2014 IEEE International Conference on Cloud Engineering

Agentless Cloud-wide Streaming of Guest File System Updates

Wolfgang Richter*, Canturk Isci†, Benjamin Gilbert*, Jan Harkes*, Vasanth Bala†, Mahadev Satyanarayanan*

*School of Computer Science
Carnegie Mellon University, Pittsburgh, PA
Email: {wolf, bgilbert, jaharkes, satya}@cs.cmu.edu

†IBM T.J. Watson Research Center
Yorktown Heights, NY
Email: {canturk, vas} @us.ibm.com

***Abstract*—We propose a non-intrusive approach for monitoring virtual machines (VMs) in the cloud. At the core of this approach is a mechanism for selective real-time monitoring of guest file updates within VM instances. This mechanism is *agentless,* requiring no guest VM support. It has low virtual I/O overhead, low latency for emitting file updates, and a scalable design. Its central design principle is *distributed streaming of file updates inferred from introspected disk sector writes.* The mechanism, called DS-VMI, enables many system administration tasks that involve monitoring files to be performed outside VMs.**

I. Introduction

The opaque wall of a VM cleanly separates its host and guest environments. It isolates each guest from others; it shields the host from misbehaving guests; it supports safe and guest-transparent multi-tenancy, consolidation and migration; it ensures simple physical-to-virtual transformations. VM opacity is the foundation of trust in public clouds.

But complete opacity can become an unnecessary hindrance in a private cloud operated by a single enterprise, because of the higher level of trust between tenants. In this context, there is an incentive to centralize monitoring services such as virus scanning and software auditing. This would reduce the software management burden on employees, strengthen enterprise control, and improve compliance with corporate policies. Providing these services in a non-disruptive manner, using standardized administrative interfaces that avoid introducing agents into VMs, would be especially valuable.

An *agentless* approach to centralizing cloud services has many benefits. For example, security-sensitive agents that reside within guests today are rendered useless when their guest is compromised. If placement of those agents outside the guest were feasible, it would enhance their value. A second example relates to log file monitoring, which is often the only source of troubleshooting insights in a production system [6, 11, 13]. Today, application-specific log file updates are not visible outside their VM instance without an in-guest agent or distributed file system. The market for such agents and their associated "monitoring-as-a-service" capabilities [25, 36] exceeded $1 billion as of 2013 [10]. Agentless monitoring of application-specific log files would simplify the implementation and operation of such services. A third example involves proactive configuration and compliance auditing. Today, these involve periodic scans that delay detection and are vulnerable to guest compromise. An agentless approach could provide rapid and reliable cloud-wide detection of misconfigurations.

In this paper, we show how the opaque VM boundary can be safely and efficiently pierced to allow scalable real-time observation of guest file system updates. Our approach is based on a novel *distributed streaming VM introspection* technique that can *infer* file system modifications from sector-level disk updates in real-time and efficiently *stream* them to centralized monitors. This technique operates completely outside VM instances, and does not require paravirtualization support, guest modifications, or specific guest configuration.

Our experiments confirm that the performance overhead of this approach is modest, except for extreme write-intensive workloads. For those extreme cases, we provide a visibility-overhead tradeoff that is under the control of a cloud operator. Using this control, overhead can be limited by trading off the coverage and granularity of the tracked guest file system updates. Our results confirm the efficacy of this tradeoff in meeting desired performance goals.

This paper makes the following contributions:

- It highlights the value of agentless streaming of guest file system updates and shows how this capability can be efficiently realized through a new distributed, streaming variant of classic VM introspection (VMI).
- It shows how to bridge the semantic gap between virtual disk updates and guest file system updates in near real-time while preserving eventual consistency.
- It presents the design, implementation and evaluation of an experimental prototype called *Gamma-Ray* that implements agentless streaming of updates for `ext4` in Linux guests and NTFS in Windows guests.
- It demonstrates the value of distributed, streaming introspection with two interfaces: (1) `cloud-inotify`, a selective publish-subscribe file update monitoring framework; and (2) `/cloud`, an externally-mountable, real-time view of guest file systems. These interfaces are built using Gamma-Ray and based on real use-case scenarios.

II. Distributed Streaming VMI

In-VM agent-based systems have drawbacks in environments where trust between the guest and its host is assumed: (1) they do not provide a secure view of guest state (the guest cannot be trusted when compromised); (2) they are often difficult to configure, OS-specific, and require maintenance (inside hundreds to thousands of VM instances); and (3) they consume more resources than is necessary because they are unaware of each other (n in-VM agents vs. one centralized instance). To address these issues, we propose moving file-level monitoring tasks outside the VM guest environment and into the managed cloud infrastructure. Our solution is *distributed streaming virtual machine introspection* (DS-VMI).

Our approach is based on the fact that virtual disks are emulated hardware. Hence, every disk sector write already passes through the host system. We transparently clone this stream to a userspace process on the host. This minimally interferes with the running VM instances. We only handle file system updates that are flushed from the VM instances to their virtual disks because only then do sector writes occur. Updates that have not been flushed, and therefore represent dirty state in guest memory, are outside the scope of this paper.

DS-VMI resembles classic VMI [12], 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.

