an I/O operation, the KVM kernel module invokes the userspace process emulating I/O devices for the guest VM—in this case a Qemu process. The steps described here are highlighted in Figure 2.2.

Before issuing the ioctl to the KVM kernel module to return to guest mode, the write is copied to the DS-VMI process running as a set of userspace processes not shown in Figure 2.2. It arrives at the Async Queuer shown in Figure 2.1(b). At this point DS-VMI takes over analyzing the write to determine its file-level implications in the introspection phase shown in Figure 2.1(c). The first step requires reverse mapping the partition table. DS-VMI identifies that the write is within a partition of interest—specifically the `ext4` formatted partition containing the `clock.jpg` file.

The write is then passed to an `ext4` specific handler that was initialized by crawling the containing virtual disk. The handler performs a series of reverse mappings to understand the individual file being modified and its full path. The first step in the process is to identify if the block represents data or metadata. Metadata for `ext4` includes the superblock, the block group descriptor table, inode bitmaps, block bitmaps, inode tables, and extent trees. In this case, the write is data so the first reverse mapping yields the inode responsible for this data block. The next reverse mapping yields the file name `clock.jpg` contained within the directory data block for directory `monitorme`. The directory data block reverse maps to an inode and this process recursively continues for the two other parent directories: `home`, and `/`. In the actual implementation, we optimize this lookup with a reverse index on full path names which avoids the recursion.

Thus, the third phase of DS-VMI as shown in Figure 2.1(c) has performed four reverse mappings: one for the initial data block to the responsible inode, and three for the three parent directories. DS-VMI now knows that this data block belongs to a file not a directory, and that the full path of the file is `/home/monitorme/clock.jpg`. The next step is to determine if any registered monitors are interested in this file-level update. In this example, there is a registered monitor for this file and DS-VMI uses interprocess communication via `cloud-inotify` (Chapter 3) to notify the monitoring process that there is new data to consume. The monitoring process receives the data block, updates its copy of the file, and refreshes the screen if enough new data has been written. There may be more block writes that are needed before the file is displayable. This process repeats every 5 seconds as new images are written to disk. If data blocks are written without metadata structures pointing to them, they must remain buffered by DS-VMI until it associates them with a file. For exposition, we assumed the data block was immediately associable with a file.

2.2 Hypervisor, Kernel, and File System Requirements

If virtual writes cross a network, introspecting them exactly like a deep packet inspecting firewall becomes the most logical choice. We assume techniques are already known for building such solutions. The difference between analyzing network traffic and introspecting file systems comes down largely to one of memory. Large amounts of memory may be needed for buffering writes and reconstructing complex file system metadata. If we ignore the more trivial network case, we are left with the more difficult position of adding introspection support to a hypervisor. This approach is explored in this dissertation, and this section specifically answers the question *what are the properties of the hypervisor, kernel, and file system which make correct introspection feasible*? By correct, we mean that DS-VMI provides guaranteed consistent views of file-level state.

2.2.1 Hypervisor Virtual Storage Hooks

We assume that either virtual disk writes come over the network, which makes them amenable to deep packet inspection techniques, or the hypervisor provides hooks to inspect each write. For correctness guarantees in later sections to hold, it is important that the hypervisor not coalesce or drop writes. Otherwise properties which we assume true about the guest kernel or guest file system may prove false due to hypervisor optimizations. Thus, we assume the existence of a method to essentially duplicate each virtual write to DS-VMI.

This is not a giant leap. Many modern hypervisors already support targeting more than one virtual storage device in RAID 1 style configurations. For example, KVM has a built-in command called `drive-mirror` [92]. The purpose of this command is to mirror a virtual disk to another location, and it has a mode for duplicating writes. As another example, VMware has a storage driver called Mirror [122]. Its job is to mirror a drive to another location, and it accomplishes this by duplicating live writes. For yet another example, Xen has a highly configurable `blktap` API [90] for writing custom virtual disk implementations. Using `blktap`, one could rapidly implement mirroring as needed by introspection.

Of course, configurations exist which neither send writes over the network, nor through the costly layers of a hypervisor. Performance critical VMs have storage directly dedicated to them. In such a configuration, the writes and management of storage are directly handled by the VM. Obviously, DS-VMI can not operate in such a setting. However, generally DS-VMI does not apply in such settings anyways, and these configurations are rare in cloud computing. First, the performance sensitive application probably can not tolerate the potential latency and overhead from deep monitoring. Second, such direct access configurations are rare because they enable hardware-level attacks by guests, prevent useful management services such as snapshotting by the cloud, and expose the guest to underlying hardware failures. If a guest ties itself to local hardware, migration becomes impossible should that hardware begin to fail.

Extending an Existing Hypervisor

Figure 2.2 shows the design of KVM's I/O path when guest I/O's trap into the hypervisor. By design, all I/O's get handed over to a userspace helper program—QEMU. Such a design is ideal for DS-VMI because the writes already come to userspace for inspection and further routing—very similar to the proposed design of packet filters [80]. We just need to copy the write stream to a userspace DS-VMI process from the emulator of the disk. Although QEMU handles both reads and writes, we only need to introspect write operations.

We use the open source virtual storage introspection engine for QEMU/KVM as described in [98] with a few key differences which make this feasible within a cloud framework such as OpenStack [85]. The original implementation used QEMU's tracing framework to capture writes. QEMU's tracing framework was designed for debugging purposes, and is not built into the QEMU executable by default. No vendor provides production QEMU builds with tracing activated—indeed QEMU advises against including tracing into production builds. Thus, the original method of capturing disk writes was unsuitable for production deployments especially in modern clouds.

Our new implementation [126] changes a total of 50 lines of C within QEMU, and extends a pre-existing hypervisor command called `drive-backup` [91]. The QEMU command `drive-backup` originally implemented copy-on-write, point-in-time virtual disk snapshotting. We added a mode to `drive-backup` called `continuous` which duplicates every write to another target. When introspecting, we set this target to a network drive using the Network Block Device (NBD) protocol. We have a compact, less than 1,000 lines of C code, implementation of NBD which passes writes along not to a real storage device, but instead to the DS-VMI framework.

When `drive-backup` executes at a QEMU hypervisor, writes get duplicated over a TCP socket via the NBD protocol as shown in Figure 2.3. This provides flexibility—introspection may run locally or remotely depending on the load of a node hosting a guest VM. The writes terminate at the NBD Queue as shown in Figure 2.3. Presumably this queue resides on the same node performing introspection, although technically the introspection process can also exist on a separate node, accepting raw writes over TCP.

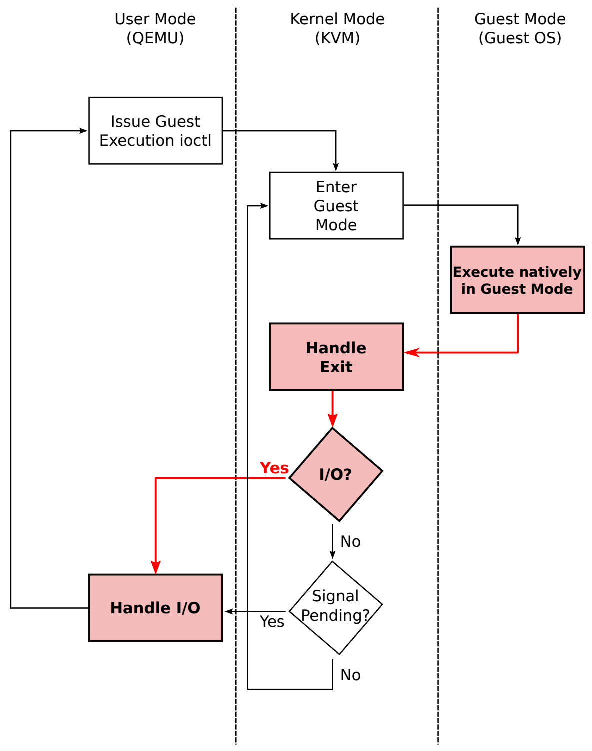


Figure 2.2: KVM architecture showing userspace, kernel, and physical boundaries. Figure reconstituted from [58].

2.2.2 Guest Kernel Invariants

Kernels are free to reorder and coalesce writes from file systems as long as they do not violate correctness. However, ordering of writes and observing all intermediate state is necessary to fulfill the file system invariants described in Section 2.2.3. Hence, kernels must respect these invariants otherwise DS-VMI may lose track of critical file system state. In addition, the tunable timeout before disk flushes are forced directly affects the minimum latency in DS-VMI. Thus, there are two questions explored in this section. First, *what are the minimal set of invariants needed from a guest kernel to ensure DS-VMI correctness*, and *how can a guest kernel assist DS-VMI*? Although DS-VMI requires no modifications whatsoever, it seems likely that either guest kernel tuning for better introspection, or direct guest modifications à la paravirtualized kernels will become standard for introspecting clouds.

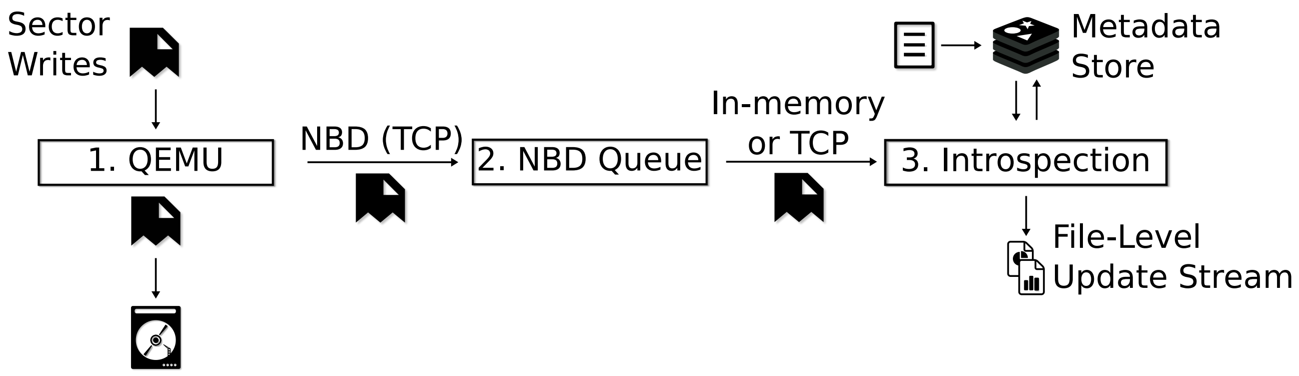


Figure 2.3: QEMU duplicates writes over a TCP socket using the Network Block Device protocol as its application-layer protocol. These writes queue for introspection, and are transferred for introspection either via an in-memory transport or another TCP hop.

Write Buffer Flush Frequency

Operating systems introduce the notion of a page cache to optimize both the read and the write path of slow persistent storage. Reads often exhibit temporal locality—the notion that there is a high likelihood of accessing recently read data in the near future again. Writes also exhibit temporal locality, thus sending every single write to a slow storage layer is not logical. Page caches serve as a fast, in-memory store of data for reading and writing backed by persistent storage. Temporal locality magnifies their benefit. They directly limit the writes visible to DS-VMI, and add additional latency from the time of write to its storage-level visibility.

At a minimum, DS-VMI requires just enough writes to be visible such that the file system invariants as laid out in Section 2.2.3 are maintained. Ideally, to minimize latency and the chance of violating a necessary file system invariant, the kernel disables its page cache for perfect visibility of every write to DS-VMI. However, the performance cost of disabling a page cache is prohibitively costly, and we can not expect this to become a best practice.

This extra latency causes incorrect decisions based on old information. For example, in elastic cloud computing scenarios, compute elements often scale to ensure service level agreements (SLAs). By acting upon old information, they scale to handle the load of the past—not the load of the present. Thus, there is a high likelihood of underprovisioning or overprovisioning when information is not exposed quickly to the introspection infrastructure. Overprovisioning directly causes waste, and underprovisioning indirectly causes waste by delaying or causing failures in dependent systems.

Either paravirtualized drivers communicating information to the hypervisor before it gets flushed, or tuning of guest kernel flush parameters to make them more frequent is needed to ensure timely introspection results. For example, `ext3` forces flushing of metadata every 5 seconds by default. On the other hand, `ext4` forces flushing of critical data every 30 seconds by default. This led to the discussion of using `fsync` much more aggressively by userspace [28]. These default cache flush intervals fundamentally limit the minimum latency of an introspection framework. Note that for *correctness*, latency tuning is not necessary.

Write Barriers and Ordering

Kernels reorder writes for various reasons, often for minimizing the seek time across a set of writes. The famous Elevator algorithm [62] tries to order and dispatch writes in a minimal seek time ordering based on the current direction of the disk head. As mentioned in the previous section, in the worst case this reordering could lead to confusing DS-VMI. However, if file system invariants are left intact, which they must be to maintain file system consistency, DS-VMI will be unaffected by kernel-level write reordering. In fact, to enforce file system invariants, the kernel offers write barriers. Write barriers ensure critical operations are flushed to disk with a certain ordering.

Write barriers are a technique for a kernel to ensure that file system metadata structures get updated in the proper order on disk. For introspection, write barriers ensure correctness by providing tight guarantees on the ordering of metadata updates. Introspection correctness is dependent on the guaranteed ordering of metadata updates from a guest kernel. The requirements for file system correctness are analyzed in the next section.

2.2.3 File System Invariants

Traditional file systems were not designed to support real-time introspection as a consumer of their writes. The properties a file system must guarantee to make real-time introspection possible are not obvious, but have been studied by the file system theory community. Sivathanu et al. [109] derive the properties a file system must exhibit for this type of file-level inference. The guarantee a file system must maintain, from [109], is,

$$\{t\,(A^x) = k\}_D \Rightarrow \{t\,(A^x) = k\}_M$$

In words, the type (k) of a block (A) on disk (D)—metadata or data—matches the file system view in memory (M) at some point in time (x). This ensures that incorrect inferences can not occur; a data block associated with a file can not be mistaken for a metadata block associated with the same file. For this guarantee to hold, a file system must exhibit, "a strong form of reuse ordering," and metadata consistency [109]. *Strong reuse ordering* means that the file system must commit the freed state of any block and its allocation data structures to disk before reuse, and *metadata consistency* means maintaining all file system metadata with a set of invariants [109] (e.g. directory entries point to valid file metadata) to ensure correct operation. A practical example occurs upon file deletion. The data blocks of a file must be marked as free and their corresponding inodes also marked free before any of those blocks are reused on disk. If they are not marked as free on disk before reuse, DS-VMI might incorrectly believe that their type, and therefore their contents, have not changed upon future writes.

What class of applications could transition to using the file-level update stream provided by DS-VMI? Any application that can resume or operate on data flushed to disk works with both whole disk snapshotting and DS-VMI. Recent research [46] reports that modern desktop applications frequently flush data to disk which means many common applications already work with snapshotting and by extension DS-VMI. Abstractly, a disk flush requires buffered I/O operations to be flushed to disk and the file system to have both metadata consistency and data consistency on disk after the flush. *Data consistency* [109] means that all flushed data safely resides on disk and the contents of the corresponding data blocks match the metadata structures pointing to them. Following a reboot, an application reading from the file system sees the side effects of all flushed I/O operations. Our technique preserves flushed file-level updates, which means applications properly flushing critical data to disk can safely use data obtained via DS-VMI.

Order of Metadata Updates

Given the analytical and theoretical underpinnings of the last section, how do we know whether or not a file system supports them? The easiest method is auditing that file system's source code. For `ext4` we have this luxury, for `FAT32` and `NTFS` we have no such luxury and must hope that they abide by certain rules. All that a guest kernel must ensure is *strong reuse ordering*, and *metadata consistency*. Basically, all metadata block type changes must directly match the view of the guest kernel's page cache at commit time in the near future. According to Sivathanu et al. [109], file systems such as `FAT32`, and `ext3` have features we desire. However, their study is not exhaustive and without access to source code difficult to prove for `NTFS`. Based on observations, we have gained high confidence that `NTFS` is crash consistent and therefore safe for DS-VMI.

2.2.4 Correctness of DS-VMI Relative to Snapshotting

DS-VMI offers the same fidelity at a file-level as current state-of-the-art snapshotting methods. We never report a file-level update that has not occurred within a file system on a virtual disk—there are no *false positives*. In addition, we never miss a file-level update that has been flushed to disk—there are no *false negatives*. If a snapshot of a disk is taken at a point in time, the inferred file-level updates DS-VMI reports are consistent with file-level changes observable in that snapshot. For example, if a tool such as Tripwire [118] runs on the snapshot and runs on a virtual disk maintained with updates from a stream of DS-VMI provided file-level updates, the results are identical. This does not mean that our technique provides the equivalent of a snapshot at a block-level.

Figure 2.4 superimposes our technique over snapshotting, and we use this figure to develop an example illustrating the difference between inferring file-level updates and snapshotting. Figure 2.4 shows an initial snapshot created at time τ, and another snapshot at time τ'. Assume that these snapshots occur at a block-level, which is how snapshotting virtual disks happens today. The snapshot at time τ' differs from the snapshot at time τ by exactly the disk blocks that were written in the intervening time period. This is usually accomplished via a technique called copy-on-write which copies disk blocks only when overwritten. Instead of providing a stream of disk block writes,

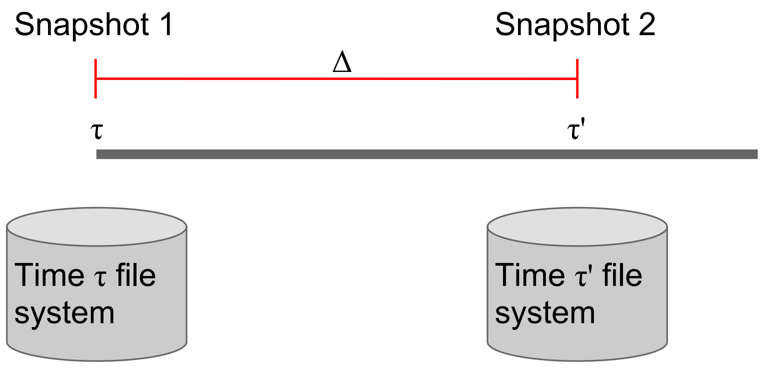


Figure 2.4: Combining file-level updates recorded as Δ with the file system at time τ yields a file system equivalent to the file system at time τ'. The file systems may be mutating state on disk when the snapshots occur.

our technique provides a stream of file-level updates in between these two time points which we refer to as Δ. Thus, Δ is a *well-ordered set* of inferred file-level updates. Applying our stream of file-level updates Δ to the snapshotted file system at time τ yields an equivalent view of the file system within the snapshot taken at time point τ'. With high probability, this view will not be consistent at a block-level with the snapshot at time point τ', and an example scenario leading to inconsistency is explained below. However, this does not mean that anything is lost *semantically* from the snapshot method.

Snapshotting techniques do not understand disk blocks at a semantic level, thus they blindly follow all disk block writes. File-level updates are reported when disk block writes are visible from a file system perspective. If disk block writes are invisible from a file system perspective, they provide no higher semantic meaning because they are uninterpretable and contribute no information to the file system. Disk block writes that are missing from inferred file-level updates are the file system invisible disk block writes. One example that leads to an inconsistency is file creation that fails to complete before the snapshot. The snapshot contains disk blocks associated with the file, but, if mounted, the file would not appear in the file system because the creation was not completely flushed to disk. A snapshot derived from inferred file-level updates would not contain disk blocks associated with the file because its creation did not complete.

2.3 Crawling Initial Virtual Disk State

DS-VMI requires a one-time crawl of each virtual disk before it commences real-time streaming of subsequent file system updates. This crawl builds a map of the virtual disk for DS-VMI so that it can very quickly infer the file system objects being modified from incoming sector updates at runtime. DS-VMI supports crawling offline virtual disks in addition to dynamically crawling online, running

```
{
‘type’       : [BSON_STRING, 3] ’mbr’
‘gpt’        : [BSON_BOOLEAN] false
‘sector’     : [BSON_INT32] 0
‘partitions’ : [BSON_INT32] 1
‘mbr’        : [BSON_BINARY, 512] ...
}
```

Figure 2.5: An MBR BSON document example showing details about an entire virtual disk, starting from the MBR.

VM instances.

The offline case typically occurs when a virtual disk is first added to a cloud. Upon addition of the virtual disk, the *Disk Crawler* produces serialized metadata associated with the virtual disk’s partitions and stores it alongside the virtual disk in a virtual disk library. It only needs to run once per unique virtual disk.

In the online case, DS-VMI live-attaches to an already-running VM. Here, the Disk Crawler crawls and indexes the virtual disk while it is also being modified by the executing VM. To handle the transient dirty state and not miss any new state, the dynamic component of DS-VMI, described in Section 2.4, buffers the incoming write stream from the start of the disk crawl. Once the Disk Crawler finishes indexing, DS-VMI replays the dynamic write stream buffer to obtain the latest updates since the time of crawling and finally catches up to the live, real-time write stream updates.

The disk crawler is implemented in C with file system indexers for `ext2`, `ext3`, `ext4`, `NTFS`, and `FAT32`. The entire disk is crawled, and serialized metadata is produced for each active partition containing a valid supported file system. The metadata is formatted as serialized BSON [15] documents and compressed using `gzip`. We chose BSON because it is compact, supports binary data, has an open specification, and has been successfully used in scalable systems such as YouTube [48].

Figure 2.5 shows a sample BSON document. We do not construct a single nested BSON document per drive because that would require loading the entire document into memory, and thus would not scale to large virtual disks. Instead, we represent the metadata as a collection of documents, each of which can be loaded separately. We currently serialize five document types: MBR, Partition, File System, BGD (not used with `NTFS`), and File. A File document contains a serialized form of the related inode and, in the case of a directory, a list of files in the directory.

Disk analysis starts at the Master Boot Record (MBR) that contains a partition table. Each entry in this table may point to a valid primary partition or to a linked list of secondary partitions. As an illustrative example, we consider what happens when an `ext4` partition is detected. An `ext4` partition is analyzed by first examining and serializing its superblock. The `s_last_mounted` field identifies the most recent mount point of this file system, which helps in recreating pathnames. The superblock points to the Inode Table, which captures a wealth of information about each file. In `ext4` the “`i_block`” field is typically the header of an *extent tree*. Immediately following the header are pointers to extents, each of which in turn points to a set of data blocks for the file. The disk crawler collects and serializes necessary metadata from all allocated files by walking the inode table and directory entries. Directory entries are contained in the data blocks of directories and map file paths to inodes.

The `NTFS` disk format poses special challenges. In this format, the Master File Table (MFT) plays a role analogous to the Inode Table in `ext4`. It stores File Records, which are the equivalent of `ext4` inodes. However, the MFT itself is managed as a file and can become fragmented throughout a disk. The positions of metadata cannot be computed in advance with simple offsets. In addition, there are proprietary intricacies that are not documented openly and can only be inferred via the trial and error process of reverse engineering. In spite of these challenges, we have been successful in implementing support for NTFS.

The `FAT32` disk format is the simplest of the three main file systems considered, but comes with its own challenges. For example, there are no separate inodes for files. File metadata is embedded within the containing directory. This means that it is impossible to assign unique identifiers to files in `FAT32` that are divorced from path. In spite of this, `FAT32` is well and openly documented now, and was the quickest to implement—by someone originally unfamiliar with introspection and our pre-existing codebase.

2.3.1 Impact on Virtual Image Library Operations

As described at the beginning of this chapter and in-depth in the last section, DS-VMI requires metadata from the file systems present in a virtual disk. This metadata is then used for inference when VM instances are instantiated. But how will a modern cloud obtain this metadata? Virtual disk libraries already collect metadata and act upon metadata during two key operations: check in of a new virtual disk, and checkout during instance creation.

DS-VMI's metadata naturally fits with these two operations. DS-VMI extends the check-in process with a crawl over the virtual disk to extract relevant metadata from guest file systems. When a cloud user requests that a VM instance boot from a particular virtual disk, we can retrieve the crawled metadata, and ship it to DS-VMI anywhere in the cloud. The rest of this section explores the feasibility of this approach with standard VM images by answering the following question:

What is the overhead of crawling for, transferring, and loading metadata? With our prototype implementation we found typical crawl time of 18-26 seconds, metadata size of 8-20 MB (compressed), and metadata load time of 30-73 seconds with a standard Ubuntu 12.04 LTS server virtual disk image, and a Windows 7 virtual disk image. Keep in mind that both disk images are 20 gibibytes in size, which takes approximately 3 minutes to transfer assuming an unloaded 1 gigabit network link. Moving our 20 mebibytes of metadata—0.2 seconds—and loading it in 73 seconds is dwarfed by the large virtual disk transfer time. In other words, the load has plenty of time to finish while the disk streams in the background. Of course, this assumes loading is a synchronous operation and we must wait for it to complete. We can instead buffer writes while the load continues in the background, and begin processing the buffered writes once the load completes.

`ext4`

Used (GB)	Raw (MB)	gzip (MB)	Crawl (s)	Load (s)
6.4	64	8.2	20.30 (1.91)	30.15 (0.15)
8.4	70	9.4	21.43 (2.00)	35.98 (0.20)
11	77	11	22.25 (1.68)	41.54 (0.25)
13	83	12	23.20 (1.85)	47.24 (0.45)
15	90	13	24.22 (1.74)	52.91 (0.43)
17	96	15	25.85 (1.83)	59.19 (0.43)

`NTFS`

Used (GB)	Raw (MB)	gzip (MB)	Crawl (s)	Load (s)
6.9	67	14	17.93 (2.12)	43.74 (0.20)
8.9	73	15	18.13 (2.01)	49.58 (0.23)
11	79	16	18.39 (2.26)	55.10 (0.24)
13	85	18	18.51 (1.85)	60.60 (0.36)
15	92	19	18.72 (2.01)	66.21 (0.65)
17	98	20	19.48 (2.47)	72.68 (0.82)

Table 2.1: Metadata size uncompressed, compressed, crawl time and load time into Redis (20 runs) as a function of used virtual disk space. Used is used disk space reported by `df`, Raw is the uncompressed metadata, gzip is compressed metadata with `gzip --best`, Crawl is the time taken to index a disk, and Load is the time to load metadata into Redis.

Checking in an Image

The critical factor affecting the check in process of a VM image into a virtual disk library is crawl time. Crawl time is dictated by the amount of metadata in the file systems of the virtual disk, its sprawl across the disk, and the amount of lookups required to unpack the file system metadata structures. Large amounts of sequentially packed metadata take time only in unpacking the data structures of the underlying file system. Metadata spread throughout the disk causes many seeks and could potentially slow down the crawl process. To examine this effect, and also to get a sense of the overhead of crawling in general, we measured crawl times with two file systems: `ext4`, and `NTFS`. `NTFS` packs all of its metadata into a single global area of the disk known as the Master File Table (MFT). In contrast, `ext4`, typically spreads metadata across block groups. Thus, for crawling, we expect the central, sequential MFT in `NTFS` to perform better.

The metadata collected by the Disk Crawler grows with used disk space because it serializes a mapping from sectors to files, and with the number of reachable live inodes representing files in the system because each inode is serialized as well. Table 2.1 shows how metadata grows as a function of used disk space for `ext4` and `NTFS`. We increased used disk space by writing a single large file with random data inside a virtual disk. The relationship for both file systems is linear in the used disk space because we use a canonicalized form of metadata independent of file system. The raw `ext4`

inodes	Raw (MB)	gzip (MB)	Crawl (s)	Load (s)
127,786	64	8.2	20.30 (1.91)	30.15 (0.15)
250,000	101	12	21.19 (1.85)	41.53 (0.32)
500,000	178	19	22.52 (1.29)	63.56 (0.29)
750,000	256	27	23.87 (1.68)	85.87 (0.61)
1,000,000	333	35	26.08 (1.75)	109.68 (0.68)
1,250,000	410	44	27.05 (1.47)	132.12 (0.68)

`NTFS`

inodes	Raw (MB)	gzip (MB)	Crawl (s)	Load (s)
103,152	67	14	17.93 (2.12)	43.74 (0.20)
250,000	106	16	24.36 (2.64)	58.53 (0.24)
500,000	174	19	34.95 (2.19)	83.02 (0.29)
750,000	242	23	44.04 (2.93)	108.31 (0.56)
1,000,000	309	26	54.62 (2.65)	132.96 (0.66)
1,250,000	377	29	63.99 (2.52)	159.32 (0.45)

Table 2.2: Metadata size uncompressed, compressed and load time into Redis (20 runs) as a function of number of inodes. The headers are the same as in Table 2.1 except the first column is a unitless count of live inodes in the file system rather than used disk space.

metadata grows at a rate of 6–7 megabytes per gigabyte of used disk space, but only 1 megabyte compressed. `NTFS` crawls are quicker because its on-disk metadata is in a single, linear stretch on disk. Load times are also comparable, but `NTFS` is slower because it starts with approximately 500 megabytes more used disk space. In addition, `NTFS` often has multiple names for the same file, further magnifying metadata. As we stated earlier in this section, we can mask these load times by concurrently loading metadata while the virtual disk transfers over the network to its host system. Crawling is a one-time operation, so the overhead in time shown in Table 2.1 is amortized over the lifetime of the virtual disk image in a cloud.