DS-VMI is challenging to implement for three reasons. First, there is a *semantic gap* [9] in mapping disk sector writes into file system updates. This could arise, for example, because a file system abstraction such as a directory could be spread over many sectors on a disk. Second, there is a *temporal gap* in collecting a series of seemingly-unrelated writes and coalescing their effects. For example, the creation of a new file may involve operations that are widely separated in time, and separately update the metadata and data portions of a file system on disk. Third, *bounded overhead* is an important operational requirement in a production cloud. This overhead includes host memory pressure, slowdown of guest writes, and increased network traffic. Our solutions to the first two challenges, architecting VMI for the cloud, are described in Section III. We describe the interfaces Gamma-Ray provides to applications in Section IV, and an evaluation of overhead and latency in Section V. We explain optimizations and address bounded overhead in Section VI. We finish with related work in Section VIII, and a conclusion in Section IX.

III. Design and Implementation

We have built an experimental prototype of DS-VMI, called Gamma-Ray, for the KVM hypervisor [22] using QEMU [3] for disk emulation. Via custom introspection code, Gamma-Ray supports commonly-used file systems including `ext2`, `ext3`, and `ext4` for Linux guests and `NTFS` for Windows guests. For brevity and ease of exposition, we focus our discussion on `ext4` with a single virtual disk per guest.

Gamma-Ray has a three-stage structure. The first stage is an indexing step, performed once per unique virtual disk (not needed for clones), for initializing Gamma-Ray. The other two stages are specific to the runtime of each VM instance executing in a cloud, as shown in Figure 1. We summarize these stages below, with details in Sections III-A through III-C:

1) **Crawling and indexing virtual disks. (Figure 1a)**
This stage generates indexes of file system data structures via a *Disk Crawler*. The indexes are generated live or loaded at instance launch from a central store.
2) **Capture and cloning of disk writes. (Figure 1b)**
This stage is implemented via a user-space helper process called *Async Queuer* that receives a stream of write events from a modified QEMU. Normally, it runs at the hypervisor hosting the VM for low latency.
3) **Introspection and translation. (Figure 1c)**
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.

A. Crawling and Indexing Virtual Disks

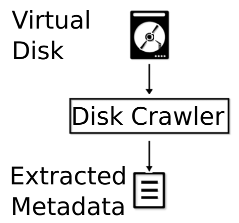
Gamma-Ray requires a one-time crawl of each virtual disk before it commences real-time streaming of subsequent file system updates. This crawl builds a map of the virtual disk for Gamma-Ray so that it can very quickly infer the file system objects being modified from incoming sector updates at runtime.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

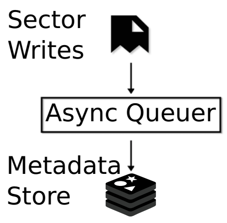
In the online case, Gamma-Ray live-attaches to an already-running VM. Here, the Disk Crawler crawls and indexes the virtual disk while it is also being modified by the executing VM. To handle the transient dirty state and not miss any new state, the dynamic component of Gamma-Ray, described in Section III-B, buffers the incoming write stream from the start of the disk crawl. Once the Disk Crawler finishes indexing, Gamma-Ray 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`, and `NTFS`. The entire disk is crawled, and serialized metadata is produced for each active partition containing a valid file system. The metadata is formatted as serialized BSON [5] 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 [16].

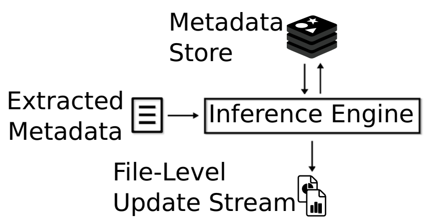
Disk analysis starts at the Master Boot Record (MBR) that contains a partition table. Each entry in this table may point to a valid primary partition or to a linked list of secondary partitions. An `ext4` partition is analyzed by first examining and serializing its superblock. The `s_last_mounted` field identifies the most recent mount point of this file system, which



(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.

Fig. 1: Three-stage DS-VMI architecture.

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 creating robust support for NTFS in Gamma-Ray.

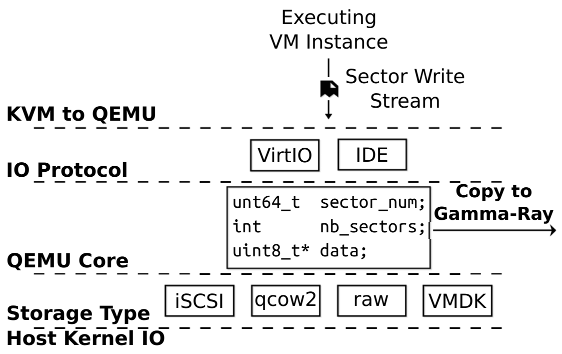
B. *Efficient Cloning of Disk Sector Writes*

KVM sends emulated I/O to QEMU, which is a userspace emulator. We copy this write stream from QEMU to Gamma-Ray using QEMU's tracing framework. This framework is configured at runtime by providing a list of events to trace. We extended the framework to include the binary contents of disk write requests. As Figure 2 shows, our extensions are located within QEMU's core, between the layers that communicate with the guest VM and with the backing storage. This enables Gamma-Ray to work with any virtual disk format and I/O protocol supported by QEMU. The write stream is copied to a pipe by a trace event.

The other end of the pipe is connected to the *Async Queuer*, shown in Figure 1b, which collects the write events and copies them uninterpreted into an in-memory queue for further processing. 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 must empty the pipe 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.

C. *Translation to File System Updates*

The *Inference Engine* 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 (Section VI-B). The Metadata Store queues sector writes awaiting translation, and also stores metadata in translation tables for fast lookup. We use Redis [30], an efficient in-memory key-value store, as our Metadata Store.



```
unt64_t  sector_num;
int      nb_sectors;
uint8_t* data;
```

Fig. 2: Connecting QEMU to Gamma-Ray.

Once virtual disk metadata is loaded and the Async Queuer starts copying raw write events into the Metadata Store, Gamma-Ray 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 Gamma-Ray thread on its host, started alongside the VM. Since Gamma-Ray runs as a separate multi-threaded Linux process, 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. Gamma-Ray stores and maintains metadata in a file-system agnostic format, implemented via multiple Redis keyspaces.

Given a write to an arbitrary position on disk, Gamma-Ray 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, Gamma-Ray 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, Gamma-Ray queries the `sector` keyspace. To retrieve the file pathname, Gamma-Ray queries the `path` keyspace. If any process registers interest in a path, data writes are passed on. In the case of metadata, the write is inspected by Gamma-Ray 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.

Channel	Monitors
`gs9671:test:/var/log/*`	Logs in VM instance test on host gs9671
`*:*:/var/log/*`	Logs on all VM instances on all hosts
`gs9671:*:/var/log/auth.log`	`auth.log` on all VM instances on host gs9671
`gs9671:test:/var/log/syslog`	`syslog` on VM instance test on host gs9671

TABLE I: Examples of filter specifications.

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 than disk sectors. Thus, Gamma-Ray maintains mappings at the granularity of file system blocks. In `ext4` the block size is derived from the superblock. For NTFS, block size comes from the "Boot File."

IV. Interfaces to Gamma-Ray

We now describe two very different interfaces to Gamma-Ray layered on top of the Metadata Store. The first, described in IV-A, provides a publish-subscribe model allowing applications to selectively track streams of file system updates. The second, described in IV-B, provides a POSIX file system model allowing unmodified legacy applications to leverage up-to-date insights from Gamma-Ray.

A. *Selective Monitoring:* `cloud-inotify`

`inotify` [26] 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. `inotify` has been used to implement desktop search utilities, backup programs, synchronization tools, log file monitors, and many more file-level event-driven applications. There are two key challenges to extending `inotify` to cloud computing. First, the file systems being monitored are remote rather than local. Second, instead of monitoring a single file system at a time, cloud-level monitoring requires an efficient abstraction for monitoring thousands of file systems at once.

Gamma-Ray solves both of these challenges with its `cloud-inotify` interface. `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 may take action. We call `cloud-inotify` applications *monitors*, and the `cloud-inotify` interface provides a *strongly consistent* stream of file-level updates to monitors.

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. A monitor may execute on the same host as an instance of Gamma-Ray it is receiving updates from, or it may be located on another machine. A local copy of the monitor could filter file-level updates from the virtual disk and pass on important updates to a remote monitor. Cooperating via Gamma-Ray, local monitors and remote monitors obtain a consistent view of the file update stream in real-time.

We use the publish-subscribe capability of Redis to implement Gamma-Ray channels. Gamma-Ray channel names are a combination of three components: the hostname, a VM name supplied by a user or derived from a generated UUID, and the full path of interest in the guest file system of the VM instance. A monitor connects to Redis' well-known TCP port and subscribes to channels using filters similar to those shown in Table I. The monitor then receives BSON-serialized messages relevant to its filter specification, each containing the changed metadata fields and corresponding file data. Monitors subscribe without exposing themselves to a firehose of irrelevant data. We consider two use cases below.

Log monitoring: 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. Sangpetch et al. [31] 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. Gamma-Ray can derive the same metric directly from application logs.

Auditing: Auditing file-level changes is useful for enforcing policy, monitoring for misconfigurations, and watching for intruders. Unlike agent-based solutions, Gamma-Ray cannot be turned off, tampered with, or misconfigured by guests. Centrally-managed auditing ensures that all VMs are checked for the most recent security updates and best practices. 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 [8] reported an outage in early 2011 that affected 15-20% of its production fleet. The root cause was a permissions change to a folder in the path to the dynamic loader. Google found troubleshooting difficult because logging into affected servers was impossible. Gamma-Ray does not depend on guest health; thus, cloud customers never "fly blind" even if they cannot access instances.

B. *Synthetic File System:* `/cloud`

There are some applications for which an event-driven model is inappropriate. An example is querying arbitrary system state such as a log entry from several days ago on a long-running VM instance. Other examples include scanning a VM instance for a newly-discovered vulnerability, or checking for a newly-discovered misconfiguration. To support such use cases, we provide a complementary interface into Gamma-Ray — a file system interface called `/cloud`. Because we store metadata in a normalized format, we implement `/cloud` in a single FUSE driver. Our driver offers a read-only view into the file systems of any VM instance in a cloud.

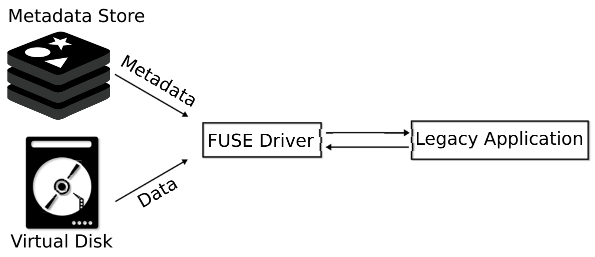


Fig. 3: /cloud implementation.