Checking out an Image

When checking out a virtual disk from a virtual disk library, the key parameter deciding overhead is the size of the metadata we need to send for DS-VMI to run close to the destination of the virtual disk. In this section, instead of exploring the relationship between used disk space and crawled metadata size, we explore the relationship between live paths and crawled metadata size. In other words, we artificially boosted the number of active inodes in our file systems trying to maximize the size of extracted metadata. In this manner, we have deliberately tried to stack the deck against DS-VMI. We again explore with both `ext4` and `NTFS`.

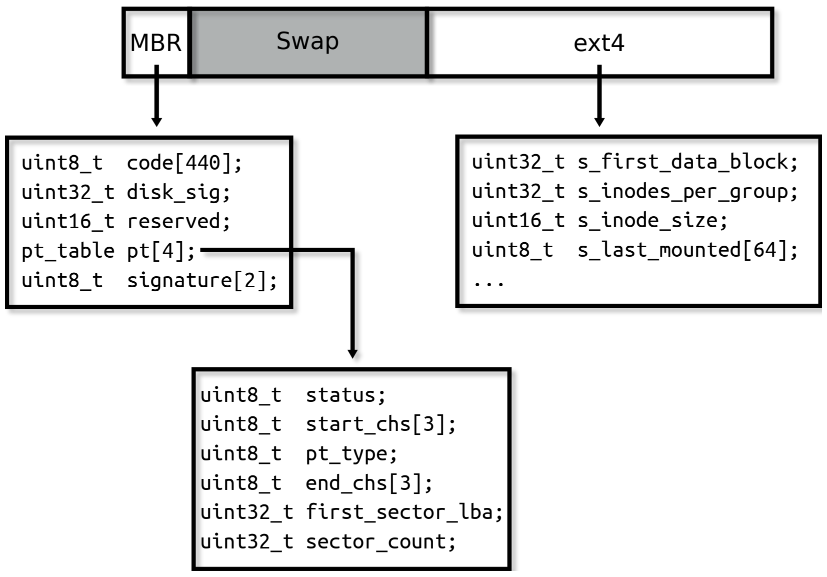
Many portions of metadata are filled with zeroes, and runs of similar numbers or duplication within pathnames are common. Therefore, the extracted metadata compresses to generally less than 10% of its uncompressed size. In a cloud datacenter with gigabit or 10-gigabit networking, transferring on the order of 10 megabytes worth of compressed metadata will take less than one second. This seems very reasonable given the optimistic 18 seconds – 3 minutes necessary to transfer a 20 gibibyte virtual disk. As a percentage of used disk space, the compressed form of our extracted metadata is always less than 1 percent.

Table 2.2 shows how metadata grows as a function of live files in `ext4` and `NTFS`. These files were created by the `touch` command within a single directory. Once again, we see a linear relationship for both file systems. The raw metadata grows at a rate of approximately 323 bytes per file for `ext4` and 285 bytes for `NTFS`. Compressed, this overhead drops to approximately 34 bytes per file for `ext4` and 13 bytes for `NTFS`. For `ext4`, the average path length was 32 characters, and for `NTFS`, it was 24 characters $(1, 250, 000$ cases$)$. This implies approximately 8 more bytes are in an `ext4` path, although with BSON serialization each string also has a type byte and a field of four bytes representing length. Thus, we were adding approximately 13 bytes more length per file in `ext4` then `NTFS`. Paths are stored at least twice: once as a full path associated with a file in an index, and once as a directory entry. Because of this duplication, we have an extra 26 bytes approximately per active inode with `ext4`. This accounts for the discrepancy in raw metadata size between `ext4` and `NTFS` as we scale up the number of active inodes.

2.3.2 Crawling a Virtual Disk

We make no assumptions about a virtual disk and currently our DS-VMI prototype supports two different forms of partition table, along with three file systems. The old-style MBR partition table was the simpler and first format our prototype supported, and GUID Partition Table (GPT) support was implemented in the fall of 2014 by a two-person graduate student team initially unfamiliar with our prototype codebase. Because of the modular design, they were able to implement support quickly and independently in their own module for parsing GPT partition tables. In this section we explain the typical layout of a Linux-based virtual drive image.

Figure 2.6 shows a typical virtual drive layout. At the start of the disk is an MBR that contains a partition table. Each entry in this table may point to a valid primary partition or to a linked list of secondary partitions. In Figure 2.6, two partitions are defined. One is for "Swap," which is not analyzed further because it represents memory pages and not an actual file system; the other is a valid `ext4` partition. The `ext4` partition is analyzed by first examining and serializing its superblock. Figure 2.6 shows that even a cursory look into the superblock of an `ext4` partition reveals a lot about the underlying file system. For example, the `s_last_mounted` field identifies the most recent mount point of this file system; this information helps in recreating pathnames.



```
uint8_t  code[440];
uint32_t disk_sig;
uint16_t reserved;
pt_table pt[4];
uint8_t  signature[2];
```

```
uint32_t s_first_data_block;
uint32_t s_inodes_per_group;
uint16_t s_inode_size;
uint8_t  s_last_mounted[64];
...
```

```
uint8_t  status;
uint8_t  start_chs[3];
uint8_t  pt_type;
uint8_t  end_chs[3];
uint32_t first_sector_lba;
uint32_t sector_count;
```

Figure 2.6: View of a raw disk split into partitions by the partition table within an MBR at the start of the disk.

2.3.3 An Example Journaling File System: `ext4`

A deeper look into the `ext4` partition is shown in Figure 2.7. `ext4` file systems are divided into block groups, which are listed in the Block Group Descriptor (BGD) Table. Each block group is associated with a group of inodes in the Inode Table and a group of data blocks on disk. Our static analysis serializes all of this information, because it could change during runtime and affect the emitted stream of file-level updates.

As shown in Figure 2.7, the Inode Table captures a wealth of information about each file. In `ext4` the "`i_block`" field is typically the header of an *extent tree*. Immediately following the header are `eh_entries`—pointers to extents—each of which in turn points to a set of data blocks for the file. The inode shown in Figure 2.7 describes a directory; its data blocks contain filenames and inode numbers for each file in the directory. By walking the inode table and directory entries, starting from the root inode, the disk analyzer collects and serializes the necessary metadata from all allocated files within the file system.

2.3.4 An Example Closed Source File System: `NTFS`

Out of the three file systems considered in this chapter, `NTFS` has the most complex on-disk layout and data structures. Implementing introspection for `NTFS` was further exacerbated because `NTFS` has no open specification, and thus the least amount of documentation out of the three file systems we support in our DS-VMI prototype. This made the initial parsing of `NTFS` difficult and excruciatingly

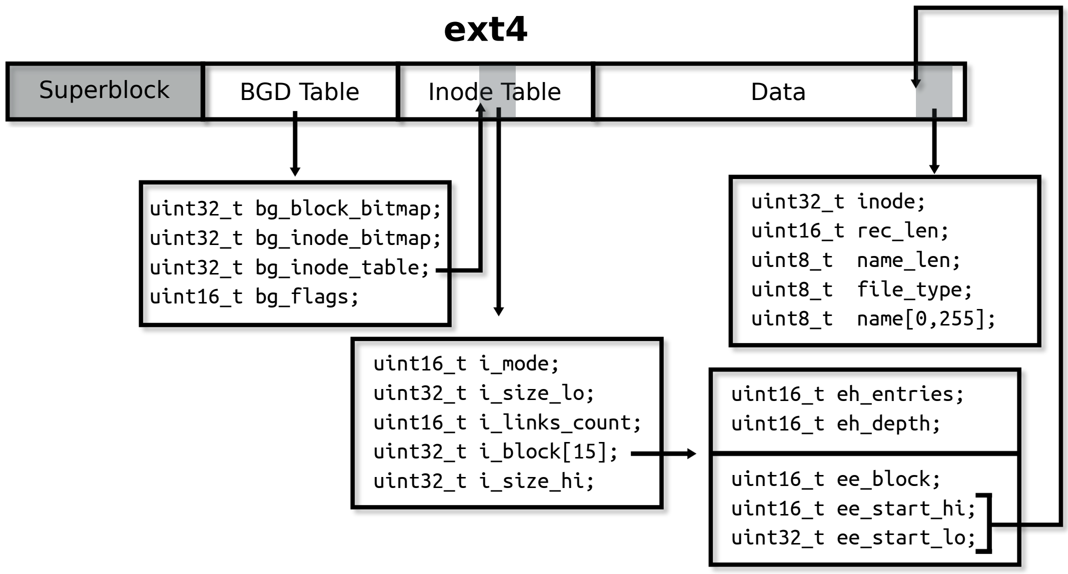


Figure 2.7: View of an `ext4` partition and critical metadata on-disk structures. The structure in the data section is a directory entry.

tedious. What little documentation that exists openly about NTFS comes from reverse engineering efforts, and none of it is complete.

Figure 2.8 shows the general layout at a high level of a NTFS partition. Similar to FAT32, NTFS keeps all critical metadata in a single location—the special Master File Table (MFT) file. Everything in NTFS is considered a file, including all portions of the disk dedicated to metadata. This is a very clean design, which allows the file system to manage even metadata as just another file entry. However, this early design decision led to an accumulation of complex metadata headers and types. For example, as shown in Figure 2.8 the metadata for a single file, called a File Record, is split amongst several different data structures. Most File Records consist of at least 3-4 metadata structures, each with their own header.

Bootstrapping an NTFS file system involves parsing values in the first sector of the partition. This portion of the disk is also a file and appears in the MFT with the special name of `$Boot`. These special files, which includes the MFT itself with the name `$MFT`, do not appear in the normal directory hierarchy visible to users. But, they are manageable by NTFS as simple files. Critical fields of the `$Boot` file are shown in Figure 2.8 and include an offset to the `$MFT`, the size of a sector in bytes, the amount of sectors making up a cluster, the number of clusters per File Record in the `$MFT`. When crawling files, many, if not all, of their attributes need parsing. Three key attribute data structures for a File Record, pertinent to DS-VMI, are shown in Figure 2.8.

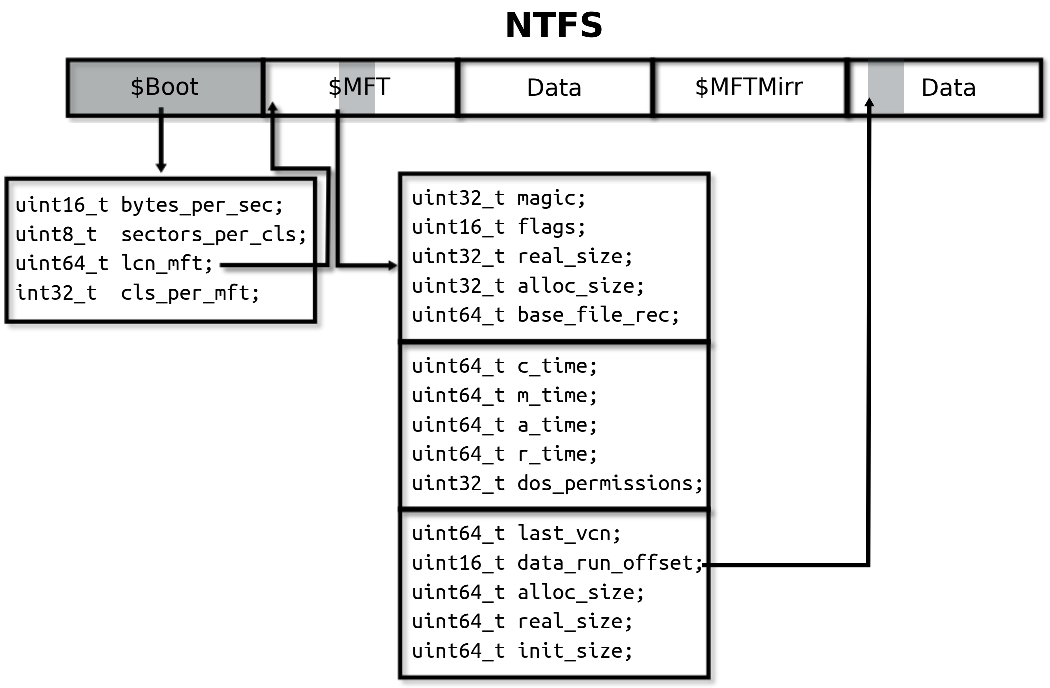
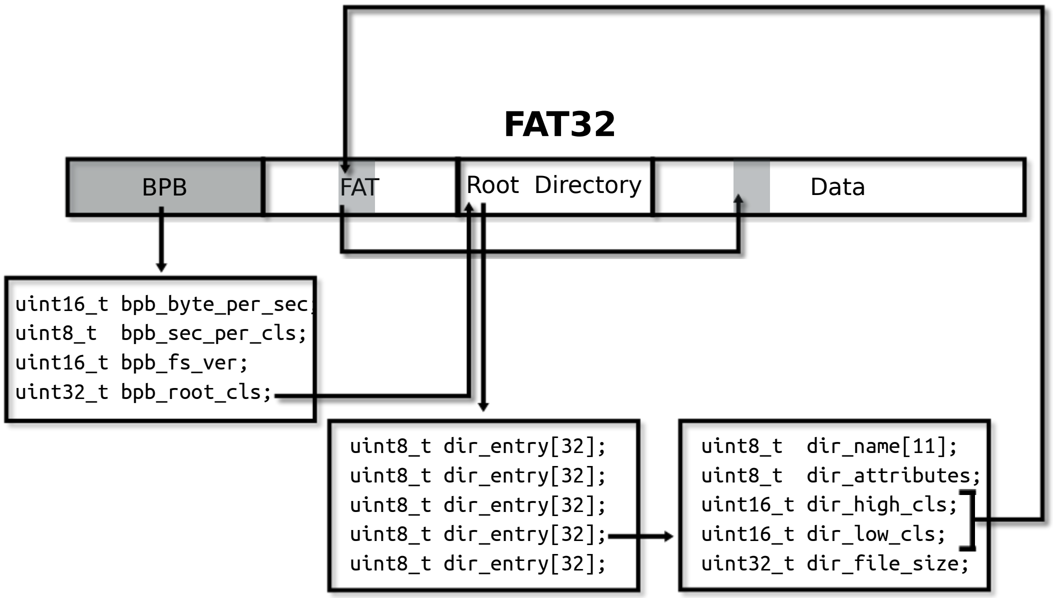


Figure 2.8: View of a NTFS partition showing the complexity of reading a single file from the Master File Table. Although all in one location, NTFS has the most complex on-disk layout out of all the considered file systems.

The first attribute contains general file information such as whether or not the referenced File Record is active, flags on the file, and the size of the file. The next structure describing a file is called the File Name and contains important information such as timestamps. The last attribute indicates a non-resident—not within the $MFT—pointer to the potions of disk making up the byte stream for the file associated with this File Record. Directories further complicate the picture, and are not shown in Figure 2.8.

NTFS was the most difficult file system to implement introspection for because its on-disk specification is not open. We worked almost entirely from reverse-engineering documentation and even with that had to guess occasionally about the data structures laid out on disk when portions of them were not described in the reverse-engineering documentation. NTFS keeps all metadata in a contiguous region of the disk called the Master File Table (MFT). This is in contrast with ext4, which splits inodes across block groups. Thus, crawling a NTFS file system is a more sequential and efficient workload, although it requires more complex knowledge. This added complexity may outweigh the costly seeks incurred by ext4 across the disk as inodes are being processed. Our NTFS implementation serves proof that implementing introspection for a complex closed-source file system without guest modifications is possible although it may be undesirable without support from



```
uint16_t bpb_byte_per_sec
uint8_t  bpb_sec_per_cls;
uint16_t bpb_fs_ver;
uint32_t bpb_root_cls;
```

```
uint8_t dir_entry[32];
uint8_t dir_entry[32];
uint8_t dir_entry[32];
uint8_t dir_entry[32];
uint8_t dir_entry[32];
```

```
uint8_t  dir_name[11];
uint8_t  dir_attributes;
uint16_t dir_high_cls;
uint16_t dir_low_cls;
uint32_t dir_file_size;
```

Figure 2.9: View of a FAT32 partition showing traversing the root directory and looking up the start of a file via a singly linked list in the FAT data structure. The FAT portion of the disk is an array of linked lists defining the clusters assigned to a file byte stream.

the vendor of the closed-source file system.

2.3.5 An Example Non-Journaling File System: FAT32

FAT32 is a closed-source, but open specification file system as of 2000 [74]. The availability of the on-disk format specification meant that implementing introspection for FAT32 was much easier than for NTFS, which is only known openly via reverse-engineered data structures. FAT32 is organized around a single large lookup table called the File Allocation Table (FAT). This giant table contains entries to files, which are lists of clusters on the disk. In FAT32 parlance, as in NTFS, a cluster is some number of disk sectors and equivalent to the concept of a block in ext4. The order of these lists is determined by lookups inside the FAT32 table.

Figure 2.9 shows a typical FAT32 partition layout. It starts with the BIOS Boot Parameters (BPB) section which contains bootstrapping variables and is similar to ext4's superblock or NTFS's $Boot file. Technically, there is an additional Reserved Region before the BPB, but that is not important for this discussion of FAT32-specific on-disk structures. The BPB provides details for indexing into the Root Directory. Directories are arrays of 32-byte directory entry structures. A directory entry pointing to a directory has the ATTR_DIRECTORY attribute set in the dir_attributes field. The map of disk clusters to the byte stream making up a directory or a normal file is stored as a singly linked list within the FAT portion of the disk.

The FAT portion is a large array of 32-bit integers. Each position corresponds to a cluster on the disk. The contents of each position, if not zero, indicate a pointer back into the array for the next cluster position in a file. Thus, these singly linked lists are null terminated indicating the end of a file or directory. For example, imagine a file starting at cluster 3. To obtain the next cluster in the byte stream, we index into the FAT at position 3 and get 7. This means the next cluster in the sequence is cluster 7, and this process continues until a 0 is encountered, which represents no more clusters for any particular file. Directories are just files with a special attribute set, as mentioned, and a structured list of directory entries as their contents.

The complicating factor in tracking information for introspection with `FAT32` is that `FAT32` has no unique data structure separating a file from its path. Thus, files are uniquely defined by their containing directory, which makes their metadata tied directly to their position in the directory hierarchy. To account for this problem and assign each file a unique numeric identifier, similar to an inode number, DS-VMI crawls the FAT file system tree in a depth-first scan. We use the implicit depth-first post-order file system tree position of a file to derive its unique identifier. This value can be calculated easily when doing an initial crawl of the file system tree. Unfortunately, as the tree mutates over time, this can cause large portions of the file system tree to change their unique identifiers. To prevent this, a hash of the absolute path could be used which is invariant to post-order positioning of a file in the file system tree. However, when new directories are introduced into the file system hierarchy, potentially large numbers of hashes also require updating. Our DS-VMI prototype currently only implements the post-order file system tree identifier. It is difficult to create a better identifier because nothing else uniquely identifies a file, neither implicitly nor explicitly.

2.4 Asynchronous Queuing of VM Writes

Remember that KVM sends emulated I/O to QEMU, which is a userspace emulator. We copy this write stream from QEMU to our DS-VMI prototype over a TCP socket using a modified version of QEMU's `drive-backup` command. As Figure 2.10 shows, our modifications are located within QEMU's core, between the layers that communicate with the guest VM and those that communicate with the backing storage. This lets our DS-VMI prototype operate independently from virtual disk formats, such as KVM's `qcow2` or VMware's `VMDK`, and I/O protocols, such as IDE or the paravirtualized VirtIO. Our prototype thus supports any virtual disk format and any disk I/O protocol supported by QEMU.

We anticipate that similar implementations are possible with other hypervisors—that is, implementations divorced from the specifics of the virtual storage setup. The format of the raw writes we obtain from QEMU is very simple. It consists of a small header structure detailing the position on disk of the write, and the number of bytes in the write. The last part of this structure is a pointer to the actual bytes. We serialize this structure again using BSON. We have our own implementation of BSON in C, which we released open source. This BSON-serialized version of the write stream

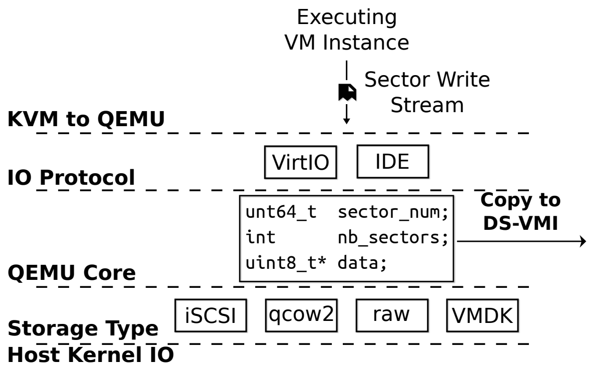


Figure 2.10: Connecting QEMU to DS-VMI.

gets converted into NBD write messages for transport to and consumption by DS-VMI. The TCP socket could route to the local host, or to another host on the network for further processing.

The other end of the NBD TCP socket is connected to the *Async Queuer*, shown much earlier in this chapter, in Figure 2.1(b), which collects the write events and copies them uninterpreted into an in-memory queue for further processing. In our case, the Async Queuer is little more than a buffer and a translator between the NBD protocol, and our in-memory queue. Our custom NBD implementation, which is also open source, does not need to implement reads as we only need writes duplicated for DS-VMI, although we implemented reads for testing purposes. Limiting the amount of duplicate operations reduces overhead. Our implementation is built upon the `libevent` library, and we believe it to be performant. In tests, our implementation can saturate gigabit Ethernet network links. The challenge is to minimize I/O stalls on the write path of the introspected VM. In order to minimize or eliminate stalls, the Async Queuer empties the socket buffer quicker than the incoming stream of writes. To accomplish this the Async Queuer processes events as quickly as possible, and uses double-buffering during flushes to further minimize stalls. Although in the worst case it is not possible to keep up, and writes wait for sometime in the TCP socket buffer.

2.5 Introspecting Live Virtual Disk Writes

The *Inference Engine*, introduced early on in this chapter in Figure 2.1(c), first retrieves the BSON-serialized metadata associated with the virtual disk being monitored, decompresses it, parses it, and stores it in a Metadata Store either all at once or lazily (lazy loading optimization described in Section 2.8.6). Remember that this metadata was obtained via crawling the disk as described in Section 2.3. The Metadata Store queues sector writes awaiting translation, and also stores metadata in translation tables for fast lookup. We use Redis [101], an efficient in-memory key-value store, as our Metadata Store. Redis also doubles as an implementation of a publish-subscribe message broker, which we use to implement `cloud-inotify` as described in Chapter 3.

Once virtual disk metadata is loaded and the Async Queuer starts copying raw write events into the Metadata Store, our DS-VMI prototype begins processing the write events by translating the received virtual disk sector writes into actual file system updates. To achieve this, each VM instance has an associated DS-VMI process on its host started alongside the VM. Or, if necessary, DS-VMI may run over the network instead of at the same host as the executing VM. Since DS-VMI runs as a set of separate Linux processes, it can benefit from multiple cores on the host.

At runtime, disk addresses are reverse-mapped using the lookup tables in the Metadata Store in order to determine which file or directory is modified by a disk sector write. Creations and deletions of files and directories are detected via inference based on metadata manipulations; this is file-system-specific and may require monitoring of a journal, inodes, or other file system data structures. Our DS-VMI prototype stores and maintains metadata in a file-system agnostic format, implemented via multiple Redis keyspaces. The benefit of a file-system agnostic format is that it enables easy implementation of generalized tools which work independent of the monitored file system type.

Given a write to an arbitrary position on disk, DS-VMI begins by first identifying if the write is data or metadata. This is done based on mappings maintained in the Metadata Store. If a write is data, DS-VMI only needs to determine which file and which bytes within that file were modified. To reverse-map a write operation to a data block of a file, DS-VMI queries the `sector` keyspace. To retrieve the file pathname, DS-VMI queries the `path` keyspace which maintains an index from file metadata to full path. If any process registers interest in a path, both metadata updates, and full data writes are passed on. In the case of metadata, the write is additionally deeply inspected by DS-VMI and appropriate Metadata Store data structures are updated to maintain correct mappings. For example, the metadata might indicate creation of a new directory, or truncation of a file.

Naïvely, disk mappings could be maintained at the level of disk sectors, the smallest unit addressable on disk. However, it is much more efficient to match the granularity of file system blocks, because file system block sizes are typically 8-16 times larger (4-8 kibibytes) than disk sectors (512 bytes). Thus, our prototype implementation of DS-VMI maintains mappings at the granularity of file system blocks. In `ext4` the block size is derived from the superblock. For `NTFS`, cluster size comes from the "Boot File"—`$Boot`. And for `FAT32`, the cluster size is derived from the BIOS Parameter Block (BPB) data structure.

2.6 Live Attachment and Detachment

Up until this section, we considered DS-VMI as an always on type of monitoring mechanism. But what about already running VMs? What if the overhead of DS-VMI proves too costly for a new workload? These types of questions led to developing the capability of DS-VMI to live attach to a running VM instance, and to also detach on command. Adding these two capabilities also makes DS-VMI fit into a cloud ecosystem easier, as we will see in Section 2.7, because it does not actually require changing the check-in, check-out process for virtual disk images if it can live attach.

We describe our approach to live attachment via live crawling in Section 2.6.1. We describe our approach to detaching in Section 2.6.2.

2.6.1 Live Crawling and Attaching

As described in Section 2.3.1, crawling a practical virtual disk image takes generally less than 60 seconds. Because this time is so short, when we attach to an executing VM we just live crawl its attached virtual disks. The key difficulty with this approach is that the file system is mutating at the same time. Hence, it is possible for our crawl to produce inconsistent metadata. For example, consider a virtual machine in the middle of deleting a file as we crawl through the file's metadata. Initially we begin with a valid file. As we crawl imagine the file system eclipsing us and mangling an important data structure we are about to read. As we continue parsing what we believed was a valid file, we eventually begin parsing invalid metadata. This can happen even with crash consistent file systems. However, as we describe below, by buffering outstanding writes during our crawl, we can catch up and fix our inconsistencies.

Once the online crawl completes, we must fix up the metadata to represent the actual consistent live state of the crawled file systems. In order to do this, we need to know every change which occurred while we crawled. By understanding the changes which occurred during the crawl, we create a consistent view caught up to the current state of the virtual disk and its file systems. When the live crawl starts we begin buffering writes that occur during the crawl. This buffer only grows until the crawl finishes. Then, we run all of the writes through the normal introspection logic for that respective file system. We let introspection catch up normally with the in-memory metadata, which results in a consistent state.

If we are not careful, this catching up phase leads to incorrect file-level events being generated in the file-level update stream. For example, imagine the creation of a new file and how it affects directory entries. Further imagine that the crawl has already recorded the existence of this new file, but the write buffer contains writes to this directory both before and after the existence of the new file. While DS-VMI catches up to these writes in the buffer it makes erroneous inferences. When it introspects a write to the directory not containing the new file, DS-VMI believes the file has been removed from the directory and emits that update. Later, when DS-VMI sees a write to the directory containing the file it believes the file has been added to that directory again. Unfortunately, both inferred file-level updates are wrong.