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 3. Metadata versions guarantee consistency for metadata, but not data. Thus, `/cloud` provides an *eventually consistent* file system view of files within executing VMs.

Our read-only file system view 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. 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`, or standard log monitoring applications such as Splunk [35], to quickly search, or monitor, subsets of VMs without agents inside those VMs.

Log Monitoring: In the log monitoring example from Section IV-A, we assumed the insights we want from log files are already known. In some cases, such as an investigation following a security breach, we need to re-crawl logs looking for evidence. Failed password-based `ssh` logins are normally indicative of break-in attempts. Using `/cloud` and `grep` we can quickly scan recent logs across all instances to find password-based failed logins: `grep "Failed password" /cloud/*/*/var/log/auth.log`.

Auditing: In the Google example from Section IV-A, we assumed operators could be notified nearly instantaneously about misconfigured permission bits. Of course, this can only occur if they are already being monitored. With `/cloud` using familiar commands such as `find`, one can check permissions across the cloud: `find /cloud/*/*/lib -maxdepth 0 -not -perm 755`.

V. Evaluation

In the following sections, we seek to answer the following questions about Gamma-Ray:

Sec. V-D: *How close to real-time is Gamma-Ray?*

Sec. V-E: *What is the overhead of crawling for, transferring, and loading metadata?*

Sec. V-F: *How much slowdown does Gamma-Ray cause in a running guest?*

Sec. V-G: *What is the memory footprint of Gamma-Ray?*

A. 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 Ubuntu 12.04 LTS AMD64 Server. We base all of our work on QEMU release 1.6.0. We use Redis server version 2.2.12 and `libhiredis` version 0.10.1. For BSON, we use our own custom implementation in C. Each host has two hard drives: a primary 250 GB drive (Seagate ST3250310AS) and a secondary 1.5 TB drive (Seagate ST31500341A). The secondary drive was used to write and store log files, and the primary drive hosted the virtual disks. This setup minimized I/O contention while collecting results from experiments. Unless otherwise stated, timing experiments were run 20 times and both the average and standard deviation are reported.

When running a VM, we follow IBM's KVM best practices [18]. 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 async I/O backend 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 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, 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, we reduce the risk of invalid timing data due to unreliable VM clocks.

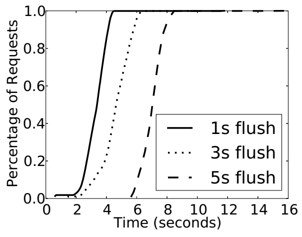
We used a single VM guest pre-loaded with all software. The guest runs Ubuntu 12.04 LTS AMD64 Server, with 1 CPU, 1 GB RAM, 20 GB disk, and a single partition containing an `ext4` file system with default file system options and 2.6 GB of used space. By comparison, in a survey of 30 Amazon EC2 cloud images, we found an average root disk size of 18 gigabytes, a maximum of 80, and a minimum of 5.

B. Write Intensive Benchmarks

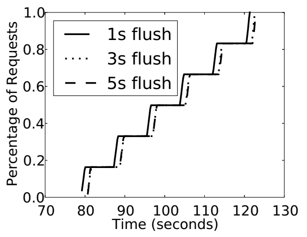
bonnie++ [7] (4.1 GB written) is a microbenchmark tool designed to measure the overhead of various file system operations such as create, delete, write, and read. We used its default settings.

PostMark [21] (2.9 GB written) is a well-known benchmark designed to simulate mail server disk I/O. We used it with a configuration suggested by [37]: file size $[512, 328072]$, read size 4096, write size 4096, number of files 5000, number of transactions $20,000$.

Modified Andrew Benchmark [17] (277 MB written) is another well-known benchmark. In the `MakeDir` phase, our modified version creates a directory tree mirroring the `linux-3.5.4` kernel tree [24]. 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



(a) Sync every second



(b) No extra syncs

Fig. 4: Latency CDFs.

in the tree. Finally, the `Make` phase compiles the Linux kernel using a configuration provided by `make defconfig`.

Software Install (1.9 GB written), inspired by a benchmark used in [32], 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, Java application servers, and many others.

C. Gamma-Ray Tunables

In our experiments, the most critical parameters are the "Async Flush Timeout" (default 5 seconds) and "Async Queue Size Limit" (default 250 MB). The flush timeout helps bound the maximum latency from a disk write to an emitted inferred file-level event on a channel. The queue size limit bounds the amount of memory that the async queue process may consume. The outstanding write limit (default $16,384$) bounds the number of writes queued for the inference engine, preventing it from falling too far behind the guest VM. The "Unknown Write TTL" (default 300 seconds) defends against denial of service: if a guest writes a large amount of data, but never associates it with live files, the data is eventually discarded.

D. Latency for File Monitoring Tasks

To measure latency, we designed a microbenchmark (6.4 MB written) to trace writes from a guest through Gamma-Ray to a monitoring application. We used an HTTP client to construct GET requests with headers containing a timestamp from the host. These requests went to an in-guest Apache web server serving a static file and were logged by Apache. We sampled the log file every second via our `/cloud` interface. When a new log entry appeared, we timestamped it with the host's time and saved it separately.

Figure 4 shows the results of $10,000$ requests during the microbenchmark. Figure 4a shows the best case when the guest frequently syncs data to disk with an additional latency of 1, 3, or 5 seconds on average. Figure 4b shows an untuned guest where latency is at the mercy of guest kernel I/O algorithms which flush at much lower frequency than Gamma-Ray's default. The step-like nature is because many updates appear at once — many log file lines fit inside a single file system block. Akamai's system [6] would work on Gamma-Ray without any tuning of guests, but low latency performance monitoring [31] may require tuning of guest flush algorithms.

E. Crawling and Bootstrapping

Table II shows how metadata grows as a function of used disk space for `ext4` and NTFS. We increased used disk space

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 II: Metadata as a function of used virtual disk space (20 runs, std. dev. in parentheses). Used is used disk space reported by `df`, Raw is uncompressed metadata, gzip is compressed metadata with `gzip --best`, Crawl is the indexing time, and Load is the load time into Redis.

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 III: Metadata as a function of number of inodes (20 runs, std. dev. in parentheses). The headers are the same as in Table II except the first column is a unitless count of live inodes in the file system rather than 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 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 more compact and quicker to scan. 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. Load times are masked by concurrently loading metadata while the virtual disk is transferring over the network to its host system. Crawl is a one-time operation, amortized over the life of a virtual disk.

Table III 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).

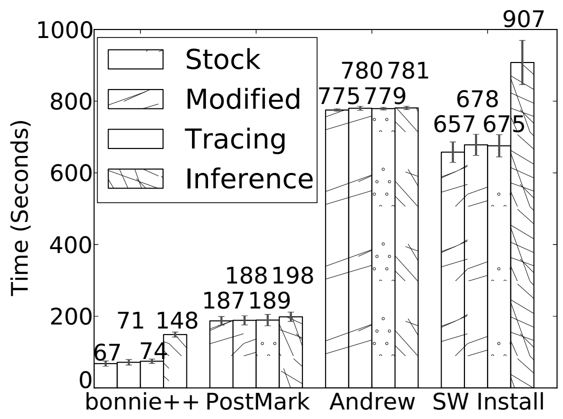


Fig. 5: Application benchmarks: Stock QEMU is without modification, Modified is with instrumentation compiled in, Tracing is with it turned on and sent to `/dev/null`, and Inference is full system (20 runs each).

Paths are stored at least twice: once as a full path associated with a file, and once as a directory entry. This accounts for the approximately 20-megabyte increase in metadata size for `ext4` with a large number of files.

F. VM Slowdown

Figure 5 shows the time to complete, on average, all four of our main benchmarking applications. Gamma-Ray shows overhead in three of the four write benchmarks: bonnie++, PostMark, and Software Install. We observed an average overhead of 15% across the more practical application benchmarks, and a worst case of 120% for the bonnie++ microbenchmark. This subsection explores why each benchmark has overhead, and we show how to mitigate overhead in Section VI.

Clustered Large Writes. A breakdown of memory usage, I/O pattern, and async queue flushes for bonnie++ is the first row in Figure 6. The inference engine shows very little memory usage. The memory usage of the Async Queuer, however, repeatedly spikes up and down. This occurs because bonnie++ is a write-intensive microbenchmark, and the Async Queuer, in this case, is regularly hitting its configured queue size limit of 250MB and flushing to Redis. If flushes were only triggered by the 5-second writeback timer, this implies a maximum of 40 flushes (200 second run). However, the right-most graph shows 53 flushes, 13 triggered by the 250 MB ceiling. The middle graph confirms: this experiment was the most write-intensive. Its closely-clustered, intense write pattern causes the performance degradation.

Small Clustered Writes. PostMark shows similar behavior as bonnie++; however, its write pattern is more dispersed. Column 2, row 2 of Figure 6 shows that PostMark has many smaller clusters of writes. Its async 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.

Low Volume Small Writes. The Modified Andrew Benchmark shows negligible overhead when Gamma-Ray is introspecting disk writes. This is because the write traffic was not sufficient to fill the async queue, and when flushed due to timeout, the write volume was small enough to avoid any performance degradation. The write pattern shown in row 3, column 2 of Figure 6 demonstrates less intense write clustering compared to PostMark and bonnie++. The Modified Andrew Benchmark had the fewest writes: $5,293$.

High Volume Writes. The Software Install benchmark has the largest number of writes: $61,694$. 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 async queue memory. Even though it does not trigger extra async queue flushes, however, it is interrupted too frequently by timer-based flushing. This effect was sufficient to significantly slow it down.

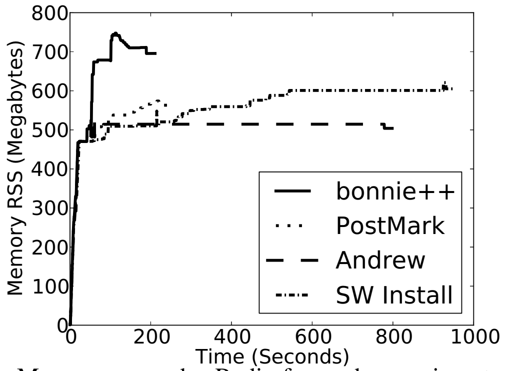


Fig. 7: Memory usage by Redis for each experiment. Only a single run was examined.

Experiment	Async Q. (MB)	Inf. Eng. (MB)	w/ Redis (MB)
bonnie++	250	49	1043
Andrew	88	9	630
PostMark	214	27	739
SW Install	81	26	708

TABLE IV: Peak memory usage of the Async Queuer, inference engine, and Redis combined.