To counteract these inconsistent file-level update streams while live attaching, we disable emitting of the file-level update stream until we ensure a consistent view by crawling and introspecting each write in our buffer. Once fully caught up, we re-enable emitting of file-level updates. This ensures that we never emit a false update while live attaching.

Alternatively, we could crawl a snapshot of a storage device which provides safer semantics during the crawl and obviates the need to disable emitting the file-level update stream while introspecting the buffered writes. We must still buffer writes for introspection post-crawl, but will no longer need to worry about erroneous inference of file-level updates. Scanning a snapshot of a virtual disk drive is equivalent to taking a very long time to process a single write to the device. Thus, just as with normal DS-VMI, we can guarantee a consistent file-level update stream from the moment we finish the crawl and begin introspecting buffered writes.

2.6.2 Detaching Introspection

Turning off DS-VMI is a much simpler process than attaching it to a VM. In theory, DS-VMI stops immediately with the cessation of duplicating writes. Detaching is basically effortless. In our prototype implementation, detachment occurs in three simple steps. We anticipate similar steps for most other hypervisors, as there is no reason why more complication would be required. The steps to detach are:

1. Issue a cancel of the `drive-backup` block device job to QEMU
2. Stop the Async Queuer process
3. Stop the Inference Engine process

Canceling the `drive-backup` command causes an immediate end to the primary source of overhead to a monitored system: duplication of writes. Once the job is canceled, no more NBD writes come over the TCP socket. Thus, we can safely terminate the Async Queuer process without causing any exceptions within the hypervisor. Finally we teardown the Inference Engine, letting it finish introspecting the last few outstanding writes. This whole process generally takes less than 10 seconds, thus detaching is a much more rapid operation than attaching.

2.7 Integration with Existing Clouds

Although designed for integration into cloud infrastructure, none of this chapter explicitly dealt with how to structure such an integration. In this section, we deal with this problem directly by describing an API extension to OpenStack, and providing an open source reference implementation [127, 128]. The implementation is more generic than just disk-based introspection techniques, and designed to also support other forms of introspection such as memory introspection.

There are two paths towards implementing DS-VMI within cloud infrastructure. The first path we assumed for most of this chapter: crawl virtual disks on check-in, upon execution of a VM from a virtual disk load the crawled metadata and activate DS-VMI. This path brings every single executing VM and its associated virtual disks under the purview of DS-VMI. It would work well for a fully managed cloud, such as an enterprise cloud, but this path requires invasive changes to the storage of virtual disks, and the execution of instances from those disks. The second path we just described in Section 2.6. This path decouples DS-VMI's implementation from the life cycle of a VM instance, letting DS-VMI activate on-demand for cloud users and operators. This second path is the most promising path for acceptance into a cloud infrastructure, because it only requires an extension API, not invasive changes to existing APIs. For the remainder of this section, we explore taking this second path over the first one.

2.7.1 Designing an API within OpenStack

We have taken the DS-VMI prototype implementation as described in this chapter and outfitted a research cloud with DS-VMI to better understand how DS-VMI fits into real cloud work flows. We used a research OpenStack cluster running on Ubuntu 14.04 LTS server hosts using the KVM hypervisor with QEMU emulating hardware. This is a common, and in fact the default, setup for OpenStack deployments. We deploy a customized QEMU, patched to add the special `drive-backup` command to duplicate writes to our DS-VMI prototype. This modified hypervisor has been in production use across hundreds of VM launches for research purposes and course projects, and also used to host VMs with no downtime over the course of a full year.

Although the hypervisor needs modification, we required no modification of the supported guest: Ubuntu 14.04 LTS Server. The default cloud image provided by Canonical works out of the box with DS-VMI. Thus, DS-VMI delivers on its promise in a real cloud on unmodified production VM images. With zero modifications we are able to capture writes and stream them as an interpreted file-level update stream for any executing Ubuntu guest across our research cloud. In addition, to illustrate the near-real-time nature of our file-update stream, we built a WebSockets-based front-end which has the capability of showing updates in near-real-time. The updates are streamed as BSON-serialized messages from a back-end compute node, converted into JSON-serialized messages via a BSON-to-JSON proxy, and transported to a client web browser over a WebSocket. We provide a minimal JavaScript library for subscribing to updates via `cloud-inotify`.

What is the API we extended OpenStack with implementing DS-VMI? As with most clouds, our API extension is a set of REST endpoints. We have deliberately left the DS-VMI OpenStack API generic to allow for more forms of introspection other than disk-centric introspection. We integrated our implementation into OpenStack as an extension API for the `nova` compute project. The core representation of state in our implementation is the `IntrospectionEntity`. An `IntrospectionEntity` is an abstract representation of an arbitrary underlying introspection primitive. Primarily we envision both memory- and disk-based introspection being unified under this single API; however, any form of introspection falls under the `IntrospectionEntity` umbrella. Also, note that this API makes no assumptions about when an introspection begins, or when it ends. By default we live attach to executing VM images and crawl their disks in real-time. This lets our framework only affect the VMs that our users want included in DS-VMI.

As Table 2.3 shows, the API extensions follow a simple pattern provided by OpenStack. We have deliberately left the API very generic and not directly tied it to the virtual disks serving an instance. Although that is the only form of introspection supported by our implementation, we wanted an API which could subsume future introspection efforts. For example, memory introspection of live, volatile state can benefit from this same API. This API is designed around the capability of live attaching to a running VM instance. The general workflow is to activate introspection for a VM as

Type	Endpoint	Returns
GET	`/servers/<instance_id>/os-introspection`	Lists active introspections
GET	`/servers/<instance_id>/os-introspection/<introspection_id>`	Retrieve details about a specific introspection
POST	`/servers/<instance_id>/os-introspection`	Success at starting introspection or error code
DELETE	`/servers/<uuid>/os-introspection/<introspection_id>`	Teardown introspection for an instance

Table 2.3: These are the REST API calls extending OpenStack for Introspection.

identified by an OpenStack UUID, ensure that it is operating correctly, and then access one of the two interfaces described in the next two chapters. The next chapter discusses `cloud-inotify`, and we directly implemented a front-end for `cloud-inotify` within OpenStack in the form of a proxy web application. Our front-end is a proxy designed to proxy `cloud-inotify` messages via WebSockets to a client's web browser or another listening system. We chose WebSockets because this model fits directly with DS-VMI's file-level update stream. We also provide a matching JavaScript example application managing subscriptions to upstream virtual instances and virtual storage devices. Our proxy currently resides on a known TCP port, and there is no API yet for starting or finding the proxy.

2.8 Evaluating Overall Overhead

This section provides an answer to the question: what is the overhead on the write path of virtualized storage with DS-VMI monitoring its stream of writes? We find worst case behavior in line with expectation—we have a write amplification of 2, and expected worst case slowdown of 2x. We find overhead on realistic workloads to be either negligible, or at least within reason.

We begin this section by describing the experimental setup, used throughout, in Section 2.8.1. We then explore overhead via four realistic workloads in Section 2.8.3 through Section 2.8.5. We finish this section with two optimizations for DS-VMI to reduce overheads in the worst case. We explore lazily loading metadata to reduce memory pressure in Section 2.8.6. We explore dropping writes as early as possible in Section 2.8.7 to reduce costs associated with copying them out even just to another process.

2.8.1 Experimental Setup

All host nodes are identically configured throughout all of the following experiments. Each machine has a 3.00GHz Intel Core 2 Duo E8400 CPU and 4 GB RAM and runs an up-to-date version of Ubuntu 12.04 LTS AMD64 Server. We base all of our work off of a recent QEMU source tree, git commit `71ea2e01`. We use Redis server version 2.2.12 and `libhiredis` version 0.10.1. For BSON, we have our own custom implementation in C. Each host has two hard drives: a main 250 GB drive (Seagate ST3250310AS) and a secondary 1.5 TB (Seagate ST31500341A). The secondary drive was used to write and store log files, and the main drive was hosting the VM virtual disks. This setup minimizes I/O contention while collecting results from experiments. Unless otherwise stated, timing experiments were run 20 times and both their average and standard deviation are reported.

When running a VM we follow IBM's KVM best practices [52]. Both the guest VM and host OS are configured to use the 'deadline' elevator algorithm for disk I/O scheduling, the VirtIO paravirtualization solution for I/O communication, and the asynchronous I/O back end native to their host. Before running a VM, we `sync` the host and drop all file system caches. Once the guest VM is booted, we repeat the procedure inside the guest. We configure and begin executing an experiment via `ssh`. VMs are only run for a single experiment, then discarded by deleting their hard drive and replacing it with a pristine copy. When an experiment begins within a VM guest we use a simple Python script to send a single UDP packet to a host daemon process. This process records a timestamp for the UDP packet and acts as the timer for experiments within VM guests. When an experiment finishes inside a guest VM, the Python script sends a final UDP packet to the host daemon process and shuts down. By using an external clock tied to the host VM we reduce the risk of invalid timing data due to unreliable VM clocks.

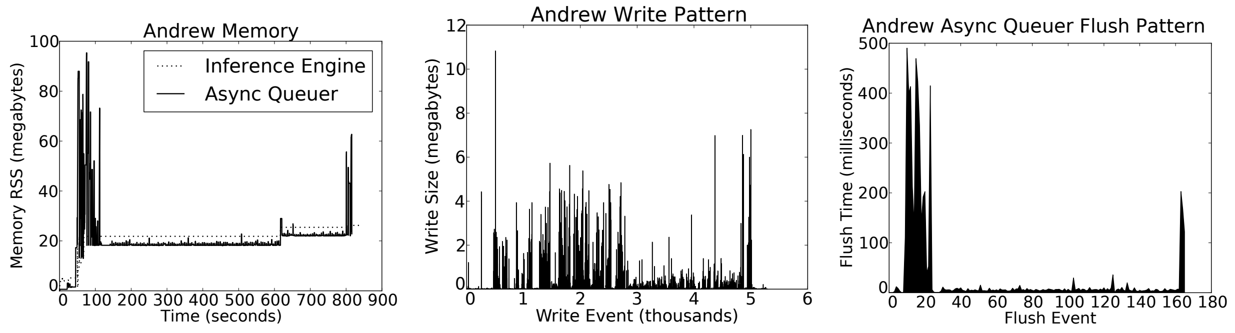
For all VM experiments we used a single VM guest pre-loaded with all software needed to perform the experiments. The guest is an Ubuntu 12.04 LTS AMD64 Server, with 1 CPU, 1 GB RAM, 20 GB disk, and a single partition containing an `ext4` file system with default file system options and 2.6 GB of used space.

2.8.2 DS-VMI Tunables

There are several tunable parameters which we fixed throughout the experiments for DS-VMI. These tunables are shown in Table 2.4. The asynchronous queuer is the first and last point of contact between an executing VM, via QEMU, and DS-VMI. If the asynchronous queuer copies write events slowly, the performance of the executing VM guest will be negatively impacted. Table 2.4 shows the main parameters which affect the behavior of the asynchronous queuer. The "Default" values were used in the experiments. In our experiments the most critical parameters are the "Async Flush Timeout" and "Async Queue Size Limit." The flush timeout helps bound the maximum latency from a disk write to an emitted inferred file-level event on a channel for subscribers. Large monitoring systems such as the one Akamai [24] deploys on $70,000+$ servers require "near real-time" updates on the order of minutes. Our default setting of 5 seconds may be too aggressive; however, this choice explores performance when flushes occur with high frequency. The queue size limit bounds the amount of memory that the asynchronous queue process may consume. The outstanding write limit bounds the number of writes queued for the inference engine—preventing it from falling too far behind the guest VM. The "Unknown Write TTL" defends against denial of service: if a guest writes a large amount of data, but never associates it with live files, the data will not be kept in a queue indefinitely waiting to be assigned a path.

Tunable	**Default**
Unknown Write TTL	300 seconds
Async Flush Timeout	5 seconds
Async Queue Size Limit	250 MB
Async Outstanding Write Limit	$16,384$
Redis Max Memory	2 GB

Table 2.4: List of DS-VMI runtime tunables which affect performance.



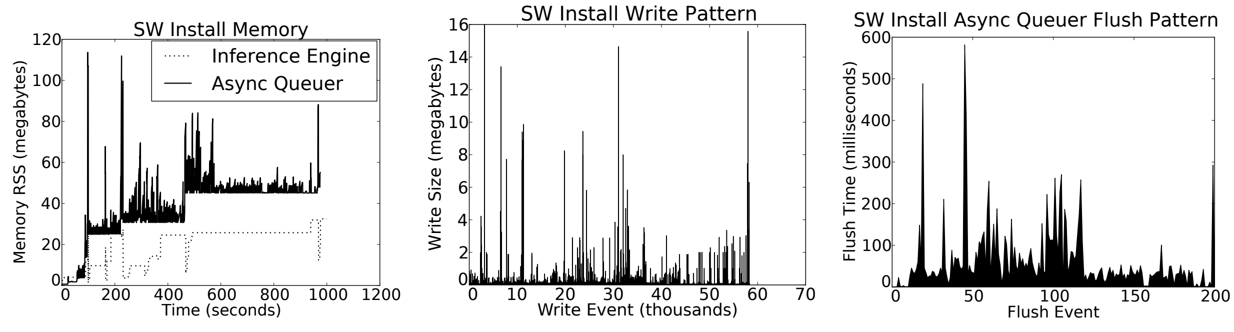
(a) Modified Andrew memory usage (b) Modified Andrew write pattern (c) Modified Andrew flush pattern

Figure 2.11: These graphs show the used memory by DS-VMI, observed write pattern of the VM guest, and the flush events triggered within DS-VMI during the Modified Andrew benchmark.

2.8.3 Light-rate Small Writes: Modified Andrew Benchmark

The Andrew Benchmark is a well known benchmark [49]. It is designed to model common operations on a developer's file system. It operates by compiling a program and performing manipulations to the source tree. We modified the Andrew Benchmark to modernize it. In the `MakeDir` phase, our modified version creates a directory tree mirroring the `linux-3.5.4` kernel tree [65]. In the `Copy` phase, it copies the entire source tree, including files, into this directory tree. The `ScanDir` phase reads file system metadata for all files in the tree. The `ReadAll` phase reads the contents of all files in the tree. Finally, the `Make` phase compiles the Linux kernel using a configuration provided by `make defconfig`.

With the default parameters, the Modified Andrew Benchmark shows negligible overhead with DS-VMI introspecting disk writes. This is because the write traffic was not sufficient to fill the asynchronous queue, and when flushed due to timeout, the write volume was small enough to avoid any performance degradation. The write pattern shown in Figure 2.11 demonstrates minimal write clustering compared to PostMark and bonnie++, two benchmarks described below. The Modified Andrew Benchmark had the fewest writes out of all the benchmarks: $5,293$ in total.



(a) Software Install memory usage (b) Software Install write pattern (c) Software Install flush pattern

Figure 2.12: These graphs show the used memory by DS-VMI, observed write pattern of the VM guest, and the flush events triggered within DS-VMI during the SW Install benchmark.

2.8.4 Clustered Large Writes: Installing Software

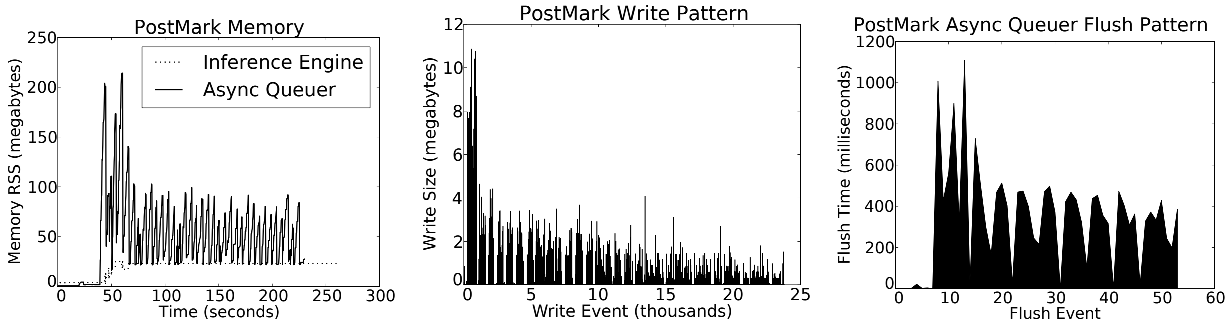
The Software Install benchmark, inspired by a benchmark used in [106], uses Ubuntu's `apt-get` tool within the guest to install a long list of server packages that have been downloaded in advance. The server packages include Apache, MySQL, PHP, Ruby on Rails, Java Application Servers, and many others.

Figure 2.12 shows the results of a run of the Software Install benchmark. The Software Install benchmark has the largest number of writes: $61,694$ in total. This benchmark, although it writes a lot of data, spreads the writes over a long period of time. There were no large bursts of heavy write activity and no wild spikes in asynchronous queuer memory. Even though it does not trigger extra asynchronous queue flushes it is, however, interrupted too frequently by timer-based flushing. Whenever the timer fires and the asynchronous queue is flushed, a few writes pending from the VM receive slightly higher latency. This effect, accumulated over the entire benchmark, was sufficient to significantly slow it down.

2.8.5 Moderate-rate Small Writes: PostMark

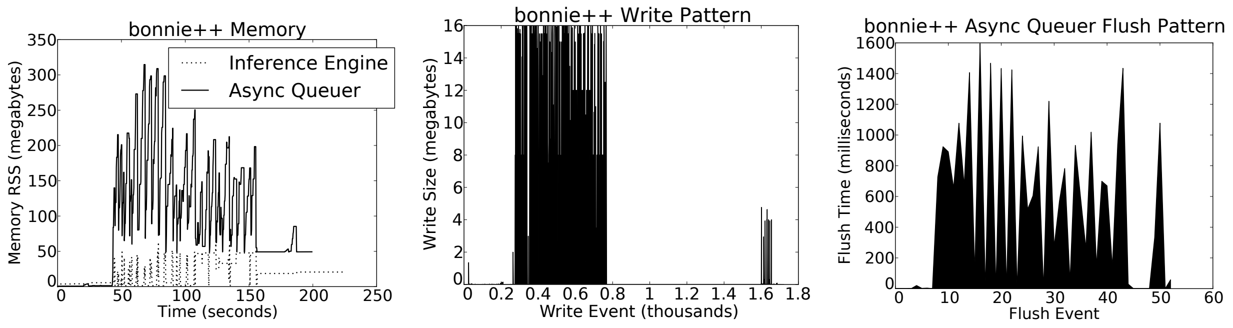
PostMark [57] is a well-known benchmark designed to simulate mail server disk I/O. We used it with a configuration suggested by [117]: file size $[512, 328072]$, read size 4096, write size 4096, number of files 5000, number of transactions $20,000$.

PostMark shows similar behavior as bonnie++; however, its write pattern is more dispersed. Figure 2.13 shows that PostMark has many smaller clusters of writes. Its asynchronous queue memory, although spiking like bonnie++, does not fill as often. These spikes do trigger extra flush events, which incur a performance penalty just as in the bonnie++ case, though on a much smaller scale. In this case the experiment ran for 231 seconds, resulting in 46 expected and 54 actual flush events.



(a) PostMark memory usage (b) PostMark write pattern (c) PostMark flush pattern

Figure 2.13: These graphs show the used memory by DS-VMI, observed write pattern of the VM guest, and the flush events triggered within DS-VMI during the PostMark benchmark.



(a) bonnie++ memory usage (b) bonnie++ write pattern (c) bonnie++ flush pattern

Figure 2.14: These graphs show the used memory by DS-VMI, observed write pattern of the VM guest, and the flush events triggered within DS-VMI during the bonnie++ benchmark.

High-rate Large Writes: bonnie++

bonnie++ [25] is a microbenchmark tool designed to measure the overhead of various file system operations such as create, delete, write, and read. We used its default settings. By default, it attempts to write a dataset at least twice the size of main memory.

A breakdown of memory usage, I/O pattern, and asynchronous queue flushes for bonnie++ is the first row in Figure 2.14. The first graph for bonnie++ shows a breakdown by memory of the various components necessary for DS-VMI. The inference engine, across all experiments, shows very little memory usage. The memory usage of the asynchronous queuer, however, repeatedly spikes up and down. This occurs because bonnie++ is a write-intensive microbenchmark, and the asynchronous queuer, in this case, is regularly hitting its configured queue size limit of 250MB and flushing to Redis. The effect is so pronounced that it slows down the VM guest while the flushing is occurring. In the experiment presented for bonnie++ in Figure 2.14, the VM guest ran for 200 seconds. If

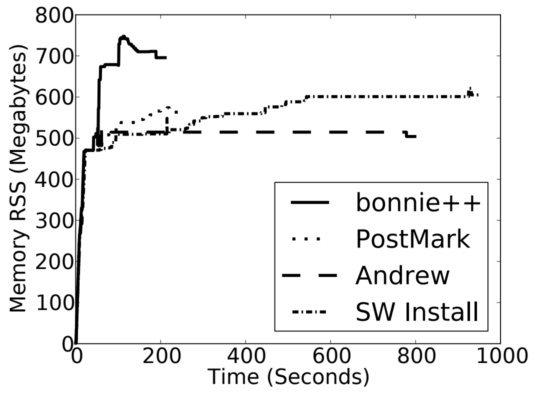


Figure 2.15: Memory usage by Redis for each experiment. Only a single run was examined.

Experiment	Asynchronous Queuer (MB)	Inferencce Engine (MB)	w/ Redis (MB)
bonnie++	250	49	1043
Andrew	88	9	630
PostMark	214	27	739
SW Install	81	26	708

Table 2.5: Peak memory usage of the Async Queuer, inference engine, and Redis combined.

flushes were only triggered by the 5-second writeback timer, this implies a maximum of 40 flushes. However, the right-most graph shows 53 flushes, 13 triggered by the 250 MB ceiling. In this and the other experiments, there were not enough write operations to trigger the outstanding-writes tunable. The middle graph confirms: this experiment was the most write-intensive of all of them—the other experiments have more dispersed write patterns. It is this closely-clustered, intense write pattern that causes the performance degradation. We simply can not hide the overhead of copying writes once buffers fill.

2.8.6 Reducing Memory Footprint: Lazily Loading Metadata

Figure 2.15 shows memory used by Redis during each of the four experiments, and Table 2.5 shows peak memory usage in Resident Set Size (RSS) by the inference engine and Async Queuer combined with Redis. At startup, the Redis database fills with metadata and the Async Queuer awaits writes from a booting VM guest. At this stable point the Async Queuer process uses 652 KB of memory, the inference engine uses 4096 KB of memory, and Redis uses 393.23 MB of memory.

This memory overhead is 15% of used disk space, when Redis is pre-populated with metadata for all files in the guest file system. This is a fairly high overhead, and becomes prohibitive with larger and larger used disk size. However, during benchmarks, as well as in expected day-to-day

Experiment	File Tree Loaded	Old Peak (MB)	Peak (MB)
bonnie++	4.7%	1043	766
Andrew	17.2%	630	357
PostMark	5.2%	739	562
SW Install	11.3%	708	533

Table 2.6: Lazy loading optimization effect on memory.

use, only a small fraction of the file system tree within the guest is modified. Thus, loading and caching metadata only for recently written files promises to greatly reduce the memory overhead of our approach.

The results of implementing lazy loading of metadata are shown in Table 2.6. With this optimization, the startup memory footprint drops to 4% of used disk space (114 MB instead of 392 MB), the loading of metadata takes 73% less time (5 seconds in the base case), and peak memory usage drops. Further optimization seems possible with customized data structures cutting out Redis, but we did not explore this path. We implement lazy loading of metadata by leaving pointers into the serialized metadata file within our in-memory datastructures. For example, we load into memory the information that a given sector is related to a certain inode. But, we don't load the inode's metadata into memory until we encounter a write to that sector.

2.8.7 Dropping Writes

Blocks in a file system can be categorized into metadata and data. A high write throughput implies that large quantities of data blocks are being written. The intuition for this optimization is that dropping data writes should significantly reduce the throughput required, while maintaining proper mappings of disk sectors to files by continuing to consume metadata writes. The only loss of information occurs when blocks transition from data to metadata such as when a data block becomes a listing of files in a directory. We either wait for follow-up writes, or read directly from the virtual disk to catch up from data blocks which may have been dropped before the transition was reflected on-disk. Figure 2.16 shows the effect of data dropping. All of the benchmarks with significant write overhead show improvement. The application-based benchmarks show that worst case overhead is reduced to 39.5% instead of the expected 100% overhead stemming from duplicating every single write. Here we trade-off completeness to improve performance. We implemented the dropping as early as possible within the hypervisor.

Our implementation of dropping writes is based off of a customizable bitmap shared between DS-VMI and the hypervisor. This lets the hypervisor drop writes early and not incur the cost of copying them to DS-VMI. In addition, it lets DS-VMI customize the writes that get passed along dynamically as metadata updates change state over time. The bitmap refers to individual disk sectors, each bit represents a sector. If the bit is a 1, the sector is passed along to DS-VMI. If the bit is a 0,

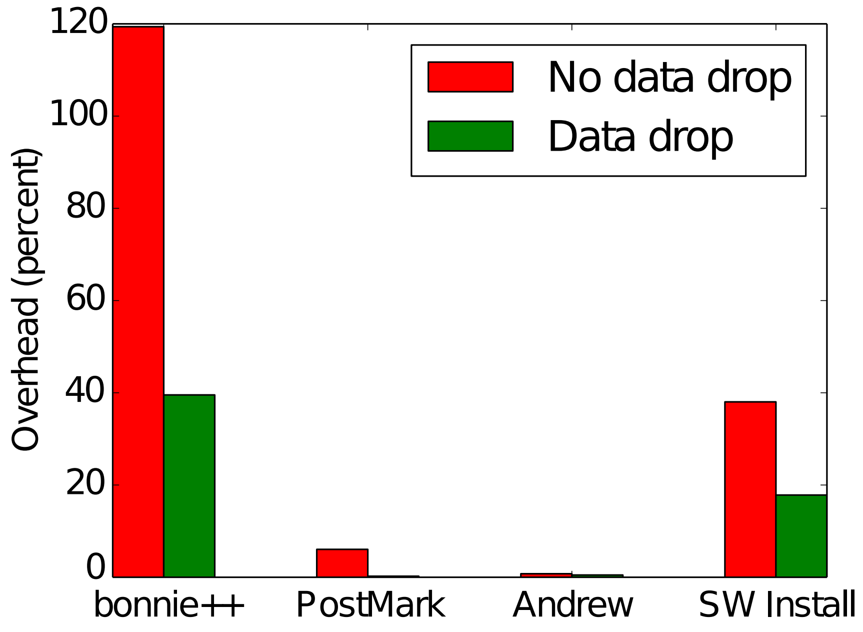


Figure 2.16: Effect of dropping data writes on DS-VMI efficiency in terms of normalized overhead.

the sector is immediately dropped by the hypervisor and not passed along for further analysis. The size of the bitmap is the overall size of a virtual disk divided by 512 bytes. For example, a 20 GiB virtual disk results in a bitmap of size 5 MiB.

2.9 Limitations of DS-VMI

This section discusses the limitations of DS-VMI as a monitoring framework. As previously described, information which resides only in the memory of an executing virtual machine guest and never flushes to persistent storage remains outside the purview of DS-VMI and its interfaces. In addition, modern protection schemes including full-disk encryption (FDE) make file-level information indecipherable to DS-VMI. We describe monitoring limitations in Section 2.9.1, and discuss the implications of modern protection schemes in Section 2.9.2.

2.9.1 Monitoring Limits

DS-VMI does not work well for all types of files and monitoring workloads. Short-lived, transient files typically exist only within the page cache of an executing virtual machine guest. Such files never flush to persistent storage. Thus, they are invisible to DS-VMI. Raw writes directly to persistent storage without using the framing of a file system are not useful to DS-VMI. Although they can be monitored, they do not fit within the structure of the file-centric interfaces layered on top of

DS-VMI. For example, a database writing directly to disk appears as a stream of unstructured writes. Sophisticated attackers hide persistent data within unused file system blocks, or unused portions of a disk. These writes do not affect any real file visible to the guest. Although they are detectable via DS-VMI, they are semantically meaningless without association to higher-level entities.

Reads are not introspected at all by DS-VMI, and certain types of writes are impossible to introspect. Cloud workloads requiring high performance storage typically receive direct access to storage devices. Their writes bypass the underlying hypervisor which makes introspecting them impossible without hardware support. Monitoring applications based on the reads performed by a guest can not use DS-VMI. For example, determining which files are loaded at boot time is not possible with DS-VMI.

The extensibility of DS-VMI mitigates some of these limitations, but some are fundamentally not solvable. Databases writing direct to storage still have an internal structure which provides semantic meaning. Extending DS-VMI to support database on-disk layouts is achievable, although it would eschew all of the architecture built upon the abstraction of a file. By using hardware-assisted introspection, DS-VMI could also introspect writes from a guest which bypass the hypervisor and go direct to persistent storage. Malicious writes to non-files must still contain internal structure usable by future DS-VMI extensions. Introspecting reads, although incurring a performance penalty, is trivial within a hypervisor. Reads and writes serviced by an in-memory page cache are never visible to underlying storage layers. Thus, memory-only structures are fundamentally invisible to DS-VMI.

2.9.2 Technologies Defeating DS-VMI

Security best practices prescribe heavy application of cryptography especially for data at rest [130]. This is a troubling trend for DS-VMI, because if the guest OS uses FDE to encrypt blocks before flushing them to persistent storage, then file system data structures become hidden. File-level encryption exposes file system data structures, but prevents deep monitoring of information within files. Such protection schemes which place no trust in the underlying cloud infrastructure render cloud-implemented services useless. However, this problem is not unique to DS-VMI.

Imagine clouds with inter-VM network bandwidth optimization achieved via packet-level deduplication. Such a technique works by recognizing data within packets that is identical. This can greatly magnify the bandwidth available between VMs in a cloud if their transmitted data has a high degree of duplication. For example, a VM broadcasting configuration state to a set of other VMs typically sends the same message many times to different hosts. By deduplicating this message across the cloud, extra bandwidth becomes available to all VMs. Should the VMs not trust their underlying cloud infrastructure and encrypt every packet, then such optimizations are impossible for the cloud to implement. This places the onus of implementing distributed optimizations firmly on the cloud user, and precludes the possibility of cloud-wide, cross-user optimizations.

The obstacle posed by protection schemes is not fundamental. It is intimately tied to the trust model applied to cloud computing. The choice to defeat such cloud-wide services and optimizations hinges on this trust model, and the tension between guest and host capabilities. If guests trust their cloud to also protect their data, then implementing FDE and other protection schemes within guest environments is purely redundant. This dissertation argues that the best layer for implementing protection and optimization lies not within the VM guests, but beneath them within the cloud infrastructure. Whether for monitoring, or for optimization, such services benefit from global knowledge, coordination, and economies of scale when implemented in the cloud infrastructure. As mentioned before, decoupling these services from guest-level misconfigurations and compromises is attractive. This is not possible without cooperating with the cloud by exposing raw state.

Chapter Three

`cloud-inotify`: Cloud-wide File Events

Monitoring for file system changes comes in two flavors: polling by scanning directories, or registering for changes via event-driven callbacks. In this chapter we focus on the latter—events—and describe an interface built on top of DS-VMI, the mechanism developed in the previous chapter. In the context of file systems, polling equates to batch-style scans through directory trees. Often, tools such as `rsync` [73] check the modified timestamp on files to determine whether or not new changes need processing. However, for tools such as Dropbox [36]—a file-level versioning and synchronization tool—rapidly scanning directory trees as they grow in size to tens of thousands of files becomes increasingly costly. Thus, the capability of registering for state changes and receiving notification when they occur is a critical capability for certain file-level monitoring workloads.

`inotify` [70] is a Linux kernel notification interface for local file system events. It provides an API to userspace applications that lets them register for events such as directory updates, file modifications, and metadata changes. Via callbacks, the userspace applications act on these events. `inotify` has been used to implement desktop search utilities, backup programs, synchronization tools, log file monitors, and many more file-level event-driven applications. Using DS-VMI, we extend the concept of `inotify` into what we call `cloud-inotify`. There are three major differences between `cloud-inotify` and `inotify`. First, in the actual interface: `cloud-inotify` is a publish-subscribe network-based messaging system, whereas `inotify` is a Linux system call API. Second, `cloud-inotify` lets users subscribe to updates across many remote file systems, but `inotify` only works for local file systems. Third, `cloud-inotify` is, as much as possible, OS agnostic, but `inotify` only works on Linux systems. Naturally, other operating systems such as Microsoft's Windows and Apple's OS X have their own versions of file-level events called FileSystemWatcher [77] and FSEvents [8] respectively. But, each interface is vendor-specific and requires specialization. `cloud-inotify` requires writing file-level event logic once, and instantly works across different OS types.

`cloud-inotify` is the first of three interfaces built on top of DS-VMI that this dissertation describes. It is designed for event-driven workloads such as log analytics, intrusion detection, or file synchronization. For example, a common use case is registering for updates to a directory and its subtree in the overall file system tree. Any updates to that directory, its subdirectories, and files contained therein gets replicated via messages over the network to the registered monitoring application. Using such technology, one could rapidly build folder-level synchronization and backup. Another use case is registering for events on critical system files such as `/etc/passwd` which contains account credentials on UNIX-based systems. Or following any changes to `C:/Windows/system32/config/SAM` which contains account credentials on Windows.

The rest of this chapter is organized as follows. Section 3.1 describes the design and implementation of the `cloud-inotify` interface. Section 3.2 goes into more detail about the types of workloads we envision using `cloud-inotify`. Section 3.3 describes potential sources of latency which could delay notification of file-level events to registered monitors. Section 3.4 provides an evaluation of the sources of latency in the `cloud-inotify` system. Latency is the primary metric by which we evaluate `cloud-inotify`'s performance. The final Section 3.5 provides an example real-world end-to-end test using a web browser and a research OpenStack [85] cloud cluster.

3.1 `cloud-inotify`'s Design and Implementation

`cloud-inotify` provides a network-accessible, publish-subscribe channel abstraction enabling selective monitoring of file-level update streams. Applications "register" for events by connecting to a network socket and subscribing to channels of interest. Via published messages, they are notified of individual events and take action. We call applications implemented using file-level events *monitors*.

Channel	**Description**
`gs9671:test:/var/log/*`	Monitor all files in subtree `/var/log` in VM instance test on host gs9671
`*:*:/var/log/*`	Monitor all files in subtree `/var/log` in all VM instances on all hosts
`gs9671:*:/var/log/auth.log`	Monitor `auth.log` on all VM instances on host gs9671
`gs9671:test:/var/log/syslog`	Monitor `syslog` on VM instance test on host gs9671

Table 3.1: Examples of filter specifications, demonstrating the use of pattern matching. The `cloud-inotify` channel implementation supports glob-style pattern matching.

`cloud-inotify` channel names are a combination of three components: the cloud-internal hostname of a compute node hosting VMs, a VM or name referencing a group of VMs, and the full path of interest in the guest file system of the targeted VM instances. Example channels are shown in Table 3.1. Any updates not associated with a specific pathname, such as superblock or MBR updates, are emitted without a path component. `cloud-inotify` allows wildcard characters when subscribing to channels. Monitors can easily subscribe to a variety of events without exposing themselves to a firehose of irrelevant notifications. This improves scalability by reducing the volume of frivolous data transmitted across a cloud and potentially externally across the WAN. When monitors subscribe from outside of a cloud, they do not specify the first component—a hostname—of the channel.

```
import bson, gray
name = 'gs9671:test:/var/log/auth.log'
channel = gray.subscribe('gray.example.com',
                         name)
for msg in channel:
    msg = bson.deserialize(msg.data)
    if msg['type'] == 'data':
        print ('New write from %d to %d' + \
              'for file %s') % (msg['start'],
                                msg['end'],
                                msg['path'])
        print 'Data: %s' % (msg['data'])
```

Figure 3.1: An example monitor in Python.

Monitors are typically application-specific, and each monitor is typically interested only in a small subset of file update activity in a VM instance. In some cases, a single monitor may receive file update streams from many VM instances and thus perform cloud-wide monitoring for a specific application. In other cases, a monitor may be dedicated to a single VM instance. Monitors interact with `cloud-inotify` via a cloud-provided WebSocket API. After DS-VMI is initiated for a VM instance, `cloud-inotify` channels become available via WebSockets. The WebSockets are provided by a front-end cloud proxy which translates subscription requests from individual WebSockets into Redis subscriptions on back-end compute hosts running VMs under the purview of DS-VMI. Access control is gated by the cloud, which also has the ability to deny subscription requests. This means cloud users can create filters allowing monitors access to subsets of file-level events originating from their VM instances. The use of a front-end proxy frees the back-end cloud to implement `cloud-inotify` using any optimizations it needs. For example, subscriptions could be filtered and aggregated at multiple levels across a cloud datacenter which improves scalability.

We call this fine-grained access control a *data firewall*. Strong guarantees are possible precisely because the cloud infrastructure acts as a mediator between cloud customers and monitor providers. Cloud customers could declaratively configure access to their data at a file-level exactly the same way they configure network firewall rules today and apply them to VM instances. Though we did not implement such enforcement and considered it out of scope for this dissertation, it is still a part of the overall vision and the hooks are in place for implementing enforcement across the DS-VMI stack. On the other side of the equation, engineering monitors becomes a simpler process. By programming to a unified publish-subscribe API, monitor vendors no longer have to implement support for several versions of operating system environments. They implement just once targeting a REST API and WebSockets channels.

We use the publish-subscribe capability of Redis to implement `cloud-inotify` channels. A monitor connects to Redis' well known TCP port and subscribes to channels using filters similar to those shown in Table 3.1. The monitor then receives BSON-serialized messages relevant to its filter specification, each containing the changed metadata fields in the case of a metadata update, or the corresponding file data in the case of a data update. BSON libraries are available in a number of programming languages; Figure 3.1 shows an example of a simple agent written in Python for monitoring `/var/log/auth.log`. The code has no OS-specific dependencies, no file-system-specific logic, and is easily re-targeted based on the channel subscription expression.

Libraries for consuming such messages exist in most modern programming languages. We selected Python to form a proxy bridge between OpenStack compute nodes in a research cloud, and monitors. Python fits well within the OpenStack ecosystem because OpenStack is a pure Python-based open source cloud software. Based on OpenStack credentials, access control filters can be placed in this front-end proxy. These filters could provide strong guarantees on the type and extent of data made available to external third-party agents. Such a capability is impossible to achieve with in-VM agents because they often require root access and it is very difficult to bound their access to information. By enabling fine-grained, strong access control guarantees, `cloud-inotify` enables new possibilities in interactions between cloud customers and file-based monitoring services.

3.2 Event-driven File System Workloads

Event-driven workloads are characterized by a tight bound on the time between an event occurring, and notification of that occurrence propagating. This near-real-time constraint makes `cloud-inotify` challenging to implement because introspection may not have a lot of time to complete, otherwise it risks introducing latency. Many types of workloads are subsumed in this category including file-based alerts, log analytics, intrusion detection, and events act as triggers to longer running batch jobs as described in Chapter 4.

In this section we demonstrate the usefulness of `cloud-inotify` and how event-driven workloads are ubiquitous. We demonstrate this via two applications for `cloud-inotify`. We currently expose `cloud-inotify` via a WebSocket proxy as an OpenStack front end. This proxy translates the BSON-serialized file-level update stream into a JSON-serialized WebSocket message stream for web browsers or other clients consumption. Using this interface we can perform fine-grained access-control via OpenStack on individual update messages, although this is currently not implemented. Two use cases which are implemented already via WebSockets are described below.

3.2.1 Continuous Compliance Monitoring

Auditing file-level changes is useful for enforcing policy, monitoring for misconfigurations, and watching for intruders. For example, many businesses monitor employee computers for properly licensed software. Using `cloud-inotify`, enterprises gain visibility into all systems across their fleet without worrying about OS-specific implementations or deployments of monitoring software.

`cloud-inotify` has a centralized design focused on aggregating the updates of file systems across a cloud. In addition to the aforementioned benefits, `cloud-inotify` cannot be turned off, tampered with, or misconfigured by guests unlike agent-based solutions. Centrally-managed auditing ensures that all VMs are checked for the most recent security updates and best practices as well. Example checks include: proper permission bits on important folders and files, and monitoring `/etc/passwd` to detect new users or modifications to existing users. Google [27] reported an outage in early 2011 that affected 15-20% of its production fleet. The root cause was a permissions change to a folder on the path to the dynamic loader. Localizing such problems within a short amount of time can be difficult, but an auditor built using `cloud-inotify` would have detected the misconfiguration almost instantaneously. Google found troubleshooting difficult because logging into affected servers was impossible. `cloud-inotify` does not depend on guest health; thus, cloud customers never have to "fly blind" even if they cannot log into VM instances.

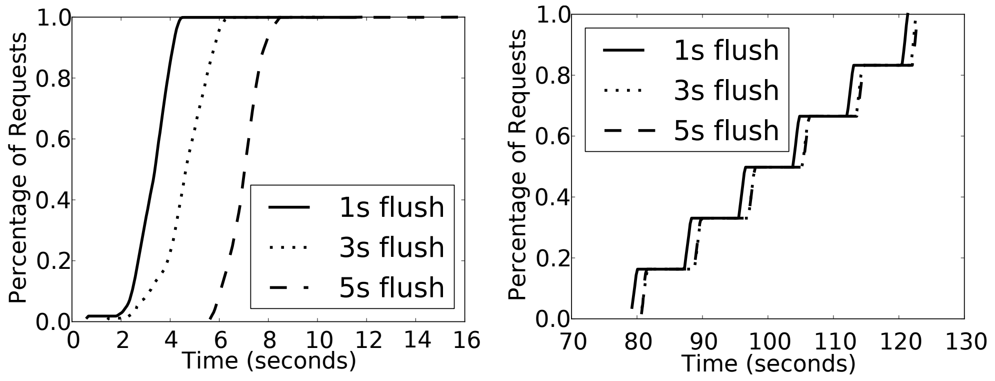
3.2.2 Real-time Log Analytics

Log files contain insights into the health of systems and the responsiveness of their applications. An example of such an insight is response time derived from web application log files. In this example we consider not just engineering monitoring solutions from the perspective of the cloud user or a monitor vendor, but also the usefulness of `cloud-inotify` for directly implementing cloud infrastructure features. One straightforward intuition is if the infrastructure has an understanding of application performance, it can more intelligently assign resources across the cloud. Sangpetch et al. [102] show that an application-performance-aware cloud can double its consolidation while lowering response time variance for customer applications. They measured response time based on network traffic; however, encrypted flows and applications not tied to network flows cannot benefit from this feedback loop. Normally, the opacity of a VM requires resorting to indirect measures such as inspecting network packets to measure response time. With `cloud-inotify`, the same metric can be derived from accurate information directly from application logs.

3.3 Sources of Latency

The sources of latency are primarily parameters inside the guest kernel and outside of our control. DS-VMI does its best, but it only receives writes when the guest kernel emits them to the virtual storage layer and not before. Thus, the fundamental limits on latency are gust kernel flush interval, DS-VMI processing time, and message propagation time. We assume the guest kernel is outside of our control, although as discussed in Chapter 2 future guests will likely cooperate with cloud infrastructure.

The guest kernel introduces latency via two primary mechanisms: the page cache, and write reordering. Page caches are used to batch updates in fast memory destined for slow persistent storage. Page caches primarily speedup small write workloads. Page caches also serve as fast access for reads, and they rely on temporal locality to maintain high hit ratios. Write reordering occurs



(a) Sync every second in the guest VM. (b) No in-guest syncing.

Figure 3.2: Latency CDFs demonstrating feasibility of near-real-time event-driven agentless monitoring using `cloud-inotify`.

when the kernel takes license to optimally order writes without impacting correctness. A famous example of write reordering is the Elevator algorithm [62] which minimizes parameters such as seek time, by taking the liberty to reorder recent writes queued for storage. An even simpler policy of just writing the next sector with shortest seek time clearly leads to reordering of writes. A kernel has the undesirable job of balancing recoverability, correctness, and performance. Though we can not blame the kernel for optimizing where it finds opportunity, tighter cooperation between a kernel and its underlying hypervisor would greatly aid introspection. However, the theme of this dissertation is to require zero guest cooperation—thus, we must live with the reordering of the guest kernel and related latencies.

3.4 Evaluation of Latency

Here we evaluate latency on real-world workloads as observed via monitoring log files. We monitored `/var/log/httpd/access.log` and found negligible delay at the time granularity we cared about—on the order of one second. That is, DS-VMI introduces negligible delay from the emitting of a storage write, to its introspected form.

Figure 3.2 shows the results of $10,000$ requests during the microbenchmark. Figure 3.2(a) shows the best case when the guest frequently syncs data to disk with a latency of 1, 3, or 5 seconds on average. Figure 3.2(b) shows an untuned guest where latency is at the mercy of guest kernel I/O algorithms which flush at much lower frequency than the latencies we tuned. The step-like nature is because many updates appear at once—many log file lines fit inside a single file system block. A representative monitoring system such as Akamai's [24] is an example that would tolerate these latencies without any tuning of guests, but low latency performance monitoring [102] may require tuning of guest flush algorithms.

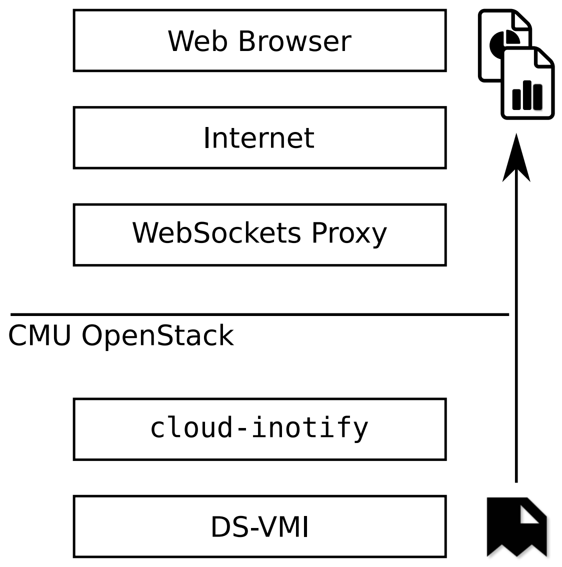


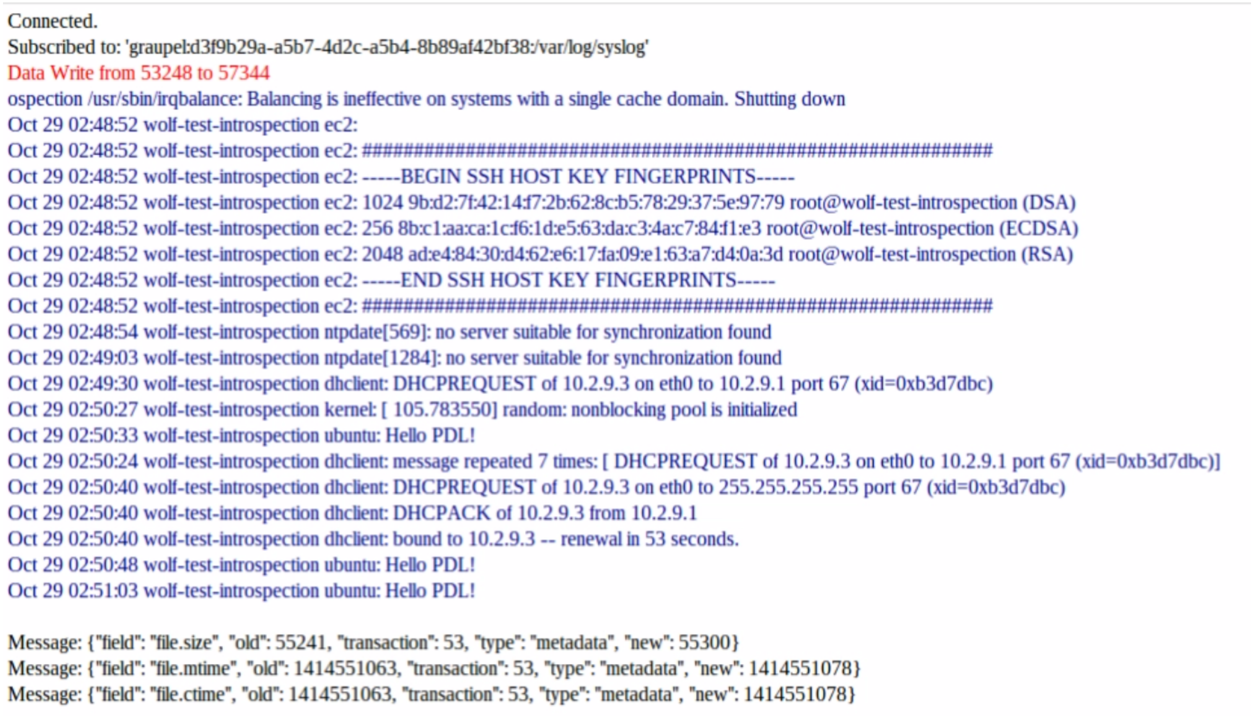
Figure 3.3: Writes are introspected by DS-VMI. DS-VMI was activated by a cloud user using a standard OpenStack command-line utility extended to support our DS-VMI cloud API. Emitted file-level updates are sent to the user via a front-end WebSockets [38] proxy over the Internet to a web browser.

This is a positive result overall for applying DS-VMI technology to implement a near-real-time system such as `cloud-inotify` on top. With zero tuning and zero guest configuration the latencies are tolerable. In addition, the DS-VMI framework shows negligible overhead in terms of latency. Especially for an untuned guest—the timeouts inside the guest kernel dominate the latency equation in this case.

3.5 Using `cloud-inotify` in a Research Cloud

As laid out in Chapter 2, we implemented DS-VMI within an OpenStack cloud computing cluster at Carnegie Mellon University (CMU). This cluster had between 15-20 healthy compute hosts over the course of the DS-VMI deployment. Each host had a modified hypervisor capable of duplicating writes to our DS-VMI framework. In this section, we demonstrate a working end-to-end application using this research cloud.

Figure 3.3, shows the flow of file-level updates as they traverse the cloud boundary. Users activate DS-VMI using a modified OpenStack command-line utility. The modified utility supports our DS-VMI API extensions residing inside the OpenStack API server. Once activated, the compute host responsible for this VM sends a command to the KVM hypervisor activating duplication of virtual disk writes. The writes are duplicated to DS-VMI which transforms them into file-level updates and emits them on channels for listening subscribers. The virtual machine guest in this



```
Connected.
Subscribed to: 'graupel:d3f9b29a-a5b7-4d2c-a5b4-8b89af42bf38:/var/log/syslog'
Data Write from 53248 to 57344
ospection /usr/sbin/irqbalance: Balancing is ineffective on systems with a single cache domain. Shutting down
Oct 29 02:48:52 wolf-test-introspection ec2:
Oct 29 02:48:52 wolf-test-introspection ec2: ################################################################
Oct 29 02:48:52 wolf-test-introspection ec2: -----BEGIN SSH HOST KEY FINGERPRINTS-----
Oct 29 02:48:52 wolf-test-introspection ec2: 1024 9b:d2:7f:42:14:f7:2b:62:8c:b5:78:29:37:5e:97:79 root@wolf-test-introspection (DSA)
Oct 29 02:48:52 wolf-test-introspection ec2: 256 8b:c1:aa:ca:1c:f6:1d:e5:63:da:c3:4a:c7:84:f1:e3 root@wolf-test-introspection (ECDSA)
Oct 29 02:48:52 wolf-test-introspection ec2: 2048 ad:e4:84:30:d4:62:e6:17:fa:09:e1:63:a7:d4:0a:3d root@wolf-test-introspection (RSA)
Oct 29 02:48:52 wolf-test-introspection ec2: -----END SSH HOST KEY FINGERPRINTS-----
Oct 29 02:48:52 wolf-test-introspection ec2: ################################################################
Oct 29 02:48:54 wolf-test-introspection ntpdate[569]: no server suitable for synchronization found
Oct 29 02:49:03 wolf-test-introspection ntpdate[1284]: no server suitable for synchronization found
Oct 29 02:49:30 wolf-test-introspection dhclient: DHCPREQUEST of 10.2.9.3 on eth0 to 10.2.9.1 port 67 (xid=0xb3d7dbc)
Oct 29 02:50:27 wolf-test-introspection kernel: [ 105.783550] random: nonblocking pool is initialized
Oct 29 02:50:33 wolf-test-introspection ubuntu: Hello PDL!
Oct 29 02:50:24 wolf-test-introspection dhclient: message repeated 7 times: [ DHCPREQUEST of 10.2.9.3 on eth0 to 10.2.9.1 port 67 (xid=0xb3d7dbc)]
Oct 29 02:50:40 wolf-test-introspection dhclient: DHCPREQUEST of 10.2.9.3 on eth0 to 255.255.255.255 port 67 (xid=0xb3d7dbc)
Oct 29 02:50:40 wolf-test-introspection dhclient: DHCPACK of 10.2.9.3 from 10.2.9.1
Oct 29 02:50:40 wolf-test-introspection dhclient: bound to 10.2.9.3 -- renewal in 53 seconds.
Oct 29 02:50:48 wolf-test-introspection ubuntu: Hello PDL!
Oct 29 02:51:03 wolf-test-introspection ubuntu: Hello PDL!

Message: {"field": "file.size", "old": 55241, "transaction": 53, "type": "metadata", "new": 55300}
Message: {"field": "file.mtime", "old": 1414551063, "transaction": 53, "type": "metadata", "new": 1414551078}
Message: {"field": "file.ctime", "old": 1414551063, "transaction": 53, "type": "metadata", "new": 1414551078}
```

Figure 3.4: Textual display of file-level updates affecting `/var/log/syslog` within an unmodified executing Ubuntu 14.04 Server. Red text denotes the affected byte range, blue text is the file contents being written, and the bottom black text is a metadata update.

case is an unmodified Ubuntu 14.04 Server 64-bit cloud computing image [17]. Our user's web browser has subscribed to some number of channels and receives updates over a WebSocket [38]. Currently, only a textual display is supported in the browser. However, by using web technologies we enable the development of GUIs utilizing the full power of the modern web.

Figure 3.4, shows the textual view presented to the user in their browser. Google Chrome was used in this demo as the web browser. In this demo, we subscribed to the `/var/log/syslog` log file inside the unmodified Ubuntu Server guest virtual machine. The red text denotes the bytes in the file being updated. The blue text represents the data being written into the file. The output respects the recorded file size, and does not print extra bytes even if the actual data written exceeds the size of the file. The black text at the bottom represents a metadata update. Note, that the file system, `ext4`, safely updates metadata *after* writing data to disk. This safe ordering of operations led to the temporal gap challenge as mentioned in Chapter 1. Also note, that metadata updates are assigned a transaction number. This is because block-level writes with granularity larger than a single file system data structure often contain many file-level metadata updates.