G. Memory Footprint

Figure 7 shows memory used by Redis during each of the four experiments, and Table IV 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 combined overhead is 15% of the used disk space of 2.6 GB.

VI. Optimizations Bounding Overhead

In the following sections, we discuss two optimizations which reduce (1) write overhead, by relaxing completeness (Section VI-A); and (2) memory footprint (Section VI-B). These optimizations let an operator configure the performance-latency-completeness tradeoff inherent to DS-VMI.

A. Overhead vs Completeness Tradeoff

Intense file system activity within a VM instance results in high transient overhead for inference and streaming, as

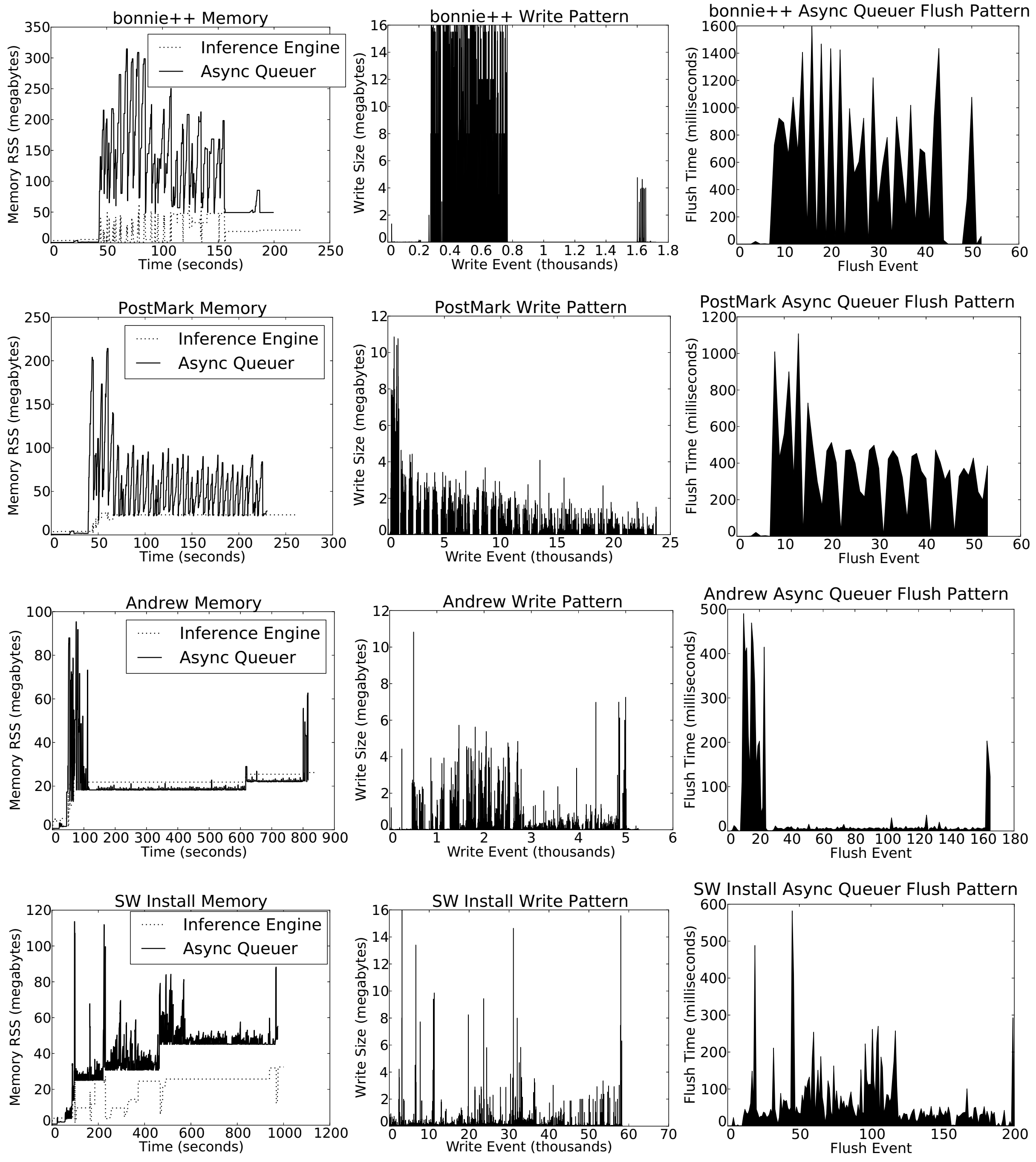


Fig. 6: Rows represent an individual benchmark, and columns represent type of data: the first column shows memory overhead, the second column shows the spread of I/O writes, and the third column shows async queue flushes. All of these data were gathered from a single run of each experiment. (Averaging different runs is difficult due to timing differences and non-deterministic write reordering, although emergent patterns are present in all runs.) Memory data was gathered by polling `/proc/PID/status` every 0.25 seconds.

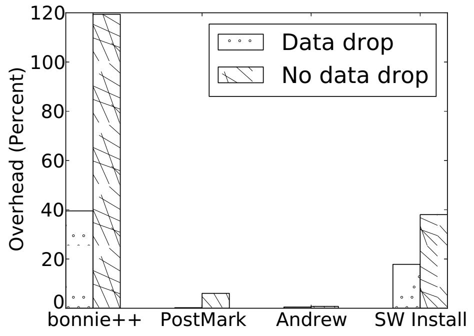


Fig. 8: Effect of dropping data writes optimization.