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

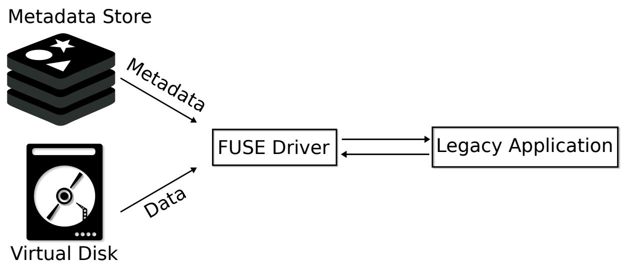


Figure 4.2: `/cloud` implementation

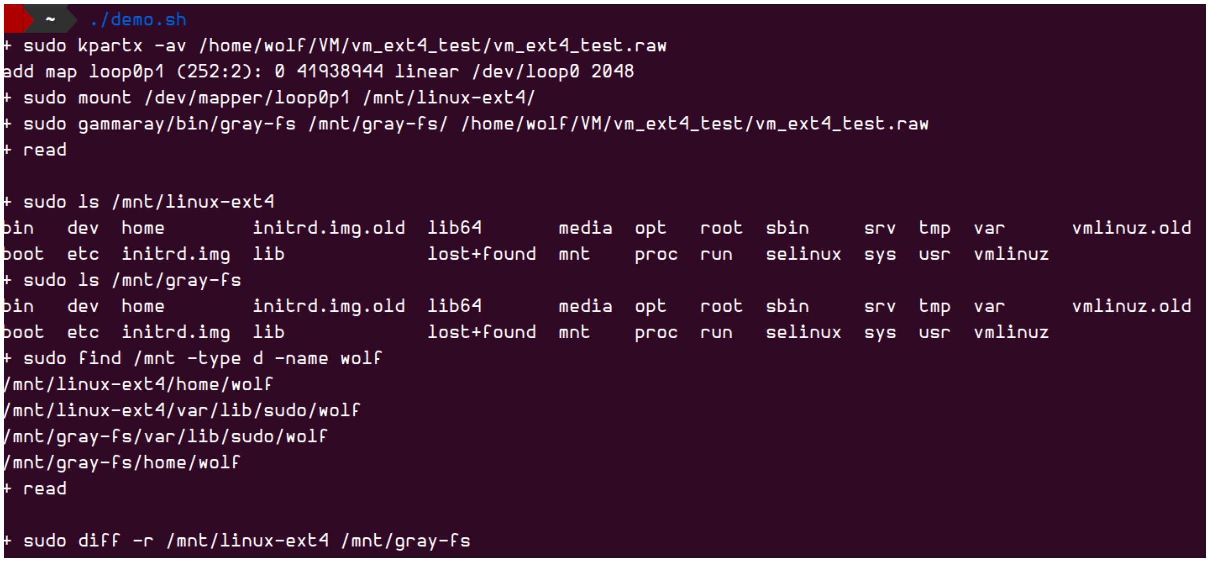
virtual disk. In other words, readers using our FUSE driver must occasionally handle stale data. For append-only style workloads, such as logging, this is rarely an issue because earlier data blocks are never overwritten unless log rotation occurs.

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.

Chapter Five

`/cloud-history`: Searchable Backup

Of course, no story about storage is complete without backup. Persistent storage maintains all long term information for individuals and businesses. Business data has a direct relationship with revenue for many businesses, and individuals are increasingly storing important memories in digital media formats. Storage is, therefore, subjected to immense amounts of protection. For example, common backup strategies include keeping three copies of data in at least two formats, with one copy off-site in accordance to the 3-2-1 rule [100]. Intriguingly, as in the beginning of this dissertation with virtualized storage, we find that the backup storage landscape is also changing. Backup storage systems no longer have the slow, high latency properties of technologies traditionally used for archiving data such as tape. Modern backup storage systems are based on faster storage technologies such as arrays of magnetic disks with access times on the order of milliseconds to seconds, instead of the slower minutes to hours for retrieving and reading tapes. Public clouds reflect this several orders of magnitude difference in access times across archival storage offerings. Amazon's Glacier [30], priced at 1 cent per gigabyte month, takes 3-5 hours to begin retrieving data. Google's Nearline [81], also priced at 1 cent per gigabyte month, takes 3 *seconds*—three orders of magnitude faster. Enterprise backup storage systems such as Sepaton [107] are also based on magnetic disk technology and have retrieval times of seconds to minutes. Three orders of magnitude faster access to archived files opens up a new opportunity for rich queries over history previously trapped within slow to access and infeasible to query backups.

`/cloud-history` is a dual exploration into architecting backup with DS-VMI and also optimizations reducing the time to index archived files. This chapter answers two questions: if we assume a cloud built with DS-VMI, *how would we architect backup differently*? And, given modern fast access backup storage systems, *how can we efficiently index backup data*? In answering the first question, we find that simply storing the file-level update streams provided by DS-VMI gives a data structure and format which exactly matches the needs of backup. The log-structure maximizes sequential throughput, and enables retrieval of previous versions by reverting or replaying events in the log. The generality of the second question makes it hard: we do not limit the type of index, nor the type of data being indexed other than assuming it resides within files. Thus, we explore application-agnostic methods of speeding up the time to index over general files. This second question, and indeed our solution, is orthogonal to the capture mechanism—DS-VMI—although we imagined solving it in the context of DS-VMI.

This chapter is organized as follows. Section 5.1 describes at a high level the benefits of using DS-VMI to capture historic state. However, the optimizations and results of this chapter are independent of the capture mechanism. Thus, the lessons learned here are applicable to a class of backup systems which feature fast time to retrieval. Section 5.2 provides a first of its kind analysis of backup data with a look at whole-file deduplication not for saving space, but for saving computation time. Section 5.5.2 describes our technique of creating versions of files without system call feedback. We resort to a heuristic-based approach, as we can not decide when a userspace process "saves" a file. Section 5.3 describes the amount of file-level deduplication we expect, how we implement it within the DS-VMI framework, and its effect on indexing workloads. Section 5.5 deals with storing file logs and versions efficiently on disk for maintaining a high ingest rate, and a quick retrieval time. Section 5.6 describes how we use another FUSE driver to retrieve historic versions stored on-disk within file-level log streams. Section 5.7 describes securing the search mechanism, and how secure search is implemented.

5.1 Transforming Live State into History

In this section we provide an overview of the properties we want in transforming the cloud's virtual disk state into a format suitable for archiving and indexing. These properties guide two key architectural decisions: how do we capture historic state, and how do we store that state for future indexing?

5.1.1 Desired Properties

We focus in this chapter on three important properties:

1. Capture complete, tamper-free history

2. Scale across different OS's and applications

3. Support semantically meaningful, efficient indexing

Preserving integrity means tamper-proof capture of the complete history of virtual storage. It implies independence from guest faults and compromises. Thus, any solution fulfilling this property needs isolation from the guest environment. In addition, the desired solution must generalize across different guest environments without requiring guest cooperation. Finally, the desired solution must provide efficient access to a semantically meaningful version of history. Otherwise, indexing becomes inefficient, and in the worst case intractable.

Given our desired properties, we explore applying DS-VMI as the state capture mechanism most well-suited to our problem. DS-VMI provides isolation by leveraging the strong isolation between a VM and its hypervisor. DS-VMI provides generality by capturing state at the level of virtual disk writes without any guest support. Yet, DS-VMI is not limited to the coarse-granularity of the low-level writes it captures. By intelligently tracking file system metadata, DS-VMI maps each write into its semantic, file-level interpretation. This makes DS-VMI ideally suited to capture historic state for `/cloud-history`.

Of course, either snapshotting or an in-guest agent, such as a versioning file system, could be used as a state capturing mechanism for `/cloud-history`. However, snapshotting requires additional processing to map each snapshot into its file-level interpretation, and does not capture a complete version of history. Versioning file systems, or any form of guest support, force constraints onto the guest environment and are vulnerable to faults affecting the guest. We ruled out both snapshotting and versioning file systems for these reasons, but recognize that with minor changes to our architecture below, they could replace DS-VMI as a capture mechanism.

5.2 Learning from History: a Backup Case Study

Is the third property discussed in Section 5.1.1 feasible in real world backup systems? Specifically, is expecting scalability of file-level indexing a valid assumption? Is there any way of increasing scalability, should it initially prove intractable, while maintaining the generality of indexing? And are backup systems capable of quick enough retrieval of data to make indexing and searching them a reasonable expectation? As we discussed in the introduction to this chapter, Google Nearline is an example cloud-based storage system designed for backup, with high bandwidth and low-latency data access which undeniably makes the answer to this question yes. Today, Google Nearline promises time-to-first byte between 2-5 seconds for a bucket, which is typically a set of files. The rest of this section deals with a study done over a large corpus of backup data in a quest to answer the first two questions.

5.2.1 Description of Dataset

In order to answer these questions and many more, we studied a large corpus of backup data from a production backup system. The backups consist of approximately 1 year's worth of backups, over 58 unique systems. Most these 58 systems are servers running a variant of Linux. The backup system, deltaic [13], automatically ages the backups. Thus, the first few crawled backups are monthly, then weekly, and finally daily. Not all of the 58 systems were included in backups for the entire time period. This is why the total number of system snapshots is less than the number of systems multiplied by the number of backups. In addition, some systems occasionally fail to backup further reducing the overall number of system snapshots. Our backups and system snapshots showed significant variance in the number of files per snapshot, and average growth in files per backup.

Statistic	Value
Number of Systems:	58
Number of System Snapshots:	3268
Number of Discrete Backups:	69
Time Period Studied:	1 year
Total Files:	1.676 billion (14 million deduplicated)
Total Bytes:	146 TiB (4 TiB deduplicated)
Average Number of Files per Snapshot:	512,898 (2,496,899)
Average File Size:	98 KiB (9 MiB)
Average Backup Growth in Files:	-10,826 (245,903)
Average Growth in Bytes:	681 MiB (24 GiB)

Table 5.1: This table shows the details of our study of a research backup system used in production support of a research group at CMU. Parenthetical values are standard deviations, unless otherwise noted.

We speculate that this variance comes from a single system with a very large amount of small files, which was eventually taken offline, thereby removing it from backups.

We anonymized the data by only storing HMAC's of the pathnames in our database. We do not maintain the private key used by our HMAC function. We are compatible with `rsync`'s notion of similarity, which was the primary tool used for capturing file-level state in deltaic. This means that if we have already recorded a certain HMAC and its modification time has not been modified, we did not count it as a new unique file. Preliminary statistics and a summary of our collected database is shown in Table 5.1. Most of the numbers agree with prior backup studies [71]. However, we observed negative average growth in the total number of files snapshotted due to an outlier backup which dropped over 1.7 million files at once. Excluding this outlier gives an average backup growth in files of 14,594 (standard deviation of 125,602).

Table 5.1 shows a two orders of magnitude drop in the total number of files vs the number of deduplicated files. As we will see in the next section, eliminating file-level duplicates has an immense impact on the time to index for real-world indexing applications. In addition, we note a similar two orders of magnitude reduction in the number of bytes. Although research on backups shows an even greater space savings with variable-sized chunk deduplication, we find that the metric with highest impact on time to index is in the number of files—not the number of bytes. This is directly reflected in our experiments.

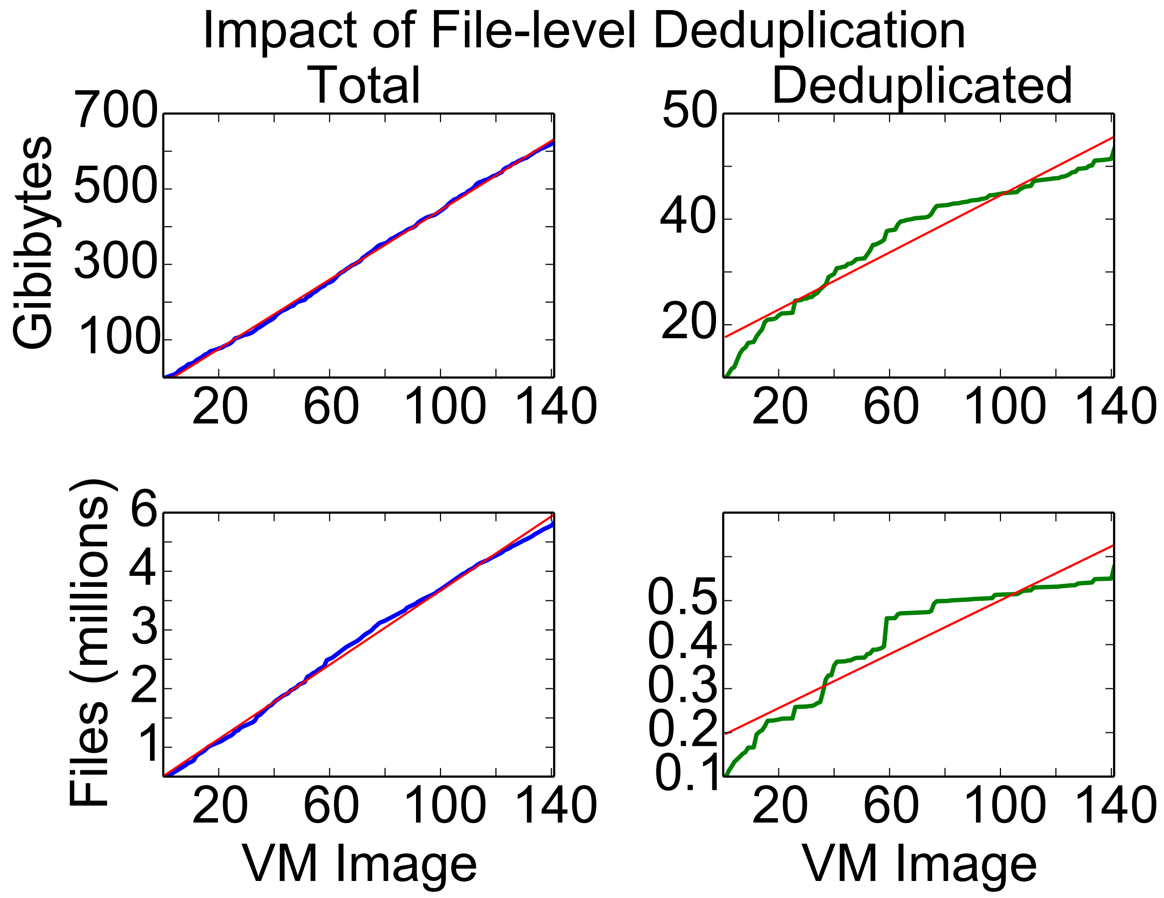


Figure 5.1: Deduplicating at a file-level leads to a 9.1x reduction in the number of files to index, and a 7.2x reduction in storage requirements (NC State, VCL Windows-based images). The raw bytes and files appear to grow linearly. The additional unique bytes and files appear to grow sub-linearly, and possibly logarithmically. Linear fits are shown as the red lines.

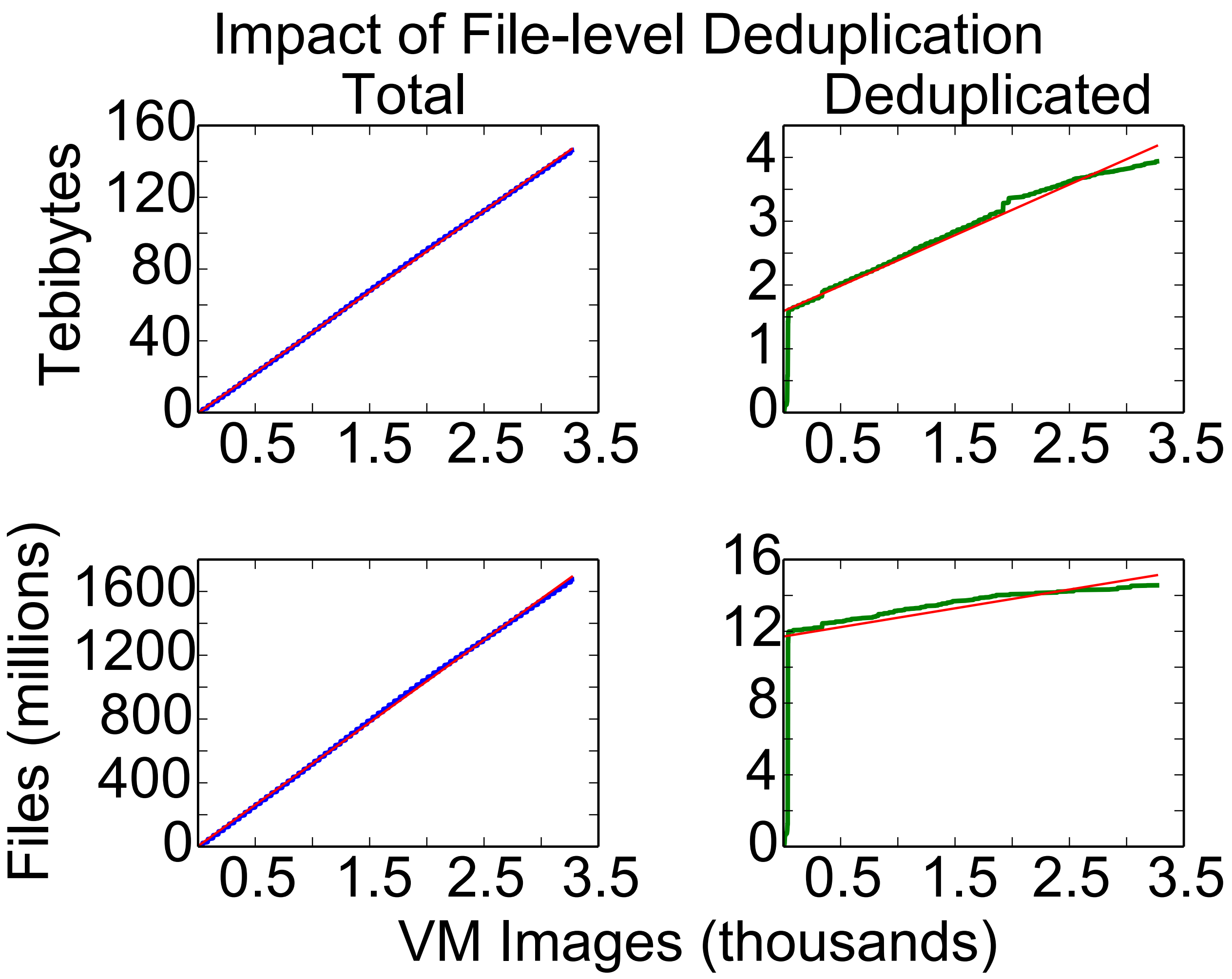


Figure 5.2: Deduplicating at a file-level is even more effective when applied to backups of systems. The unique file curve flat lines as more and more snapshots were taken of file systems by Deltaic. Linear fits are shown as the red lines.

5.2.2 Effect of Duplication

Although we had some early hints that file-level deduplication would greatly assist an indexing workload, we had no true validation that file-level deduplication works at scale until this backup study. Initially, we took a dataset of close to 140 virtual machine images from a cloud at NC State and investigated deduplicating them at a file-level. The results are shown in Figure 5.1. We observed a very promising 9.1x single order of magnitude drop in the number of unique files needing indexing, and 7.2x drop in the amount of space needed to store these images. These VM images had significant duplication between them because they were all based on Windows OS with only application-level customizations.

Although proving the point that significant deduplication exists across many VM images, we really want to know if it will be tractable to search at a file-level the snapshots of virtual storage. Figure 5.2, shows the results from an almost one-year study of the Deltaic backup system. We cut down the intractable 1.7 billion files stored in snapshotted file systems, to the more tractable 14 million unique files after applying file-level deduplication. In addition, there are significant storage savings to be had from close to 160 TiB stored down to less than 4 TiB stored. Both reductions represent two orders of magnitude reduction. As will be shown in later sections, the reduction in the number of objects to index drastically reduces the time it takes to re-index for various indexing workloads. Indexing workloads appear CPU-bound, not disk-bound. Thus, traditional deduplication at a block-level had very little impact on the overall time to index. This is an important point because it demonstrates the immense value of file-level deduplication for indexing workloads. Classic deduplication would need a second index over files not just blocks to speedup file-level indexing workloads. At the same time, classic deduplication offers the best savings in terms of raw bytes used. In reality, there is no reason to choose one over the other because file-level deduplication easily layers over block-level deduplication technology operating at a lower abstraction level. In fact, using them in tandem seems like the best idea because we can pair the bytes saved with quick file-level indexing.

5.2.3 Analysis of Trends

Putting this all together, we see that over time the number of unique files grows with a very low slope. It may asymptotically approach a constant value given enough data and workloads with little file-level churn. If we ignore the ramp up part of the curve with the initial set of 40 snapshotted systems, our slope becomes negligible and we are essentially asymptotically approaching 14 million unique cloud-wide files. The amount of bytes added by this smaller set of files accounts for 3/4 of the unique bytes in the systems studied. This is an interesting trend, because it implies that the new, non-unique files are fairly large. Potentially they are large multimedia files such as video files, although we do not know due to the HMAC scrambling of path names. The trend without deduplication is precisely as expected—the number of files and bytes grows linear in the number of systems snapshotted. This is expected because backup workloads repetitively add the same data

Version	Size (MB)	Files Same	Bytes Same	Release Date	Days Stale
2.10.1	60	100%	100%	12/14/09	0
2.10.0	59	88%	60%	10/26/09	49
2.9.0	57	57%	30%	04/17/09	241
2.8.0	53	40%	25%	10/20/08	420
2.7.0	53	35%	22%	04/22/08	601

Table 5.2: R Source Tree Similarity

Version	Size (MB)	Files Same	Bytes Same	Release Date	Days Stale
1.4.1	47	100%	100%	01/15/10	0
1.4.0	47	99%	78%	12/08/09	38
1.3.4	48	98%	73%	12/01/09	45
1.3.3	47	88%	43%	06/14/09	215
1.3.2	46	82%	37%	04/22/09	268
1.2.9	33	20%	5%	02/17/09	332

Table 5.3: OpenMPI source tree similarity.

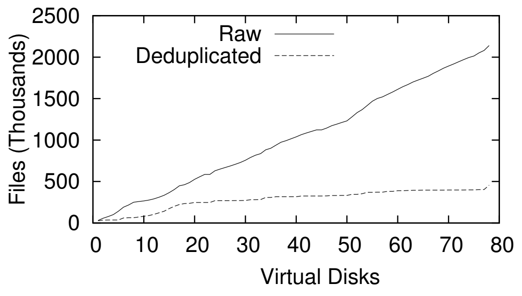
over and over again with each backup. Unless there is significant churn in the number of systems backed up, we expect generally linear growth.

Of pivotal importance to this chapter and the viability of indexing backup data is the trend of unique files. After the first backup, we see the vindicating trend that not many new files are encountered. There is a clear diminishing returns effect, which is exactly what we want for efficient indexing. After an initial index is constructed, smaller incremental updates are needed to keep it up to date. In addition creating new indexes is cheaper with such a large reduction in the file space. This discovery, along with the industry trend of ever cheaper storage with quick access times, makes a searchable backup system practical. As we will show, even a single node could index this cut down unique file space.

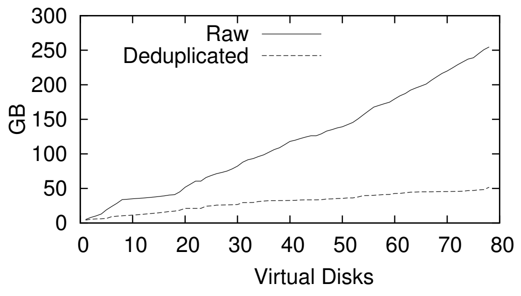
There is a step nature to all four graphs in Figure 5.2 which is due to a special system designated as an encrypted VM disk storage server which stores millions of small chunks for virtual disks in the form of small files. Such a system is not necessarily representative of cloud workloads, but it is a real example backup user.

Version	Size (MB)	Files Same	Bytes Same	Release Date	Days Stale
2.6.33.1	353	100%	100%	03/15/2010	0
2.6.33.0	353	99%	98%	02/24/2010	19
2.6.32.9	341	71%	49%	02/23/2010	20
2.6.31.12	326	56%	33%	01/18/2010	56
2.6.30.10	315	47%	27%	01/06/2010	68

Table 5.4: Linux Kernel Source Tree Similarity



(a) Size of virtual disk library in files.



(b) Size of those files in bytes.

Figure 5.3: Effect of file-level deduplication on virtual disks in a virtual disk library.

5.3 Sources of Whole-file Duplication

In this section we develop the intuition for why large amounts of duplicate files reside inside and across monitored machines. Based on this developed intuition, we posit that rapid indexing of backup data is feasible by orders of magnitude reduction in the number of unique files to index. It is this intuition coupled with experimental indexing results that guide the addition of a unique file index to `/cloud-history`.

As an example of duplication that can occur within a single system, we studied three open source projects over large time scales. We expect source code to have a relatively high rate of file-level changes in comparison to installed system files, applications, or media files. On a day to day basis, developers touch many different files. But most system files, applications, and media files are written once and read many times. They only change during installation, upgrade, or removal.

Figure 5.2 shows the amount of duplicate files within the R open source project. The R project is a widely used statistical computing package. Yet, over a 2.5 year time period, over one-third of all files in the project remained identical. The files accounted for one-fifth of all on-disk bytes for the R source tree. This suggests that the frequency of change at the file-level is not high in general, even over long periods of time.

Figure 5.3 shows the number of files that are identical in different releases of OpenMPI. OpenMPI is a message passing library frequently used in the context of supercomputers and high performance computing. Even versions that are more than six months apart show substantial similarity: between versions 1.3.3 and 1.4.1, nearly 88% of the files are identical. These files account for nearly 43% of total source tree size in bytes. Other open source projects show similar results.

Figure 5.4 shows the amount of duplication within various Linux kernel source trees. The Linux kernel is one of the most active open source projects in the world, and is comprised of millions of lines of C code. Out of the three open source projects studied, the Linux kernel had the highest amount of file-level churn. However, over three months of releases almost half of the files remained identical, which accounted for 27% of the bytes.

We also studied real-world Windows XP file systems with a dataset from NCSU's VCL cloud. Figure 5.3 shows the impact of deduplication across systems at the file-level on 78 virtual disks from the VCL cloud. The number of files with distinct content grows much more slowly than the total number of files. Figure 5.3(a) shows less than 500 thousand distinct files out of two million total files in 78 virtual disks—almost all from a production cloud at NCSU. The storage capacity and I/O bandwidth savings from deduplication of these files is substantial: 50 GB rather than 250 GB, as shown in Figure 5.3(b). Although reproduced here for quick reference, a more complete picture continuing the trend lines for the NCSU cloud dataset is in Figure 5.1.

5.3.1 Impact of Whole-file Deduplication on Indexing Workloads

We studied four representative indexing workloads to understand the impact of whole-file level deduplication on their performance. The results are shown in Figure 5.4. `tar` [39] is a baseline program which has very little CPU usage, although it has to crawl all files as if it were an indexer. ClamAV [21] is an open source virus scanner which searches for viruses throughout all files. Recoll [94] is a full-text search tool for Linux and UNIX desktops. Apache Solr [116] is a full-text document search tool designed for high performance. The applications studied show an average speedup of 5x over this dataset. Cutting down the time to index is critical for making a flexible search system that is low-latency, efficient, and usable. Today, waiting for the retrieval of backup data may take hours. With `/cloud-history`, waiting for the answer to a new query might take hours. However, once indexed, future queries benefit from cached indexes with low-latency response times.

We assume user queries are deterministic: given the same object, they return the same result. A trivial optimization for minimizing user query time is result caching. Result caching helps queries by caching the results of previous queries. While it minimizes wasted CPU cycles, result caching only helps future queries. However, we can leverage the deterministic nature of queries to also optimize individual queries.

Many modern file systems compute hashes over the data in files to ensure data integrity [33, 99, 133]. Such hashes offer a backup system that indexes file systems a free opportunity to leverage pre-computed hashes for duplicate file detection. Even if a file system does not compute a hash

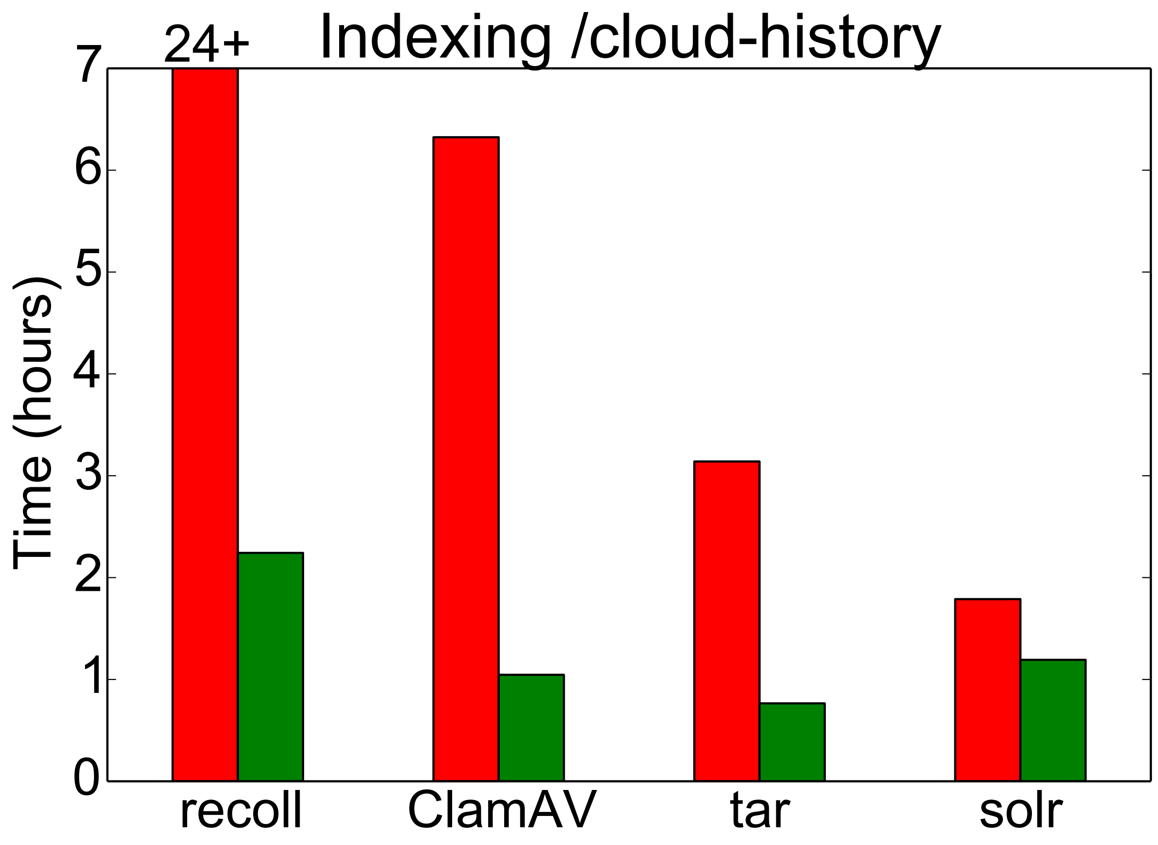


Figure 5.4: File-level deduplication not only saves space, it also saves immense amounts of computation. On average, the three workloads shown above experienced a 5x speedup by using just an application-agnostic unique file index. These experiments were carried out on a single node, on the NC State VCL Cloud image dataset.

over file data, hashing represents a fixed cost at the ingest of a backup system. There are many hashing algorithms and one with suitably high performance may be chosen to minimize impact on the ingest rate of a backup system. Thus, whether opportunistically obtained from a file system or computed at ingest, we can safely assume whole-file hashes are cheap.

Even single one-off queries, with no repeat or reuse of results in the future, benefit from duplicate tracking. They benefit by skipping duplicates resting on potentially slow backup medium, and eliminating the wasted cost of the query computation on duplicate objects. Of course, we implicitly assumed that backups are highly redundant. As shown in previous sections, this is a safe assumption because often backed up systems contain similar operating systems, libraries, and userspace applications which is what we found in the last section. For example, UNIX-like operating systems account for over 67% of all servers worldwide [125]. Microsoft Windows variants account for over 90% of all desktop and laptop computers [125]. Linux runs on 97% of all supercomputers [125]. Leveraging this similarity is an opportunity to optimize user query execution time.

5.4 Architecture of /cloud-history

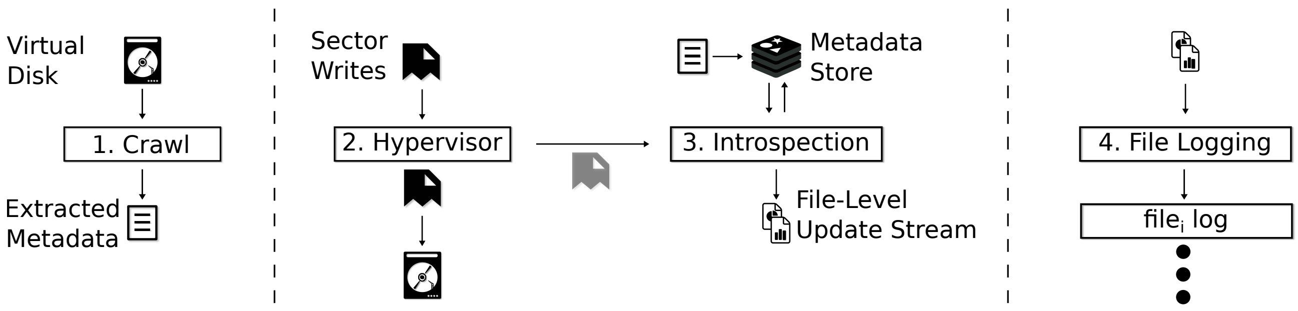


Figure 5.5: An overview of /cloud-history's architecture capturing writes via the hypervisor and translating them into a file-level update stream. The final stage stores the file-level update stream into per-file logs for indexing, garbage collecting, and application of retention policies.

Guided by the properties in Section 5.1.1, we chose DS-VMI as the technology to capture and map back into a semantic space virtual disk writes. In Figure 5.5, we review the stages of DS-VMI for virtual disks, and explain how /cloud-history stores logs of the output from DS-VMI which form the indexable history. We discuss the details of efficiently storing and indexing such logs in Section 5.5.4.

The first three stages remain the same, as they were described in Chapter 2. First, as shown in Figure 5.5, we scan the virtual disk (discussed in Section 2.3). This step extracts critical pieces of file-system metadata used at runtime to map virtual disk writes into their semantic, file-level meaning. This step can be performed either offline or online. Second, the hypervisor or network expose writes to the DS-VMI system (discussed in Section 2.4). This duplication may be turned on or off at will. The third step takes an arbitrary number of steps to resolve the semantic meaning of a write at the file level (discussed in Section 2.5). In the worst case, writes are not associated with any file system visible entity.

Once converted to a file-level update stream, traditional VMI then inspects the operation and emits the update to interested monitoring applications. In `/cloud-history`, we instead capture the stream of file-level updates into a centralized, persistent cloud store. We separate out the single large stream of file-level updates into per-file logs. Each log represents the state changes that occurred since the VM booted or was last snapshotted.

Finally, not shown in Figure 5.5, these streams are indexed to best serve the queries that users find important in `/cloud-history`. For example, a virus scanning indexer looks for traces of infections at any point of a file's life. This type of indexer inspects every update to every file. A document indexer, letting users quickly search through their documents such as Word or PowerPoint files, is only interested in files with certain extensions and their versions. A third type of indexer, a vulnerability scanner, is only interested in files containing binary, executable content. It indexes certain paths containing well-known binaries, and files with specific extensions. A vulnerability index, if kept up-to-date, is useful in answering questions such as, "which of our servers were vulnerable to Heartbleed [23]? and for how long were they vulnerable?" Such questions are important to answer because Heartbleed is exploitable without leaving a trace behind, and may lead to compromised customer data. Warning the right customers with the right dates requires answering such questions.

5.4.1 Consistently and Efficiently Naming Files without Coordination

Thus far, in this chapter, we have developed an intuition that lots of duplicate files exist both within and between systems, we experimentally confirmed this intuition, and we established the impact of whole-file deduplication on four representative indexing workloads. In short, whole-file deduplication is necessary for the scalability of indexing workloads running on top of `/cloud-history`. If it is such an important capability, how will we identify duplicate files across systems? At first glance, this seems like the perfect job for a hash function. However, modern cryptographic hash functions require re-reading all file data in order to update a hash unless they are appending writes. For potentially very large files, re-reading all of their bytes for every update is incredibly inefficient. Such a strategy would lead to the backup system slowing to a crawl as it tried to re-read large amounts of data.

Table 5.6, shows four competing methods for computing whole-file hashes. Hashing, H in the table, requires re-reading all bytes and results in the worst case $\mathrm{O}(N)$ run time. In addition, traditional hashing algorithms can not benefit from multiple processors. Merkle Trees, MT in the table, only require updating from a leaf to the root in a tree of hashes, and require a better, but sub-optimal $\mathrm{O}(\log_f N + 1)$. The special case single-level Merkle Tree, SLMT in the table, requires again a worst case of $\mathrm{O}(N)$, but without needing to re-read all bytes. Incremental hashing [12] takes an approach which provides $\mathrm{O}(1)$ updating of the whole-file hash for both random and sequential writes, and it can benefit from multiple processors. This is attractive as it is the most efficient known

Operation	H	MT	SLMT	IH
Update (S)	$\mathbf{O}\left(\mathbf{1}\right)$	$\mathrm{O}\left(\log_f N + 1\right)$	$\mathrm{O}\left(N\right)$	$\mathbf{O}\left(\mathbf{1}\right)$
Update (R)	$\mathrm{O}\left(N\right)$	$\mathrm{O}\left(\log_f N + 1\right)$	$\mathrm{O}\left(N\right)$	$\mathbf{O}\left(\mathbf{1}\right)$
Update (B)	$\mathrm{O}\left(N\right)$	$\mathrm{O}\left(\frac{fN'-1}{P(f-1)} + \lceil\log_f N'\rceil\right)$	$\mathrm{O}\left(\frac{N'+1}{P} + N\right)$	$\mathrm{O}\left(\frac{N'}{P} + \lceil\log_2 N'\rceil\right)$
Space	$\mathbf{O}\left(\mathbf{1}\right)$	$\mathrm{O}\left(\frac{fN-1}{f-1}\right)$	$\mathrm{O}\left(N+1\right)$	$\mathrm{O}\left(N+1\right)$

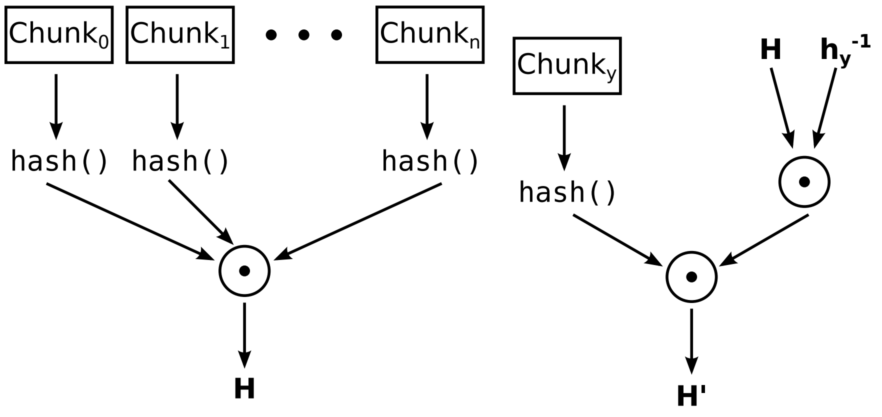
Figure 5.6: Running time of operations for three different hashing schemes. N is the number of blocks in a file, N' is the number of updated blocks in a batch update, f the fanout of the tree, and P is the number of processors. H stands for hashing, MT for Merkle Tree, SLMT for single-level Merkle Tree, and IH for Incremental Hashing [12]. For the updates, S stands for Sequential, for random, and B for batch.

hashing algorithm for updating a whole-file hash from a single write. In DS-VMI, we deal with single writes to a file all the time, and need a method of quickly computing a whole-file hash without re-reading all of the bytes of the file. Incremental hashing provides precisely this functionality.

In addition, incremental hashing is more space-efficient, requiring only $\mathrm{O}\left(N+1\right)$ storage. A Merkle tree requires $\mathrm{O}\left(\log_f N + 1\right)$ hashing operations to update a whole-file hash, and $\mathrm{O}\left(\frac{fN-1}{f-1}\right)$ space. However, for a large portion of file sizes experienced in practice the performance of Merkle trees may closely approximate that of incremental hashing. This is due to the decimation provided by having a large fanout at the bottom of the Merkle tree. This decimation keeps the Merkle tree short in height, requiring only a few extra hash operations over the simpler incremental hashing paradigm.

We envision extending DS-VMI with incremental hashing or Merkle trees for quick cross-VM whole-file deduplication. By batching updates across similar VMs, we expect whole-file deduplication to greatly reduce bandwidth requirements while providing continuous data protection to user-specified files. We need to perform a parameter sweep to determine the best default batching value across a mixture of write workload types.

Incremental hashing [12] is the only method which supports all of our unique hashing requirements. Incremental hashing provides a hash construction which supports random updates, is compact, requires no re-reading of data from the virtual disk, and offers collision resistance. As shown in Figure 5.7(a), incremental hashing works by splitting a file into n chunks, hashing each chunk using an "ideal" hash function, and combining the hashes using a group operation. As described in [12], a practical hash function could be from the SHA family, and an efficient group operator is modular addition or multiplication. Updating, shown in Figure 5.7(b), requires the inverse of the old chunk hash, the old hash, and the hash of the new block. Using DS-VMI we have the data needed as input to the hash function `hash`, and the other information must be stored as metadata. Modern file systems such as btrfs [99], and ZFS [133] already compute and store 256-bit hashes or checksums for every data block, thus external incremental hashing could have



(a) Incremental hashing of file chunks. (b) Updating an incremental hash.

Figure 5.7: The incremental hashing construction.

very low overhead. We propose using incremental hashing to implement file-level deduplication for `/cloud-history`, because it enables efficient re-computation of a whole-file hash on every write with no read requirements.

5.5 Storing and Indexing Historic State

`/cloud-history` collects file-level update streams via the DS-VMI mechanism introduced at the beginning of this dissertation. `/cloud-history` centrally demultiplexes these streams into per-file logs kept in a cloud secondary storage service. Each file log is separately garbage collected, versioned, and pruned. Users of `/cloud-history` may retrieve any version of any file across all of their virtual machine instances in the cloud. `/cloud-history` further maintains a unique file index over the files of individual tenants. This unique file index serves as the basis of efficient indexing. Building an index over backups, for example, which files harbor a critical vulnerability, is imperative for quick query response times.

In this section we describe our implementation of `/cloud-history`. This includes capturing writes with a hypervisor, and converting those writes into file-level update stream logs (Section 5.5.1). In addition, we discuss important operations over those logs such as versioning (Section 5.5.2), and garbage collection (Section 5.5.3). We finish the section by describing important optimizations making storage scalable, and indexing tractable: version deduplication to speedup indexing (Section 5.5.4), and supporting block-level optimizations for storage scalability (Section 5.5.5).

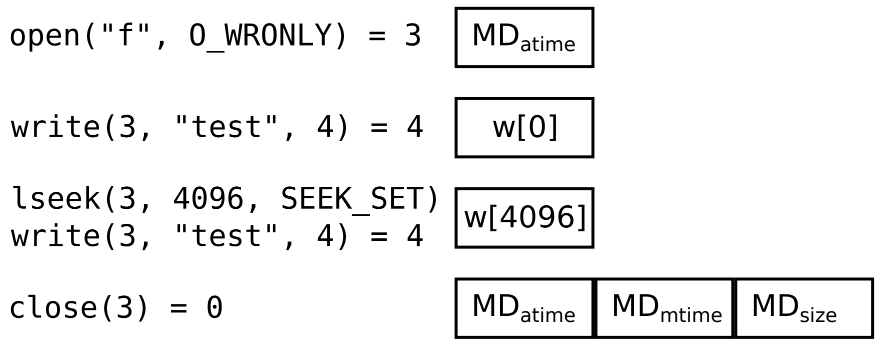


Figure 5.8: On the left-hand side are system calls occurring in userspace within a guest. These system calls cause block writes to a virtual disk. The corresponding block writes are introspected resulting in the high level events shown on the right-hand side of this figure.

5.5.1 Conversion to a File-level Update Stream

As shown in Figure 5.8, DS-VMI translates low-level operations initiated on behalf of guest actions into high-level events on files. `/cloud-history` captures these high-level events into a log file, and keeps one log per tracked file. Virtual machine introspection transforms block writes into a stream of semantically important file-level operations. Thus, a file system is represented by many such streams. This is contrary to the familiar single log structure of log-structured file systems. However, merging all of these streams into a single log is also possible if maximizing write bandwidth is desired. By operating on per-file streams, we have flexibility in this design space.

Log-structured files enable quick retrieval of the history of individual or sets of files. Traditional log-structured file systems must traverse much larger whole-file-system logs to reconstruct per-file historic state. In addition, log-structured files are self-contained logs of operations to individual files and are easily manipulated with tools outside of the context of an entire file system. For example, performing a diff on the history of two log-structured file streams is a cheap operation.

We do not believe that log-structured file streams are a drop-in replacement for whole-disk snapshotting. It is still valuable to be able to reinstate an entire byte-equivalent disk at some point in the past and directly boot from that version. Instead, log-structured file streams are useful by providing a log of file-level operations in between major whole-disk snapshots. Especially for forensics which often requires detailed logs of operations in order to understand exactly how a compromise occurred. They are also useful for architecting a backup system with efficient indexing and querying whole-file versions.

5.5.2 Inferring File Versions with Optimistic File Snapshotting

Creating versions out of file-level update stream logs is as simple as recording important locations in the log. We could of course let humans manually mark points in the log associated with important events. More than likely, the human marking the log chooses important points in time,

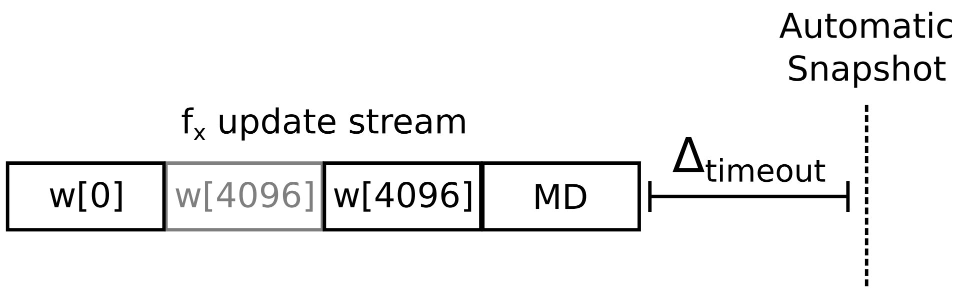


Figure 5.9: Files are optimistically snapshotted after waiting time Δ_{timeout}. MD stands for metadata, and the first write to position 4096 is gray because the second write supersedes it, representing an opportunity for garbage collection.

although it is possible that they inspect the file contents and make decisions based on the contents as well. However, ideally we would keep meaningful file versions without any human intervention whatsoever—completely automated. What type of semantics should we implement? Close-to-open semantics, as in AFS [49], map well into the expectations of users and are easy to understand. Every time you close a file a new version is created. However, with DS-VMI we do not have any insight into the *system calls* of a monitored system. We only have the writes, thus we must pick some heuristic for creating versions of files. We settled on a timeout since last write heuristic as an approximation to a close system call. Empirically it has been shown that cold files tend to remain cold over long time periods. Thus, as time passes the probability of an actual close increases.

Introspection provides the what—file-level updates—but not the how—system calls. This means that `/cloud-history` has no clear boundaries to perform versioning unlike a guest-supported versioning file system. In-guest agents have the luxury of leveraging system call information such as `open`'s and `close`'s to choose key versioning points for a given file. For example, many distributed file systems such as the Andrew File System (AFS) [49] have close-to-open semantics for consistency. `/cloud-history` has no such luxury. Ideally, versions of a file match at least each `open–close` pair. This follows what a normal user might expect: they `open` files to modify them, and `close` them signifying the "completion" of some task. Although valuable, understanding such `open–close` pairs is impossible with introspection.

Optimistic File Snapshotting determines when to version a file by waiting a tunable timeout, represented by Δ_{timeout} in Figure 5.9. This timeout is reset with every update to a file in the update stream provided by DS-VMI. As studies show [71], practically no files experience continuous updates. Thus, we can choose a timeout after which no more updates will occur with high probability. Snapshots are just positions in the log file, and work by replaying logs up till the requested version. Once created, snapshots are aggressively deduplicated and compressed at a byte level. Naturally, such an approach is prone to false positives—we may infer versions which never logically existed in the context of the user's environment. Without costly human annotation or intervention, it will be very difficult to ensure they are correct.

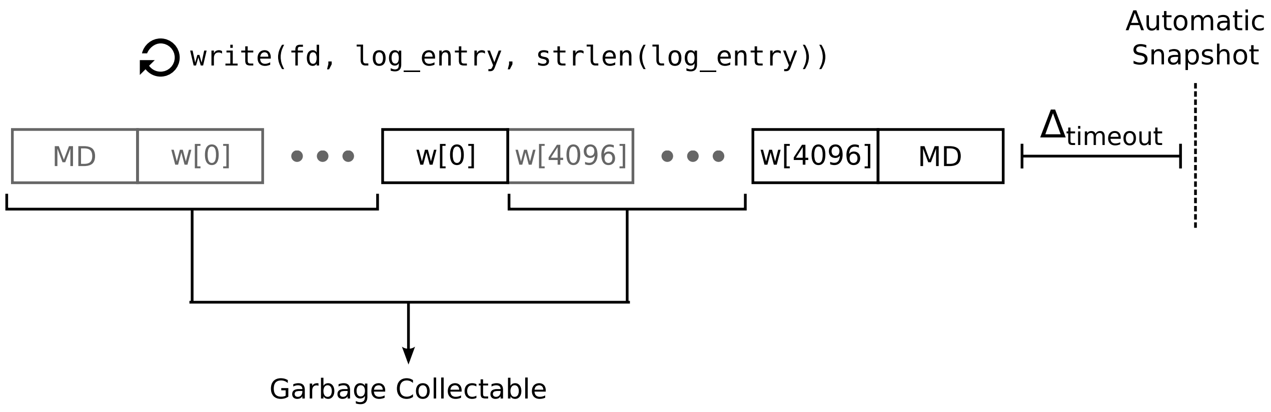

write(fd, log_entry, strlen(log_entry))
Automatic Snapshot
MD
w[0]
w[0]
w[4096]
w[4096]
MD
Δ_{timeout}
Garbage Collectable

Figure 5.10: Garbage collection is especially important for append-only style workloads, such as those for log files. Here we see some guest process continuously writes lines into a log file. Blocks are written many times with redundant log entry data. Only the latest updates matter when a version is created.

5.5.3 Garbage Collecting Stale Block Writes

Garbage collection works well for workloads which repeatedly touch the same positions in files. For example, logging programs repeatedly write to the end of log files. These constant, append-only style workloads manifest themselves as many log entries recording written data to the same portion—the end—of a file over and over again. As a concrete example, imagine the last data block associated with a file receiving 10 updates, each representing the whole block, but with log file induced changes. A version of a log file need only record the last write to any given block. An illustration of such an append-only scenario is shown in Figure 5.10. Random-write workloads, such as those generated by a database, benefit less from garbage collection. Retention polices also determine the efficacy of garbage collection. A retention policy setting a very low timeout, or desiring a version on every update, reduces the benefit of garbage collection. A retention policy determining versions with a large timeout or over large time scales benefits more from garbage collection.

In `/cloud-history`, garbage collection is implemented as a scheduled task that runs asynchronously from the rest of the system. Garbage collection can be paused and restarted at a later point in time. Typically, garbage collection runs at the same time as generation of versions based on timeouts as mentioned in the last section, but it can technically run at any time. Between the versions of a file, garbage collection keeps the last write to the unique set of positions written to in that version, as well as the last metadata update to the unique set of metadata entries updated in that version. This new stream is written out as a compacted file-level update stream. The original stream, where new file-level updates arrive, is truncated.

The effects of garbage collection on three different styles of workloads are shown in Figure 5.10. Garbage collection is an optimization which helps scale the storage used for retaining history.

5.5.4 Whole-File Indexing and Deduplication

Because VMs running in a cloud are derived from a small set of OS and application configurations, we anticipate large amounts of file-level duplication. These results are confirmed with the NC State VM disk dataset, and also by our study of backup. One of the major goals of `/cloud-history` is to enable indexing over large amounts of historic state. As shown in Figure 5.3(a), even with just a base set of 140 images, we have over 4.5 million files. Indexers running over such a large corpus take significant amounts of time. However, as shown in Figure 5.4, running indexers over just the set of unique files takes an order of magnitude less time.

We expect this trend to be magnified with our versions of files through time. The reasoning is simple: all of these running VMs will receive the same updates, similar configuration changes if managed by a single tenant, and probably similar deployed software for any individual tenant. Often, individual tenants specialize on a certain software stack. For example, many web applications expect similar back ends such as the popular Linux, Apache, MySQL, PHP (LAMP) set of applications. Thus, many file-level update logs will contain the same versions of files repeated across many VMs. Our hunch is confirmed with a crawl over real backups from a small research cloud as shown in Figure 5.2. Unfortunately, because `/cloud-history` only exists as a prototype, we do not have long-term collections of logs like we do with the backup study, but what we learn from the 3,000+ snapshots is applicable to file-level update streams.

The attractiveness of a first-level index over unique files is that this form of index is OS- and application- agnostic—a design goal of `/cloud-history`. Yet, this OS- and application- agnostic index provides tremendous benefit to index-style workloads, precisely the type of workloads we want to support with `/cloud-history`.

Periodically, similar to garbage collection, the whole-file indexer executes and computes incremental hashes over all log-structured file streams. Incremental hashes are saved at each versioning point into a database for quick retrieval. Thus, each version is still kept on-disk and only a deduplicated index kept in the database. We assume that underlying storage technologies will capture this duplication via block-level deduplication. Thus, our whole-file deduplication index is maintained solely to cut down the computation time of indexing.

5.5.5 Block-Level Deduplication and Compression

We expect many duplicate files across VMs especially when tasks such as updates execute. We also expect duplicated files within VMs as some applications write to a temporary file and then atomically rename the file to its permanent location. Thus, supporting block-level deduplication to save storage space across multiple files and multiple disks is important. We could implement complex pointers directly in the log format. However, keeping these pointers up-to-date during events such as garbage collection and log compaction unnecessarily complicates the logic required to manage file-level update stream logs. Hence, we assume underlying storage technology performs block-level deduplication. As long as the file-level update stream log logic stores writes on block boundaries

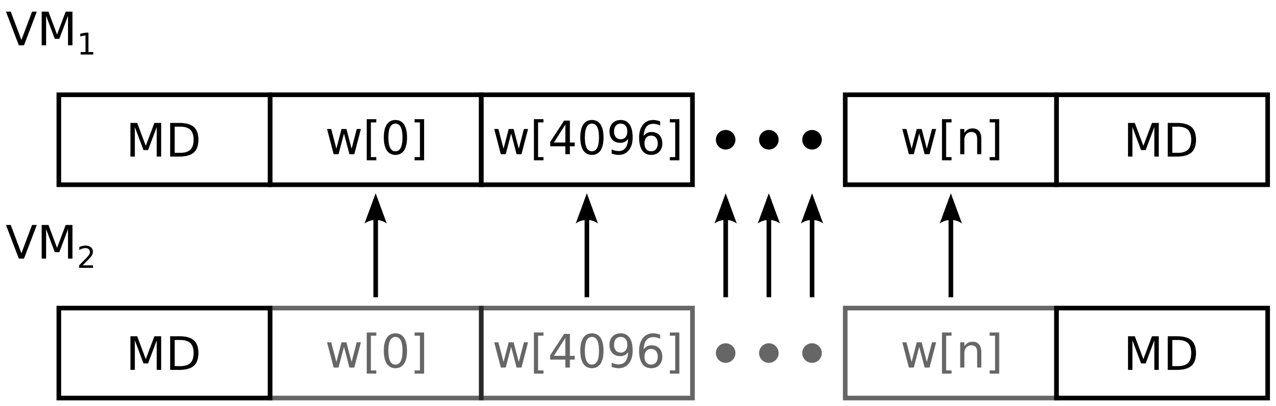


Figure 5.11: This figure shows two VMs (notionally, they could be the same VM) writing the same file. Their metadata which includes timestamps probably differs, but the data of the files is identical. Block-level deduplication is necessary to reclaim this wasted storage space. Also note that the metadata updates are often small and highly compressible.