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 V: Lazy loading optimization effect on memory.

evidenced in the bonnie++ benchmark in Section V. We avoid this by temporarily suppressing observation of the hyperactive parts of the file tree within that instance while continuing to fully observe file system activity within other VM instances. After the intense burst of file system activity passes, full observation resumes. We describe the most performant policy for relaxing completeness implemented by Gamma-Ray.

Data Drop: 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. If one dropped only data writes, but still processed metadata writes, one's view of the guest file systems would not randomly decay. The only loss of information occurs when blocks transition from data to metadata which means they may be prematurely dropped. For example, directories are often implemented as files with data streams that list file entries. In this case, a disk introspection system would be able to detect directories being created, but not the listing of files within them. It is possible to catch up if the guest writes to the transitioned blocks a second time.

Figure 8 shows all of the benchmarks from Section V with data dropping enabled. All of the benchmarks with significant write overhead show improvement. The application-based benchmarks show a worst case overhead of bonnie++ is reduced to 39.5%.

B. Lazy Loading of File System Metadata

In Section V-G we found that memory overhead is 15% of used disk space, when Redis is pre-populated with metadata for all files in the guest file system. However, during benchmarks, as well as in expected day-to-day 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 V. 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.

VII. Considerations and Future Work

Guest File System Compatibility

To guarantee the correctness of our introspection, the file system semantics must ensure that incorrect inferences cannot occur. Such a file system must exhibit "a strong form of reuse ordering," as well as metadata consistency [34]. *Strong reuse ordering* means that the file system must commit the freed state of any sector and its allocation data structures to disk before reuse, and *metadata consistency* means maintaining all file system metadata with a set of invariants that ensure correct operation. Most modern file systems meet these requirements.

Ongoing and Future Work

(a) Storage and VMM Technologies. We are currently working on support for additional storage technologies including LVM and `btrfs`. In parallel with these, we are also investigating generalizing Gamma-Ray operation to other hypervisors, in particular, Xen and VMware ESX. Due to its design, Gamma-Ray itself does not require significant changes; it only needs access to the virtual disk's write stream.

(b) Encrypted File Systems or Full Disk Encryption. If the VM's disk is encrypted, Gamma-Ray's ability to provide a meaningful service is severely curtailed. However, if the owner of the VM was willing to share encryption keys, Gamma-Ray could be extended to decrypt file system writes as they occurred. In addition, if only file-level encryption was employed, Gamma-Ray could still report data updates for unencrypted files and metadata updates for all files.

(c) Privacy Considerations. We envision a fine-grain access control policy, with configuration options allowing visibility only to the relevant parts of the guest file system. This limited visibility can also be enforced by the end user via partial disk encryption. Trusting Gamma-Ray is no different from trusting an agent from one of many vendors.

VIII. Related Work

DS-VMI provides a unique set of features not seen together before: it maintains full-fidelity update streams, it is not limited by backing storage type, it requires no in-guest support, it does not require paravirtualization, it generalizes across closed- and open-source kernels, and it directly leverages hierarchy present in the organization of modern virtualized clouds.

The storage community [1, 14, 23, 27] provides performant solutions for snapshotting which could be polled for file updates. The smart disk community [2, 33] provides methods of semantically understanding file systems and file-level updates to increase performance via intelligent prefetching and reorganizing sector layout on disk. The VMI community [4, 9, 15, 19, 20, 28, 38, 39] provides techniques for understanding the disk write stream.

Olive [1], Lithium [14], and Petal [23] create snapshots within hundreds of milliseconds and incur low I/O overhead. Parallax [27] was explicitly designed to support "frequent, low-overhead snapshot[s] of virtual disks." 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. However, information is lost in between snapshots.

Semantically-smart disk systems (SDS) [33] interpret metadata and the type of a sector on disk as well as associations between sectors, but do not support distributed streaming and would require guest support. IDStor [2] implements inference for disk sector writes for iSCSI network storage with the `ext3` file system. Zhang et al. [39] describe a VM-based approach that leverages smart disk technology.

Garfinkel and Rosenblum [12] coin the term *virtual machine introspection* and develop an architecture focusing on analyzing memory. Formally, DS-VMI is an *out-of-band* introspection method [29]. XenAccess [28] introspects both memory and disk, but only infers file creations and deletions. Zhang et al. [38] introspect disk, but for enforcing access control rules in the critical I/O path. VMWatcher [20] interprets memory and disk operations, but requires kernel source. VMScope [19] captures events such as system calls, but does not interpret virtual disk writes. Virtuoso [9] automatically generates introspection tools, but not for disk operations. Hildebrand et al. [15] describe a method of performing disk introspection to the point of identifying disk sectors as metadata or data. Maitland [4] is a system that performs lightweight VMI for cloud computing via paravirtualization.

IX. Conclusion

We proposed DS-VMI, an agentless approach independent of a VM guest's OS, to provide external applications near-real-time visibility into the file-level updates of a running VM guest. We enumerated an explicit tradeoff between performance, completeness, and latency and found that, for practical applications, low latency, low overhead, and bounded resource usage are all achievable. Using DS-VMI, we implemented two valuable interfaces for cloud-wide applications: (1) `cloud-inotify` for selective streaming of updates, and (2) `/cloud` for a read-only up-to-date file-system view of running instances. These two interfaces enable efficient implementation of cloud-wide file-level monitoring applications, such as log monitoring, without any guest support.