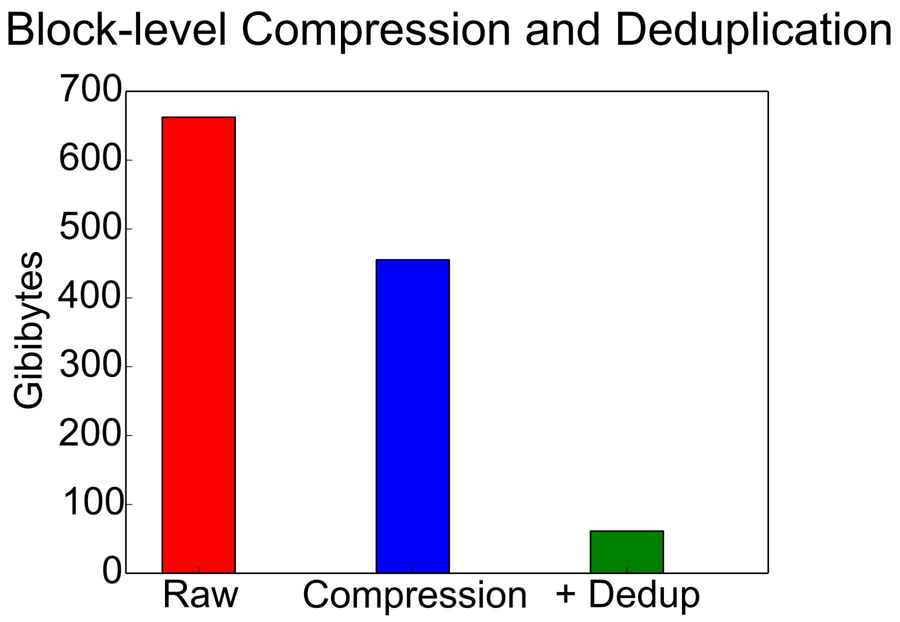


Figure 5.12: Shown are the effects of applying compression (LZ4) and block-level deduplication on the NC State dataset via ZFS.

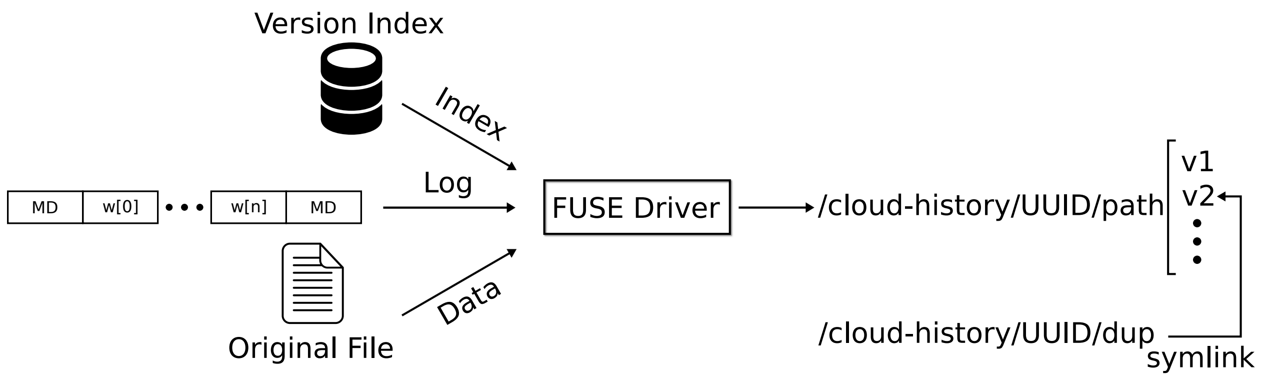


Figure 5.13: `/cloud-history` exposes versions of files via a simple synthetic FUSE file system which gives legacy indexers access to historic state without being rewritten. This figure shows that reads may come from the original file, or the file-level update stream log. Duplicate file versions are symlinked to a canonical version.

matching the blocks of the underlying storage platform, block-level deduplication proceeds without complication.

While it is valuable to have whole-file deduplication, research has shown that block-level deduplication is extremely valuable in the context of backups. Thus, we architected the log-structured on-disk layout be friendly to block-level deduplication. We assume that all of our file-level update streams are stored in storage that supports block-level deduplication. Modern file systems for example, ZFS or btrfs, both support block-level deduplication. We structured the log-file update stream's on-disk layout such that data blocks are aligned to the blocks of the underlying storage.

This means that given underlying storage that efficiently implements block-level deduplication and compression, `/cloud-history` automatically benefits from these savings. As an example of the benefits of block-level deduplication over whole-file deduplication see Figure 5.12.

5.6 Reconstructing File Versions

When an index does not exist for a specific query, the search mechanism needs efficient direct access to object-level contents. This means queries need to run as close as possible to the data to reduce as many bottlenecks as possible. In addition, many objects may be duplicated due to backups containing duplicates over time as well as across backed up systems running similar environments. Reducing the number of objects searched through is an important capability to make such search tractable [5, 96, 97, 105]. Modern backup systems can contain trillions of objects, but duplication often accounts for 90% or more of the storage space [110, 123].

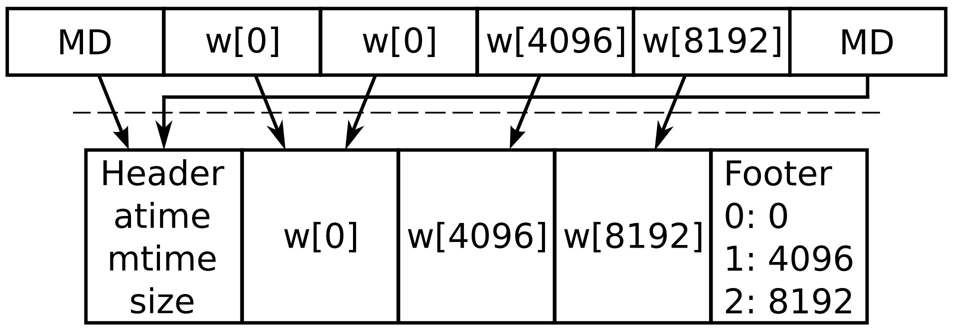


Figure 5.14: This shows the final version format after garbage collection, hashing, and timeout policies are applied. Note that metadata updates are coalesced in the header, and data updates are coalesced into a distinct set of writes to each block of a file. Each part of a version takes up a single on-disk block.

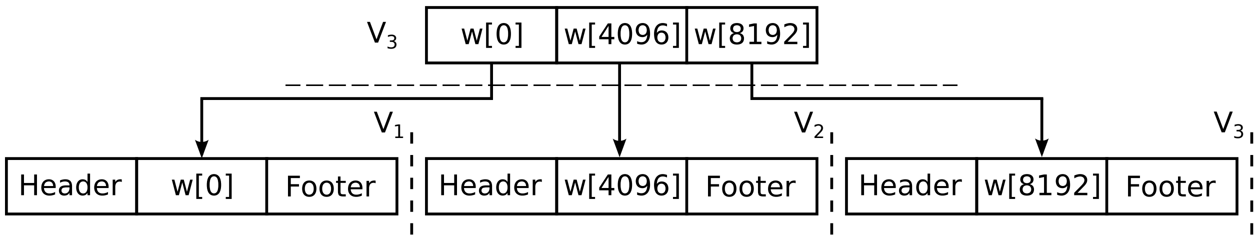


Figure 5.15: Reading files means reconstructing state from potentially multiple historic versions.

5.6.1 Efficient Object-Level Access

A backup system must minimize the overhead involved in accessing objects at the granularity of user queries. Overhead in accessing objects directly affects overall backup query time. The mismatch between the granularity of a query and the granularity of the backup system creates a fixed overhead. Ideally, this overhead is zero. In reality, the granularity of all future user queries is unpredictable. There always exists hypothetical queries with pathological overhead for any backup system. Thus, a tension arises between the level of granularity to store objects and the ease of implementing that granularity.

For example, on one extreme a backup system could implement sub-file, record-level indexing. This results in an explosion of metadata to track, as well as difficult maintenance. File formats experience significant churn due to tweaking or new emerging standards. On the other hand, a backup system could implement whole disk, block-level tracking. This results in minimal indexing and metadata tracking, and is extremely simple to maintain over time. Simplicity has led to this method being the default backup strategy in many backup systems.

We believe that whole-file granularity is the right granularity for a backup system to implement because it minimizes overhead to accessing objects, except at the record-level, and it is not difficult to maintain. Whole-file granularity requires indexing file systems. This does not pose a problem because file system on-disk data structures and layout change much more slowly than internal file formats. In addition, changes to file systems are often backwards-compatible. We therefore expect low long-term maintenance costs for file-system parsing modules. In addition, most modern file systems have excellent open source drivers and tools. Thus, whole-file granularity comes at low cost to the backup system.

5.6.2 Arbitrary Query Search

We cannot predict all potential future user queries. Thus, a backup system must maximize the freedom of query expression. Future potential queries range from the complex, "find all binaries vulnerable to a new zero-day vulnerability," to the simple, "find the latest version of my accidentally deleted document." Simple queries are generally handled well by most modern backup systems. This is because all backup systems have to track the timestamps of backups. They can therefore trivially serve simple queries looking for the latest version of important data.

Complex queries require carefully thought out architecture to enable high scalability and minimize query time. In-situ computation over backup data provides the highest form of scalability by discarding objects as early as possible [50]. Early discard minimizes wasted bandwidth and reduces overall backup system load. The query mechanism must also support arbitrary expressiveness by allowing any possible query from a user. Ideally, a user expresses queries using tools and environments familiar to them. For example, a DBA wants to express a historic query using SQL within a GUI-based client familiar to him or her. The ideal backup system fluidly conforms itself to user-familiar interfaces. Such a backup system would disrupt the status quo which forces conformance to the limited, often proprietary, interfaces that are generally provided. User-familiar interfaces require no training, and no time wasted or fidelity lost translating a query from a domain-specific to a domain-generic interface.

5.6.3 Evaluating Inferred File Versions

Figure 5.16 shows what happens with no guest coordination when versioning every 30 seconds. Only two versions match out of 12-13 versions. This implies a mismatch between the frequency of guest `sync` operations and the versioning of files. Fundamentally, we can only match versions across VMs if we match the underlying `sync`. This means that we must version at least twice as quickly as the highest frequency `sync` that we wish to match.

5.7 Securing Search

For the entirety of this chapter, we made the assumption that the user trusted their cloud with their raw backup data. What if the user does not want to trust the cloud? What if they want to store encrypted backups? We envisioned this case, and explored [97] architectures supporting

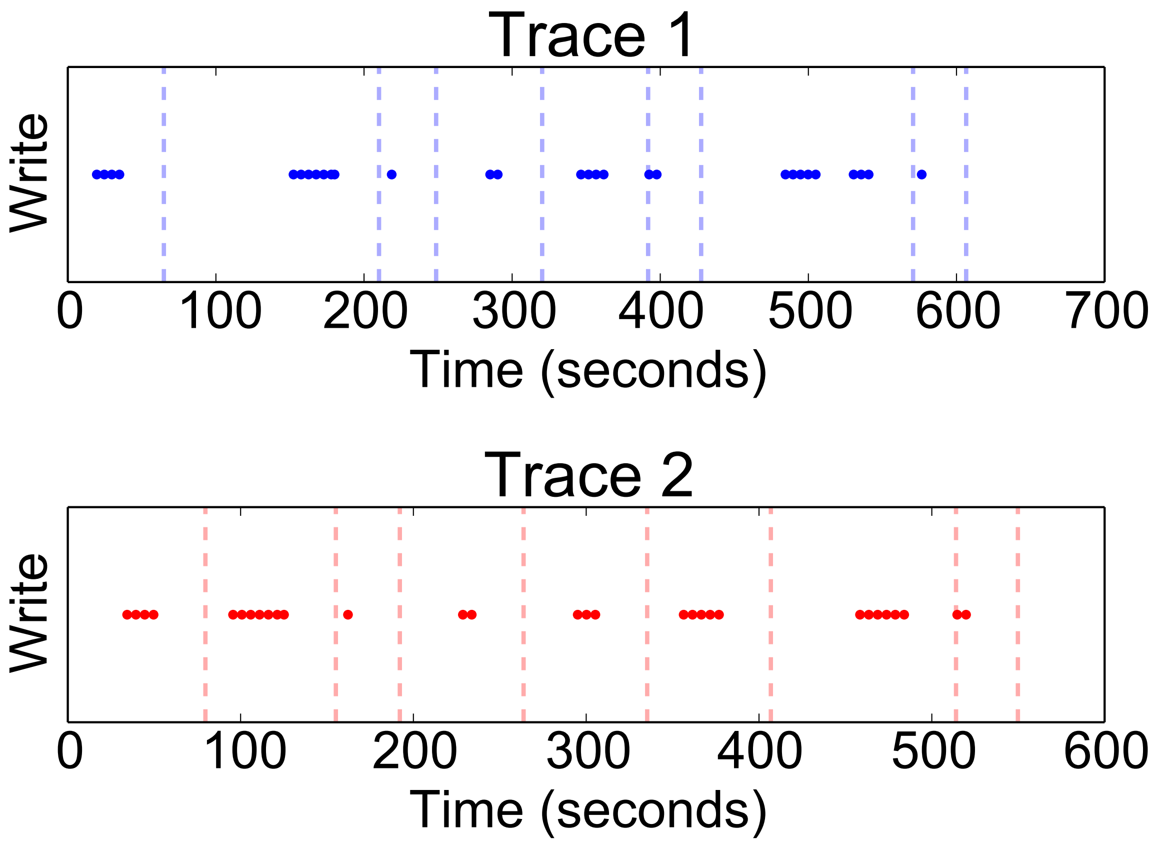


Figure 5.16: The arrival times of writes to a file without using `sync()` from within the guest OS. With an aggressively synchronizing guest OS, only 50% of the file versions matched. This demonstrates the worst case, that versions might not match always. Note that the history was compressible, for both traces we collected between 42 - 47 MiB which compressed down into 900 - 700 KiB.

it via convergent encryption [35]. At the time of indexing, users provide keys to decrypt stored objects—presumably files, although they can be at any granularity. We decided on using a convergent encryption scheme per-user as that allows for file-level deduplication without revealing the contents of the files. Each cloud user can key their convergent encryption algorithms with different keys, thus it would be impossible for a cloud to know if one user has the same files as another user. Frequency analysis would still be possible, and this means the only truly secure method requires abandoning any deduplication index. If users deem that tradeoff worth it, they are free to use non-convergent cryptography with the understanding that indexing becomes very costly, potentially intractable depending on the number of and sizes of files.

We explored three distinct architectures for indexing user-encrypted data: (1) one in which users trust the cloud, (2) one in which users trust the cloud and a key escrow service, and (3) one in which users trust no third party with encryption keys. If users trust the cloud, then as the cloud versions file-level update logs it encrypts the blocks differentiating each version with a key derived via a hash of those blocks. Thus, if a file experienced 10 block updates in a new version, those 10 blocks would be convergently encrypted with their hash and that hash stored encrypted using a key provided by the user. The cloud saves file version keys, but users control their key used to encrypt them. If users do not trust the cloud, but they do trust a key escrow service, they can convergently encrypt their files before they are saved to disk. This still exposes duplicate files to the cloud, but leaves the keys out of its reach. The key escrow service keeps keys for users, and performs decryption operations when the cloud indexes their files. In the last setting, the user trusts no third parties. In this case they still convergently encrypt their files before saving them to disk, but they also keep the keys instead of sharing them. If they want their files further indexed by the cloud, they must present the per-file keys at indexing time. Alternatively, users could index their files using their own compute instances pulling encrypted files from `/cloud-history`.

As a simple initial implementation, we allow the user to provide decryption keys when beginning their indexing or search operation. Thus, at search or index time, the user sends a list of 3-tuples: $< pathname, \ encryption\ method, \ key >$. This list is used to retrieve files from backup storage before they are indexed or searched. This mechanism could be extended to be less verbose and cumbersome for searches of large scope. This requires striking a balance between usability, privacy and performance. At one extreme is a single encryption key for an entire VM image. The other extreme (our initial choice) is a key per file within a VM image. A hierarchical file system within a VM image offers natural directory-level or subtree-level aggregation possibilities for intermediate points of this spectrum. This would require augmenting the 3-tuples mentioned above with an element denoting the granularity of the decryption key: $< granularity, \ pathname, \ encryption\ method, \ key >$, with possible granularity values of "VM image," "subtree," or "file." In addition, we are exploring the option of giving users direct control of the search infrastructure in the cloud for their searches such that they never reveal keys to any other party. This final path maintains privacy while providing a service that scales with the scope of searches.

Chapter Six

Future Work

In this chapter we discuss future directions that continue this line of research. The first two directions are incremental pieces of work and are more experimental rather than exploratory. Section 6.1 describes a necessary continuation of determining the nature of staleness and how it affects workloads in `/cloud`. Section 6.2 describes the important questions left to answer about `/cloud-history`. Section 6.3 describes a research idea with preliminary validation in the research literature which automatically generates introspection drivers. It would enable rapid, widespread adoption of introspection for any file system if proven scalable and robust. Section 6.4 discusses deriving an introspectable file system from first principles. Research down this path would answer the question, how can the architecture and on-disk layout of a file system be architected to support introspection as a first-class citizen? Ideally, it would not need to compromise on performance, but future work decides whether this is possible or impossible. Finally, Section 6.5 describes creating a filtering language for supporting and implementing file-level policies.

6.1 Evaluating the Effect of Staleness in `/cloud`

Remember that `/cloud` is an eventually consistent interface to files. It is at the mercy of the monitored system's file system logic and kernel policies. If the file system batches updates to disk this delays their visibility to `/cloud`. Similarly, if a kernel stores writes in a large page cache for tens of seconds, `/cloud` must wait until the kernel flushes the writes. For example, we have observed during experiments, although not studied in-depth, that Windows and NTFS can take up to 30 seconds and sometimes longer to emit writes to a storage device. Understanding the bound of this staleness, and portion of the file system tree that it affects is important for defining the limitations of `/cloud`. Finding the answer provides a precise meaning to the oft cited, but rarely precisely defined, eventual consistency. While we know `/cloud` is eventually consistent within at most a few minutes from an actual event, what is the lower bound for different operating system and file system combinations? What about the worst case latency? How many files and folders would this affect in practice? Should we tune monitored systems to synchronize more often to persistent storage, or are the eventual consistency bounds reasonable?

Under various workloads, with and without tuning the kernel of the monitored system, we need to collect the following metrics (1) average time from event to `/cloud` visibility, (2) average portion of the file system tree affected by staleness. For the first question, an example worst case answer would be many tens of minutes, rendering `/cloud` less useful. For the second, an example worst case would be the entire file system tree from the root.

6.2 Evaluating the Design Decisions of `/cloud-history`

`/cloud-history` has microbenchmarks and early results for all of its major design points. However, a deeper full end-to-end evaluation has not been carried out yet. Here we consider the remaining top questions such an evaluation needs to answer. We came up with he following four top-level questions:

1. How tractable is indexing large amounts of historic data?
2. Using DS-VMI, will ingest rates be enough for modern datacenters?
3. Are temporal-based file versions good enough? Or are content-based versions required to more closely match user expectations?
4. Do users prefer file-level retrieval, or whole-disk restoration when handling archival data? Does the coarse granularity of whole-disk backup render it less useful?

For the first question, we have early validation in the form of the backup study. However, we need to explore more indexing applications to ensure performance remains viable. The second question requires a detailed look at not only the efficiency of the log formats, but also their overall network bandwidth requirements. Many streams may not be scalable, even though one stream is highly amenable to certain optimizations. Per-VM and inter-VM log files are possible—but will they degrade performance past the point of usefulness? We currently have only implemented temporal-based file versioning which, at first glance, seems most intuitive to human users. However, this choice may not match meaningful user-level events such as saves when writes are rapid, reordered, or arbitrarily delayed by the kernel. This requires a study about how well temporal- vs content-based approaches match what the human user expected. However, the capability of rapid indexing may obviate the need for other forms of versioning file-level update logs. Users could use indexing algorithms to define the versions of files which are important to them. In this manner, indexes serve as filters over backup data. Finally, answering the last question—what do users prefer—requires a usability study. Such a study would enable us to make qualified statements about the concrete benefits of file-level indexable backup.

6.2.1 Sustained Ingest Bandwidth: Single vs Many Logs

Backup systems have two primary metrics. One is the amount of storage they need to ensure the safety of primary live data. By using compression and deduplication, backup systems can end up using much less storage then they logically contain. The second metric is their ingest bandwidth. In other words, how quickly can they complete a backup? In our discussion of `/cloud-history`, we considered saving a log per file across all monitored file systems. It is currently unclear how much this design decision will affect the ingest bandwidth of `/cloud-history`. Future experiments should tease out the effect of merging file-update streams at various levels. For example, the file-update streams of an entire file system could be merged together into one stream. Merging streams may incur a cost on other operations such as garbage collection.

6.2.2 Measuring Time to Index

Preliminary results show that file-level deduplication is a clear win for cutting down the time to index. What remains to be answered is how does storing file data within file update log streams affect indexing time? Do the logs hinder or aid indexers? Presumably, reconstruction time of historic file versions will negatively impact the time to index. File update logs should be compared against other techniques for storing versions of files such as copy-on-write file systems implemented with tree structures such as btrfs [99]. We expect file update logs to perform comparable to a log-structured file system, and they can be directly compared to the NILFS [82] file system.

6.2.3 Effect of Hashing Choice

We proposed using incremental hashing for computing whole file hashes which theoretically outperforms both traditional hashing and Merkle trees for random updates. This performance edge on handling random updates makes incremental hashing an ideal candidate for computing a hash on file update log streams with low latency. This low latency directly translates into higher achievable bandwidth over streams of random updates, even when performed offline. Although we explored incremental hashing for its perfect computational fit, it actually should exhibit a higher ingest rate—one of our key metrics—of file update logs. Head-to-head experiments should be done comparing the various hashing methods performance on recomputing a hash with random updates. The key metric is bandwidth in terms of bytes processed per unit time. A secondary metric is the space used to store the resulting hash. For example, a Merkle tree needs space to store a tree of hashes.

6.3 Generating Introspection Code from File System Drivers

Early validation by Virtuoso [34] shows that introspection code can be generated with a minimal helper application inside the monitored system during a training phase. This thread is very promising, because it hints at a mechanized process for the current very manual engineering task of writing introspection code. Indeed, the ability to generate such code deterministically implies that the file system is sound and consistent. This introspection code serves as a run-time check or hypervisor-verifiable proof that the file system is running correctly.

Generating introspection code by hand is a major roadblock impeding the progress of general introspection across modern clouds today—especially for closed-source kernels. An approach that is machine-performed, and thereby scalable, would completely change the introspection landscape and provide a new wave of tooling for clouds to leverage. As we have shown in this dissertation, scalable new interfaces to cloud-wide persistent state then become possible, and mechanized generation of introspection code makes scalability across OS versions and types trivial.

Introspection code generation works presently via two main techniques. The first technique could be considered supervised with access to ground truth in the form of kernel source code. It has so far worked with fair success [41]. The second technique is unsupervised without access to kernel source code. Generally, this method works with a "helper" application [34] running inside the guest writing known patterns to virtual storage. The introspection generator watches for these patterns and then infers the relationship between blocks on disk. This technique has also met with limited success in the past. A breakthrough maturing one of these techniques is necessary to make introspection a status quo reality in the future.

6.4 Designing an Introspectable File System

As far as we know, no file system has ever been developed within the context of virtual computing with introspection as a supported feature. Developing an introspectable file system is compelling for debugging purposes. Introspection enables run-time verification of writes as they are flushed from a kernel to storage devices. Thus, an introspectable file system becomes easier to develop, debug, and externally verify. The closest known work in the research literature is ReconFS [67]. ReconFS embeds inverted pointers with every written page. These pointers are written in the scratch space of flash pages. Using these pointers, the file system can be reconstructed even if core datastructures are damaged.

Reverse pointers as in ReconFS, especially with every single page, would immensely ease the implementation of introspection. It seems possible for a small augmentation which could be retrofitted to legacy file systems making them introspection-friendly. We need just enough information to understand the type of a block and how it fits into the overall file system. However, the absolute minimum mapping we want is from block to file. Thus, the only true pointers we need are from a block to a byte stream within a file, and a notion of the full path of a file. Both of which could be embedded into the notional "scratch" space of a page on a flash device, or added as a header to every block on the storage device. This reverse mapping, as in ReconFS, also makes the file system resilient to failures even those which wipe out core file system data structures.

The minimal data structure for a data block is (1) file byte offset start, (2) file final block flag, (3) file offset finish which would only be present for final blocks, and the file's name or a pointer into a table storing the absolute path. Ideally the path information would be embedded as part of the reverse mapping data structure as well. Directory blocks have a flag set denoting their status as being a list of path information. Directories could be spread throughout the disk as other files are, or centrally located in a large table like NTFS's MFT.

6.5 A Storage Introspection Language for Policy Enforcement

This dissertation was entirely concerned with the mechanism of file-level data capture, its implementation with multiple interfaces on top, and its evaluation. In Chapter 1 we described what was out of scope for this dissertation and one such topic was enforcement of policy via introspection. However, paralleling networking research, development of a filter language and mechanism over storage traffic seems inevitable for enforcement of policies. Also, having such a language makes it easier to implement introspection tasks. It would be desirable if the language was flexible enough to implement introspection. Even better if the language itself could make implementing introspection for a new file system faster than without it.

There are unique challenges which distinguish creating such a language, and enforcing it on a write stream, from the creation of the network community's packet filtering languages. First, many writes often need to be buffered before the overall operation within the file system can be understood. This is in contrast to network protocols such as TCP which have sequence numbers clearly defining the number of outstanding writes and order of a stream. Second, there are many more file systems in use than network protocols. Predominantly, the standard protocols in networking are less than a handful: IP, TCP, UDP, and today some would add HTTP. However, the file systems in use today are more than a handful and include FAT32, NTFS, ext4, ZFS, ReiserFS, XFS, NILFS, Btrfs, GPFS2, and many more. Each one has its own datastructures, on-disk layout, and algorithms for sending data to disk. In addition, writes to file systems are wrapped within a hardware-specific protocol such as SATA or IDE. These factors combine and force a rethink of packet filter and the design of policy languages when applied to storage.

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 worldwide 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 Management 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.

	Granularity of			Level of			Type of		
System	**Time**	**State**	**Query**	**Consistency**	**Scale**	**Backup**	**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.

Chapter Eight

Conclusion

This thesis proposed Distributed Streaming Virtual Machine Introspection (DS-VMI), a fundamentally new way of accessing persistent state—agentlessly—without the cooperation of the writer. Although this thread of research was born in cloud computing, it is applicable to any writer sending structured data to persistent storage. For example, a stream of writes to a network attached storage system could be introspected at the level of network packets as a form of deep packet inspection. The goal of this thesis, which guided the design of its interfaces and optimizations, was to find an agentless, application-agnostic, near-real-time view into persistent cloud-wide file-level state. With DS-VMI, we achieved this goal. As a mechanism, DS-VMI is only as useful as the interfaces layered on top of it. We took a holistic approach by designing three interfaces covering the various types of persistent state—live and archival—and the various types of workloads—batch and event-driven.

The first interface, `cloud-inotify` as described in Chapter 3, implements a publish-subscribe, eventually consistent, selective view over persistent state designed for event-driven workloads. Although it requires rewriting legacy applications to its channel-based subscriptions, it is an application and OS agnostic interface. External monitoring agents using `cloud-inotify` benefit by not needing completely separate implementations per monitored writer. Again, this is similar to deep packet inspecting firewalls which are not dependent on the operating system environments of the network hosts they monitor. The second interface, `/cloud` as described in Chapter 4, implements a strongly consistent view over persistent state with a read-only POSIX file system designed for batch workloads. Such a simple interface—a file system—enables compatibility with legacy applications without any extra work.