Gamma-Ray is released open source under the Apache 2.0 License at https://github.com/cmusatyalab/gammaray.

Acknowledgements

We thank Debjani Biswas, Kiryong Ha, Yoshihisa Abe, Zhuo Chen, and our anonymous reviewers for their valuable feedback. This research was supported by the National Science Foundation (NSF) under grant numbers CNS-0833882 and IIS-1065336, by an Intel Science and Technology Center grant, by DARPA Contract No. FA8650-11-C-7190, and by the Department of Defense (DoD) under Contract No. FA8721-05-C-0003 for the operation of the Software Engineering Institute (SEI), a federally funded research and development center. This material has been approved for public release and unlimited distribution (DM-0000276). Wolfgang Richter was partly supported by an IBM Graduate Fellowship, an NSF Graduate Fellowship, and additional support was provided by IBM, Google, Bosch, and Vodafone. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and should not be attributed to their employers or funding sources.

References

[1] M. K. Aguilera, S. Spence, and A. Veitch. Olive: distributed point-in-time branching storage for real systems. NSDI'06. USENIX Association, 2006.

[2] M. Allalouf, M. Ben-Yehuda, J. Satran, and I. Segall. Block storage listener for detecting file-level intrusions. MSST'10, May 2010.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. ATEC '05. USENIX Association, 2005.

[4] C. Benninger, S. Neville, Y. Yazir, C. Matthews, and Y. Coady. Maitland: Lighter-weight VM introspection to support cyber-security in the cloud. CLOUD'12, June 2012.

[5] BSON. BSON, September 2012. URL http://bsonspec.org/.

[6] J. Cohen, T. Repantis, S. McDermott, S. Smith, and J. Wein. Keeping track of 70,000+ servers: the Akamai query system. LISA'10. USENIX Association, 2010.

[7] R. Coker. bonnie++, 2001. URL http://www.coker.com.au/bonnie++/.

[8] C. Colohan. The "Scariest Outage Ever". URL http://www.pdl.cmu.edu/SDI/2012/083012b.html. SDI/ISTC Seminar Series, 2012.

[9] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. SP'11, May 2011.

[10] Frost & Sullivan. Analysis of the SIEM and log management market. Technical report, Frost & Sullivan, http://goo.gl/Vup9ml, November 2013.

[11] H. Garcia-Molina, F. Germano, and W. H. Kohler. Debugging a distributed computing system. *IEEE Transactions on Software Engineering*, SE-10(2), March 1984.

[12] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. NDSS'03, 2003.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. SOSP '03. ACM, 2003.

[14] J. G. Hansen and E. Jul. Lithium: virtual machine storage for the cloud. SoCC '10. ACM, 2010.

[15] D. Hildebrand, R. Tewari, and V. Tarasov. Disk image introspection for storage systems, US Patent Pending 2011.

[16] T. Hoff. 7 years of YouTube scalability lessons in 30 minutes, March 2012. URL http://goo.gl/DccW5.

[17] 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.

[18] IBM Corporation. Best practices for KVM, April 2012. URL http://goo.gl/XXS8W.

[19] X. Jiang and X. Wang. "Out-of-the-box" monitoring of VM-based high-interaction honeypots. RAID'07. Springer-Verlag, 2007.

[20] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. CCS '07. ACM, 2007.

[21] J. Katcher. PostMark: A new file system benchmark. Technical report, NetApp, http://goo.gl/xbTMD, 1997.

[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. Ottawa Linux Symposium, July 2007.

[23] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. ASPLOS-VII. ACM, 1996.

[24] Linux Kernel Organization, Inc. Linux kernel archive, September 2012. URL http://www.kernel.org/.

[25] Loggly Inc. Log management service in the cloud - Loggly, March 2013. URL http://loggly.com/.

[26] J. McCutchan. [RFC][PATCH] inotify 0.8, 2004. URL http://goo.gl/4Ez5S.

[27] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. Eurosys '08. ACM, 2008.

[28] B. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. ACSAC'07, December 2007.

[29] J. Pfoh, C. Schneider, and C. Eckert. A formal model for virtual machine introspection. VMSec '09. ACM, 2009.

[30] S. Sanfilippo and P. Noordhuis. Redis, September 2012. URL http://redis.io/.

[31] A. Sangpetch, A. Turner, and H. Kim. How to tame your VMs: an automated control system for virtualized services. LISA'10. USENIX Association, 2010.

[32] M. Satyanarayanan, S. Smaldone, B. Gilbert, J. Harkes, and L. Iftode. Bringing the cloud down to earth: transient PCs everywhere. MobiCloud '10. Springer, 2010.

[33] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. USENIX Association, 2003.

[34] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha. A logic of file systems. FAST'05. USENIX Association, 2005.

[35] Splunk Inc. Operational intelligence software - machine data collection: Splunk, March 2013. URL http://goo.gl/XgjJM.

[36] Splunk Inc. Splunk storm: Cloud data analysis and log management, March 2013. URL https://www.splunkstorm.com/.

[37] 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.

[38] X. Zhang, S. Zhang, and Z. Deng. Virtual disk monitor based on multi-core EFI. APPT'07. Springer-Verlag, 2007.

[39] Y. Zhang, Y. Gu, H. Wang, and D. Wang. Virtual-machine-based intrusion detection on file-aware block level storage. SBAC-PAD'06. IEEE Computer Society, 2006.