`cloud-inotify` and `/cloud` both interface with live persistent state. However, significant effort is also spent on archived information kept in the form of backups. Historic data holds answers to many key questions that a business must answer. For example, to fulfill its responsibility to customers, a business needs to sift through historic logs and file-level clues in the event of an information security breach. Tracing the flow of information, systems accessed, and steps taken by attackers is critical information in determining which customers are affected. In order to get a better understanding of the backup solution space, we performed an in-depth research study of over 30 backup systems in Chapter 7. We found a dearth of backup systems which could support quick random-access querying and indexing with agentless capture of state. Yet trends in archival storage such as Google's Nearline show that the future of backup includes the heretofore unthinkable feature of low-latency, random access to backup data. Services such as Dropbox and Backblaze have begun indexing historic file-level state for consumers for some applications such as fulltext document search, but the vision of this thesis is application agnostic interfaces. Thus, we designed an interface leveraging DS-VMI to provide agentless backup to monitored systems with optimizations maximizing the utilization of storage space, while minimizing the time to index at a file-level.

`/cloud-history` as described in Chapter 5, implements an agentless backup system designed to capture versions of files. File versioning occurs on file-level update streams obtained via DS-VMI with versions derived by a timeout from the last modification. This timeout heuristic tries to model user open-close, and ideally modify-save, semantics. `/cloud-history` assumes fast access to archival storage. In the case of Google Nearline, this is now a publicly available reality. Our study, presented in Section 5.2, of a research backup system confirms that although backup data quickly balloons in scale due to the frequency of scheduled backups, it contains significant amounts of duplicate files. Previous backup studies [71, 123] find significant duplication at a block- or file-level, but report in numbers of bytes saved—not in numbers of duplicate files.

In this dissertation, we took a further step in studying the practical implications of such file-level duplication. We realized, experimentally, that general indexing of backup data at a file-level becomes practical with file-level deduplication. We found indexing every backed up file without file-level deduplication is prohibitively costly. File-level deduplication—an application agnostic optimization—reduces the file space by two orders of magnitude making indexing not only tractable, but reasonable enough to repeat with some regularity. In other words, it should be possible to iteratively and interactively explore backup data even when pre-existing indexes do not exist.

At a high level, this dissertation posed the question: is it possible to completely manage and monitor persistent state agentlessly without any writer support whatsoever? This idea goes against the two traditional models of either a single reader-writer with absolute control over local file systems, or distributed file systems. Although distributed file systems allow for multi-readers and multi-writers, they require the participation, and therefore configuration, of the aforementioned readers and writers. This dissertation developed a core technology abbreviated DS-VMI, which formed the foundation for three interfaces: `cloud-inotify`, `/cloud`, and `/cloud-history`. The unifying theme of finding solutions which are agnostic to the OS or application being monitored was a difficult goal, but realized in DS-VMI. This dissertation also contributes a prototype implementation including integration into real world cloud software called OpenStack. The answer developed in this dissertation leads to more questions.

What will the clouds of tomorrow look like? They will be more managed then today, deploying technologies similar to DS-VMI. By observing the DevOps movement, we see the immediate immense popularity of agentless tooling such as Ansible. This is because agentless solutions require less configuration of monitored systems, have no moving—breakable—parts within the monitored system, and are impossible to turn off or corrupt by the entity being monitored. It is difficult to argue against the value proposition of agentless technologies, except for their overheads. Could DS-VMI be made more efficient? We believe that it can, and it is only a matter of investing engineering effort. DS-VMI is efficient for most workloads except high bandwidth writes over extended time periods. To address this overhead, one low-hanging engineering fruit is implementing a zero-copy optimization between DS-VMI and the monitored writers.

In conclusion, the future looks bright for agentless monitoring and management tooling. The agentless value proposition is constructed from foundational guarantees that are unobtainable with agents. Put bluntly, agentless technologies are technically superior to agent-based monitoring. For example, we can guarantee zero impact on quality of service via agentless solutions, which is impossible to guarantee with agent-based solutions. Even if you took drastic measures, such as turning off the monitoring agent, agents could disobey, or may have already introduced a malicious entity into the system. DS-VMI demonstrates a path towards agentless monitoring of persistent storage across the cloud landscape. It is the first description of a unifying framework for addressing persistent storage at a file-level. DS-VMI embraces the now prevalent paradigm of software defined storage. By embracing modern storage's reality rather than clinging to an anachronistic, monolithic system design, this dissertation enables a new generation of outsourced monitoring applications.

Bibliography

[1] Acronis International GmbH. Full, incremental and differential backups, November 2012. URL `http://www.acronis.com/en-us/support/documentation/ABR11.5/index.html#1370.html`. Cited on pages 103 and 106.

[2] M. K. Aguilera, S. Spence, and A. Veitch. Olive: distributed point-in-time branching storage for real systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, Berkeley, CA, USA, 2006. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267680.1267707`. Cited on pages 93 and 94.

[3] AJ Lewis. Lvm howto: Snapshots, November 2011. URL `http://tldp.org/HOWTO/LVM-HOWTO/snapshots_backup.html`. Cited on page 100.

[4] M. Allalouf, M. Ben-Yehuda, J. Satran, and I. Segall. Block storage listener for detecting file-level intrusions. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, May 2010. doi: 10.1109/MSST.2010.5496974. Cited on pages 93 and 96.

[5] G. Ammons, V. Bala, T. Mummert, D. Reimer, and X. Zhang. Virtual machine images as structured data: The mirage image library. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2170444.2170466`. Cited on page 81.

[6] Ansible, Inc. Windows support — ansible documentation, August 2015. URL `http://docs.ansible.com/ansible/intro_windows.html`. Cited on page 98.

[7] Apple Inc. Mac 101: Time machine, January 2012. URL `http://support.apple.com/kb/HT1427`. Cited on pages 97, 101 and 106.

[8] Apple Inc. Fsevents reference, August 2015. URL `https://developer.apple.com/library/mac/documentation/Darwin/Reference/FSEvents_Ref/`. Cited on page 45.

[9] Backblaze. Backblaze cloud backup services give you peace of mind, November 2014. URL `https://www.backblaze.com/internet-backup.html`. Cited on pages 102 and 106.

[10] M. Banikazemi, D. Daly, and B. Abali. Sysman: A Virtual File System for managing clusters. In *Proceedings of the 22nd Conference on Large Installation System Administration Conference*, LISA'08, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1496684.1496699. Cited on page 98.

[11] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1247360.1247401. Cited on page 10.

[12] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-62975-7. doi: 10.1007/3-540-69053-0_13. URL http://dx.doi.org/10.1007/3-540-69053-0_13. Cited on pages xiv, 73 and 74.

[13] Benjamin Gilbert. Deltaic, November 2014. URL https://github.com/cmusatyalab/deltaic. Cited on pages 63, 103 and 106.

[14] C. Benninger, S. Neville, Y. Yazir, C. Matthews, and Y. Coady. Maitland: Lighter-weight VM introspection to support cyber-security in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, June 2012. Cited on pages 94 and 97.

[15] BSON. BSON, September 2012. URL http://bsonspec.org/. Cited on page 19.

[16] bup. bup, it backs things up!, November 2014. URL http://bup.github.io/. Cited on pages 100, 102 and 106.

[17] Canonical Ltd. Ubuntu Cloud Images, August 2015. URL https://cloud-images.ubuntu.com/trusty/. Cited on pages 10 and 52.

[18] Carbonite. Cloud backup 101, November 2014. URL http://www.carbonite.com/online-backup. Cited on pages 101 and 106.

[19] J. Case, M. Fedor, M. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990. URL http://www.ietf.org/rfc/rfc1157.txt. Cited on page 98.

[20] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002. ISSN 0734-2071. doi: 10.1145/571637.571640. URL http://doi.acm.org/10.1145/571637.571640. Cited on page 105.

[21] ClamAV. Clam AntiVirus, December 2013. URL http://www.clamav.net/. Cited on pages xii, 3, 7, 70, 97 and 98.

[22] Code 42 Software. Data backup software features - crashplan - free computer backup, November 2014. URL `http://www.code42.com/crashplan/features/`. Cited on page 102.

[23] Codenomicon. Heartbleed bug, July 2014. URL `http://heartbleed.com/`. Cited on pages 73 and 99.

[24] J. Cohen, T. Repantis, S. McDermott, S. Smith, and J. Wein. Keeping track of 70,000+ servers: the Akamai query system. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924976.1924999`. Cited on pages 35, 50 and 98.

[25] R. Coker. bonnie++, 2001. URL `http://www.coker.com.au/bonnie++/`. Cited on page 38.

[26] Colin Percival. Tarsnap - general usage, November 2014. URL `http://www.tarsnap.com/usage.html`. Cited on pages 102 and 106.

[27] C. Colohan. The "Scariest Outage Ever". SDI/ISTC Seminar Series, 2012. URL `http://www.pdl.cmu.edu/SDI/2012/083012b.html`. Cited on page 49.

[28] J. Corbet. ext4 and data loss, March 2009. URL `https://lwn.net/Articles/322823/`. Cited on page 15.

[29] Craig Barrett. Backuppc information, December 2013. URL `http://backuppc.sourceforge.net/info.html`. Cited on pages 102 and 106.

[30] R. R. Curd Zechmeister, Alex Tomic and J. Nunn. Enterprise backup and recovery on-premises to aws. Technical report, Amazon, December 2014. URL `https://d0.awsstatic.com/whitepapers/best-practices-for-backup-and-recovery-on-prem-to-aws.pdf`. Cited on page 61.

[31] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`. Cited on page 54.

[32] Déjà Dup Maintainers. Déjà Dup, November 2014. URL `https://launchpad.net/deja-dup`. Cited on pages 101, 102 and 106.

[33] M. Dillon. The hammer filesystem. Technical report, DragonFly BSD, June 2008. URL `http://www.dragonflybsd.org/hammer/hammer.pdf`. Cited on pages 70, 95 and 106.

[34] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on*, May 2011. doi: 10.1109/SP.2011.11. Cited on pages 89, 90, 94 and 97.

[35] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1585-1. URL `http://dl.acm.org/citation.cfm?id=850928.851884`. Cited on page 85.

[36] Dropbox. Dropbox, December 2013. URL `https://www.dropbox.com/`. Cited on pages 45 and 97.

[37] Druve. Scalable endpoint backup for cloud and onpremise, November 2014. URL `http://www.druva.com/why-us/massive-scalability/`. Cited on pages 103 and 106.

[38] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, 2011. URL `https://tools.ietf.org/rfc/rfc6455.txt`. Cited on pages xiii, 51 and 52.

[39] Free Software Foundation. Gnu tar, July 2014. URL `http://www.gnu.org/software/tar/`. Cited on pages 70, 104 and 106.

[40] Frost & Sullivan. Analysis of the SIEM and log management market. Technical report, Frost & Sullivan, `http://www.frost.com/sublib/display-report.do?id=NC9D-01-00-00-00`, November 2013. Cited on pages 6 and 97.

[41] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 586–600, May 2012. doi: 10.1109/SP.2012.40. Cited on page 90.

[42] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, 2003. Cited on pages 9 and 97.

[43] Git. Git basics - recording changes to the repository, 2015. Cited on page 57.

[44] H. Gobioff, D. Nagle, and G. Gibson. Embedded security for network-attached storage. Technical Report CMU-CS-99-154, Carnegie Mellon University, June 1999. URL `http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-154.pdf`. Cited on page 95.

[45] J. G. Hansen and E. Jul. Lithium: virtual machine storage for the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, 2010. ACM. ISBN

978-1-4503-0036-0. doi: http://doi.acm.org/10.1145/1807128.1807134. URL `http://doi.acm.org/10.1145/1807128.1807134`. Cited on pages 93 and 94.

[46] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the i/o behavior of apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: http://doi.acm.org/10.1145/2043556.2043564. URL `http://doi.acm.org/10.1145/2043556.2043564`. Cited on page 17.

[47] D. Hildebrand, R. Tewari, and V. Tarasov. Disk image introspection for storage systems, US Patent Pending 2011. Cited on pages 94 and 97.

[48] T. Hoff. 7 years of YouTube scalability lessons in 30 minutes, March 2012. URL `http://highscalability.com/blog/2012/3/26/7-years-of-youtube-scalability-lessons-in-30-minutes.html`. Cited on page 19.

[49] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6 (1), February 1988. Cited on pages 36 and 77.

[50] Huston, L., Sukthankar, R., Wickremesinghe, R., Satyanarayanan, M., Ganger, G.R., Riedel, E., Ailamaki, A. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, April 2004. Cited on page 83.

[51] IBM. IBM SmartCloud application performance management, December 2013. URL `http://www-01.ibm.com/support/knowledgecenter/`. Cited on pages 3 and 97.

[52] IBM Corporation. Best practices for KVM, April 2012. URL `http://www-01.ibm.com/support/knowledgecenter/linuxonibm/liaat/liaatbestpractices_pdf.pdf`. Cited on page 35.

[53] Inktank Storage, Inc. Rbd layering, December 2014. URL `http://ceph.com/docs/next/dev/rbd-layering/`. Cited on page 106.

[54] X. Jiang and X. Wang. "Out-of-the-box" monitoring of VM-based high-interaction honeypots. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID'07, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74319-7, 978-3-540-74319-4. URL `http://dl.acm.org/citation.cfm?id=1776434.1776450`. Cited on pages 94 and 97.

[55] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: http://doi.acm.org/10.1145/1315245.1315262. URL `http://doi.acm.org/10.1145/1315245.1315262`. Cited on pages 94 and 97.

[56] Jonas Borgström. Welcom to attic, December 2014. URL `https://attic-backup.org/`. Cited on page 106.

[57] J. Katcher. PostMark: A new file system benchmark. Technical report, NetApp, 1997. Cited on page 37.

[58] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, July 2007. URL `http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf`. Cited on pages xii, 10 and 14.

[59] Konstantin Isakov. Zbackup, November 2014. URL `http://zbackup.org/`. Cited on pages 103 and 106.

[60] L. Kufel. Security event monitoring in a distributed systems environment. *Security Privacy, IEEE*, 11(1), 2013. ISSN 1540-7993. doi: 10.1109/MSP.2012.61. Cited on pages 3 and 98.

[61] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-VII, New York, NY, USA, 1996. ACM. ISBN 0-89791-767-7. doi: http://doi.acm.org/10.1145/237090.237157. URL `http://doi.acm.org/10.1145/237090.237157`. Cited on pages 93 and 94.

[62] S. J. Leffler and M. K. McKusick. *The design and implementation of the 4.3 BSD UNIX operating system*. Addison-Wesley, 1989. Cited on pages 16 and 50.

[63] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 132–140, New York, NY, USA, 1975. ACM. doi: 10.1145/800213.806531. URL `http://doi.acm.org/10.1145/800213.806531`. Cited on page 2.

[64] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai. Efficient batched synchronization in dropbox-like cloud storage services. In D. Eyers and K. Schwan, editors, *Middleware 2013*, volume 8275 of *Lecture Notes in Computer Science*, pages 307–327. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-45064-8. doi: 10.1007/978-3-642-45065-5_16. URL `http://dx.doi.org/10.1007/978-3-642-45065-5_16`. Cited on pages 102 and 105.

[65] Linux Kernel Organization, Inc. Linux kernel archive, September 2012. URL http://www.kernel.org/. Cited on page 36.

[66] Loggly Inc. Log management service in the cloud - Loggly, March 2013. URL http://loggly.com/. Cited on page 97.

[67] Y. Lu, J. Shu, and W. Wang. Reconfs: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 75–88, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-08-9. URL http://dl.acm.org/citation.cfm?id=2591305.2591313. Cited on page 90.

[68] Maginatics. Superior data protection with maginatics, November 2014. URL https://maginatics.com/resources/whitepapers/data-protection. Cited on pages 102 and 106.

[69] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7), 2004. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2004.04.001. URL http://www.sciencedirect.com/science/article/pii/S0167819104000535. Cited on page 3.

[70] J. McCutchan. [RFC][PATCH] inotify 0.8, 2004. URL https://groups.google.com/forum/#!topic/fa.linux.kernel/Y2aqoQyyO8w/discussion. Cited on page 45.

[71] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *Trans. Storage*, 7(4): 14:1–14:20, Feb. 2012. ISSN 1553-3077. doi: 10.1145/2078861.2078864. URL http://doi.acm.org/10.1145/2078861.2078864. Cited on pages 64, 77 and 110.

[72] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-013-5. doi: http://doi.acm.org/10.1145/1352592.1352598. URL http://doi.acm.org/10.1145/1352592.1352598. Cited on pages 93 and 94.

[73] Michael Rubel. Easy automated snapshot-style backups with linux and rsync, January 2004. URL http://www.mikerubel.org/computers/rsync_snapshots/. Cited on pages 45, 102 and 106.

[74] Microsoft. Microsoft extensible firmware initiative fat32 file system specification. Technical report, Microsoft Corporation, December 2000. URL http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc. Cited on page 27.

[75] Microsoft. Windows Management Instrumentation, October 2013. URL http://msdn.microsoft.com/en-us/library/aa394582(v=vs.85).aspx. Cited on page 98.

[76] Microsoft. What is system restore?, November 2014. URL http://windows.microsoft.com/en-us/windows/what-is-system-restore#1TC=windows-7. Cited on pages 101 and 106.

[77] Microsoft. Filesystemwatcher class, August 2015. URL https://msdn.microsoft.com/en-US/library/system.io.filesystemwatcher.aspx. Cited on page 45.

[78] S. Microsystems. Zfs on-disk specification. Technical report, Sun Microsystems, 206. URL http://www.giis.co.in/Zfs_ondiskformat.pdf. Cited on page 55.

[79] J. C. Mogul. Simple and flexible datagram access controls for unix-based gateways. In *Proceedings of Summer 1080 USENIX Technical Conference*, 1989. Cited on page 2.

[80] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, 1987. Cited on pages 2 and 13.

[81] P. Newson. Google cloud storage nearline. Technical report, Google, July 2015. URL https://cloud.google.com/files/GoogleCloudStorageNearline.pdf. Cited on pages 5 and 61.

[82] Nippon Telegraph and Telephone Corporation. Nilfs - continuous snapshotting filesystem for linux, November 2014. URL http://nilfs.sourceforge.net/en/. Cited on pages 89, 95, 101 and 106.

[83] NIST. National Vulnerability Database (CVE-2012-3448), February 2013. URL http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3448. Cited on pages 2, 3 and 98.

[84] NIST. National Vulnerability Database (CVE-2015-2668), April 2015. URL http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-2668. Cited on pages 2, 3 and 98.

[85] OpenStack Foundation. OpenStack open source cloud computing software, December 2013. URL http://www.openstack.org/. Cited on pages 13 and 46.

[86] Oracle. Overview of zfs snapshots, 2012. URL https://docs.oracle.com/cd/E23824_01/html/821-1448/gbciq.html. Cited on page 106.

[87] B. D. Payne. Simplifying virtual machine introspection using libvmi. Technical Report SAND2012-7818, Sandia National Laboratories, September 2012. URL http://prod.sandia.gov/techlib/access-control.cgi/2012/127818.pdf. Cited on page 98.

[88] B. D. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, dec. 2007. doi: 10.1109/ACSAC.2007.10. Cited on pages 94 and 97.

[89] J. Pfoh, C. Schneider, and C. Eckert. A formal model for virtual machine introspection. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '09, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-780-6. doi: http://doi.acm.org/10.1145/1655148.1655150. URL `http://doi.acm.org/10.1145/1655148.1655150`. Cited on page 97.

[90] X. Project. Blktap - xen, April 2013. URL `http://wiki.xenproject.org/wiki/Blktap`. Cited on page 12.

[91] QEMU. Changelog/1.6 - qemu, November 2014. URL `http://wiki.qemu.org/ChangeLog/1.6#Block_devices_2`. Cited on pages 13 and 100.

[92] QEMU. Mirroring commands, August 2015. URL `http://wiki.qemu.org/Features/BlockJob#Mirroring_commands`. Cited on page 12.

[93] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1083323.1083333`. Cited on pages 100 and 106.

[94] Recoll. Recoll text search finds your documents., December 2015. URL `http://www.lesbonscomptes.com/recoll/`. Cited on page 70.

[95] Retrospect. Competitive analysis retrospect and our competition, August 2014. URL `http://download.retrospect.com/partners/sales_tools/competitive/win_9_mac_11/retrospect_competitive_analysis_fall_2014_enusd.pdf`. Cited on page 106.

[96] W. Richter, G. Ammons, J. Harkes, A. Goode, N. Bila, E. de Lara, and M. Satyanarayanan. The manna plug-in architecture for content-based search of vm clouds. Technical Report CMU-CS-10-111, Carnegie Mellon University, August 2010. URL `http://reports-archive.adm.cs.cmu.edu/anon/2010/abstracts/10-111.html`. Cited on page 81.

[97] W. Richter, G. Ammons, J. Harkes, A. Goode, N. Bila, E. de Lara, V. Bala, and M. Satyanarayanan. Privacy-sensitive VM retrospection. In *Proceedings of the Third USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud '11. USENIX Association, 2011. Cited on pages 81 and 83.

[98] W. Richter, C. Isci, B. Gilbert, J. Harkes, V. Bala, and M. Satyanarayanan. Agentless cloud-wide streaming of guest file system updates. In *Proceedings of the 2nd IEEE International Conference on Cloud Engineering*, IC2E'14, New York, NY, USA, 2014. IEEE. Cited on page 13.

[99] O. Rodeh. BTRFS: The Linux b-tree filesystem. Technical Report RJ10501 (ALM1207-004), IBM Research, July 2012. URL `http://domino.watson.ibm.com/library/CyberDig.nsf/papers/6E1C5B6A1B6EDD9885257A38006B6130/$File/rj10501.pdf`. Cited on pages 70, 74, 89, 100, 105 and 106.

[100] P. Ruggiero and M. A. Heckathorn. Data backup options. Technical report, US-CERT, August 2012. URL `https://www.us-cert.gov/sites/default/files/publications/data_backup_options.pdf`. Cited on page 61.

[101] S. Sanfilippo and P. Noordhuis. Redis, September 2012. URL `http://redis.io/`. Cited on pages 29 and 59.

[102] A. Sangpetch, A. Turner, and H. Kim. How to tame your VMs: an automated control system for virtualized services. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924976.1924995`. Cited on pages 49 and 50.

[103] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, New York, NY, USA, 1999. ACM. ISBN 1-58113-140-2. doi: 10.1145/319151.319159. URL `http://doi.acm.org/10.1145/319151.319159`. Cited on pages 94, 95, 100 and 106.

[104] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: a highly available file system for a distributed workstation environment. *Computers, IEEE Transactions on*, 39(4):447–459, Apr 1990. ISSN 0018-9340. doi: 10.1109/12.54838. Cited on pages 102 and 106.

[105] M. Satyanarayanan, W. Richter, G. Ammons, J. Harkes, and A. Goode. The case for content search of VM clouds. In *The First IEEE International Workshop on Emerging Applications for Cloud Computing (CloudApp 2010)*, CloudApp '10, July 2010. doi: 10.1109/COMPSACW.2010.97. Cited on page 81.

[106] M. Satyanarayanan, S. Smaldone, B. Gilbert, J. Harkes, and L. Iftode. Bringing the cloud down to earth: transient PCs everywhere. In *International Workshop on Mobile Computing and Clouds*, MobiCloud '10. Springer, 2010. Cited on page 37.

[107] Sepaton Inc. Beyond virtual tape libraries, December 2014. URL `http://www.sepaton.com/products/beyond_virtual_tape_library.php`. Cited on pages 61, 102 and 106.

[108] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2003. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1090694.1090702`. Cited on pages 93 and 96.

[109] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha. A logic of file systems. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, Berkeley, CA, USA, 2005. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251028.1251029`. Cited on pages 16 and 17.

[110] S. Smaldone, G. Wallace, and W. Hsu. Efficiently storing virtual machine backups. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'13, pages 10–10, Berkeley, CA, USA, 2013. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2534861.2534871`. Cited on page 81.

[111] Space Monkey. Space monkey | beyond the cloud, November 2014. URL `https://www.spacemonkey.com/`. Cited on page 106.

[112] Splunk Inc. Operational intelligence software - machine data collection: Splunk, March 2013. URL `http://www.splunk.com/view/splunk/SP-CAAAG57`. Cited on pages 3, 54 and 97.

[113] Splunk Inc. Splunk Storm: Cloud data analysis and log management, March 2013. URL `https://www.splunkstorm.com/`. Cited on page 97.

[114] Stelian Pop. Dump/restore utilities, June 2010. URL `http://dump.sourceforge.net/`. Cited on page 102.

[115] P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor. Virtual machine time travel using continuous data protection and checkpointing. *SIGOPS Oper. Syst. Rev.*, 42(1), Jan. 2008. ISSN 0163-5980. doi: 10.1145/1341312.1341341. URL `http://doi.acm.org/10.1145/1341312.1341341`. Cited on page 94.

[116] The Apache Software Foundation. Apache solr, December 2014. URL `http://lucene.apache.org/solr/`. Cited on page 70.

[117] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright. A nine year study of file system and storage benchmarking. *Trans. Storage*, 4(2), May 2008. ISSN 1553-3077. doi: 10.1145/1367829.1367831. URL `http://doi.acm.org/10.1145/1367829.1367831`. Cited on page 37.

[118] Tripwire. Open source Tripwire, January 2012. URL `http://sourceforge.net/projects/tripwire/`. Cited on pages 3, 17 and 97.

[119] J. Turnbull. All about auditing with Puppet. `http://puppetlabs.com/blog/all-about-auditing-with-puppet`, 2010. Cited on pages 3 and 97.

[120] Veeam. Veeam backup & replication, February 2014. URL `http://www.veeam.com/vm-backup-recovery-replication-software.html`. Cited on pages 94, 100 and 106.

[121] vmitools. vmitools - virtual machine introspection tools, December 2013. URL https://code.google.com/p/vmitools/. Cited on page 98.

[122] VMware. What's new in vmware vsphere 5.0 - storage, May 2011. URL https://www.vmware.com/files/pdf/techpaper/Whats-New-VMware-vSphere-50-Storage-Technical-Whitepaper.pdf. Cited on page 12.

[123] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2208461.2208465. Cited on pages 81, 104 and 110.

[124] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-784-4. doi: 10.1145/1655008.1655021. URL http://doi.acm.org/10.1145/1655008.1655021. Cited on pages xii and 7.

[125] Wikipedia. Usage share of operating systems, November 2014. URL http://en.wikipedia.org/wiki/Usage_share_of_operating_systems#Market_share_by_category. Cited on page 72.

[126] Wolfgang Richter. Drive backup streaming writes patch, October 2013. URL https://github.com/cmusatyalab/gammaray/blob/master/src/patches/drive-backup-streaming-writes_1-6-0.patch. Cited on page 13.

[127] Wolfgang Richter. Introspection libvirt driver for nova, September 2014. URL https://github.com/theonewolf/nova/tree/introspection-libvirt-driver. Cited on page 32.

[128] Wolfgang Richter. Introspection api support for nova client, September 2014. URL https://github.com/theonewolf/python-novaclient/tree/introspection-API-features. Cited on page 32.

[129] ZeroVM. Zerovm and openstack swift, November 2014. Cited on page 101.

[130] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043576. URL http://doi.acm.org/10.1145/2043556.2043576. Cited on page 42.

[131] X. Zhang, S. Zhang, and Z. Deng. Virtual disk monitor based on multi-core EFI. In *Proceedings of the 7th International Conference on Advanced Parallel Processing Technologies*, APPT'07, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-76836-X, 978-3-540-76836-4. URL `http://dl.acm.org/citation.cfm?id=1785246.1785258`. Cited on pages 94 and 97.

[132] Y. Zhang, Y. Gu, H. Wang, and D. Wang. Virtual-machine-based intrusion detection on file-aware block level storage. In *Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing*, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2704-3. doi: 10.1109/SBAC-PAD.2006.32. URL `http://dl.acm.org/citation.cfm?id=1173698.1174110`. Cited on pages 94 and 96.

[133] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end data integrity for file systems: A ZFS case study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855511.1855514`. Cited on pages 1, 70, 74 and 100.

[134] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1364813.1364831`. Cited on pages 103 and 106.

[135] zmanda. Amanda network backup, July 2014. URL `http://www.amanda.org/`. Cited on page 102.