Near-Real-Time Inference of File-Level Mutations from Virtual Disk Writes

Wolfgang Richter, Mahadev Satyanarayanan,
Jan Harkes, Benjamin Gilbert

February 2012
CMU-CS-12-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We describe a new mechanism for cloud computing enabling near-real-time monitoring of virtual disk write streams across an entire cloud. Our solution has low IO overhead for the guest VM, low latency to file-level mutation notification, and a layered design for scalability. We achieve low IO overhead by duplicating the virtual disk write stream as it passes through a managing VMM. We achieve low latency by performing semantic inference at as high a level as possible—file-level. We achieve cloud scale by layering our design allowing filtering of file-level mutations by each layer such that network traffic to centralized monitoring infrastructure is minimized. We assume this technique is used on pre-indexed virtual disks, most likely derived from a cooperating VM image library such as those used in clouds today. Our new cloud primitive enables system administration tasks that involve monitoring files—virus scanning, log file parsing, etc.—to be performed outside of the running VM instance, either on the VMM host, or shipped to a central monitoring agent.

This research was supported by the National Science Foundation (NSF) under grant number CNS-0833882, an NSF Graduate Research Fellowship, an IBM Open Collaborative Research grant, and an Intel Science and Technology Center grant. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily represent the views of the NSF, IBM, Intel, or Carnegie Mellon University.

Keywords: block write, cloud, cloud computing, file-level, inference, introspection, kernel virtual machine, KVM, monitoring, near-real-time, real-time, semantic, virtual disk, virtual disk write, virtual machine, VM, virtual machine introspection, VMI, virtual machine monitor, VMM

1 Background and Problem Statement

Cloud computing centralizes the storage of virtual disks along with their associated snapshots transforming historic virtual machine (VM) disk state into big data [22]. VM retrospection [23] takes advantage of this newly available historic state and it is the mechanism for searching across frozen virtual disks and their sequences of disk snapshots at a file-level. VM retrospection is used to find prior good configurations of VM state among many other use cases, but the snapshotting techniques retrospection relies on for inspecting the past do not provide a real-time view of the cloud. For example, an instance owner can retrospect their instance's historic file data scanning for viruses, but they are limited by the frequency of snapshotting and the time needed for retrieval of file-level data from snapshots. They know that as of yesterday's snapshot their instance was free of any virus; however, a compromise can happen within seconds and attackers could have control of the instance today. The problem is that snapshots provide a *quantized* view of disk state changes over time, which limits observable events to the frequency of snapshotting.

What we need is a mechanism transforming executing VM instances' *continuous* virtual disk write streams into big data. Waiting for a snapshot and understanding it at a semantic level causes significant lag in retrieving file-level data from executing VM instances. There are two possible paths to solving this problem: (1) VM instances could cooperate by reporting file-level mutations—file creation, file deletion, file modification—as they occur in near-real-time, or (2) the cloud could provide a mechanism for obtaining them in near-real-time. Although the path of least resistance technically, the first path leads to unhappy customers for both public and private clouds. The cloud forces them to run monitoring software within their instances which steals their CPU cycles, memory, and network bandwidth. Refraining from stealing complicates billing for cloud operators because it is difficult to separate CPU, memory, and network bandwidth between what the instance owner uses, and what the monitoring software uses. Thus, we choose the second path: we obtain file-level mutations in near-real-time without the monitored instance's cooperation.

The second path requires a new mechanism for cloud computing, one that provides a stream of near-real-time file-level mutations as they occur while VM instances execute. A naïve solution approximates capturing disk state in near-real-time by snapshotting at higher frequency. However, current state-of-the-art snapshotting techniques suffer from at least one of two possible issues, and most suffer from both. The first issue is the reason why VM instances are not snapshotted in practice at high frequency today: snapshotting is a *heavyweight* operation that degrades the IO performance of the VM instance. The second issue stems from the black box nature of snapshots. The most common snapshotting techniques result in a collection of disk writes—à la copy-on-write snapshotting, an opaque binary delta, or an entire virtual disk. None of these forms efficiently map to file-level mutations. Understanding what file-level mutations occurred between two consecutive snapshots requires mounting their file systems for a file-level view. One must replay the collection of disk writes, apply the delta to an original, or directly mount the whole disk snapshot. Once mounted, one must crawl the entire file system searching for mutations at a file-level. We have encountered the second issue with current state-of-the-art snapshotting techniques: high latency notification of file-level mutations.

Can we create a new *lightweight* mechanism for cloud computing that has minimal IO overhead and also minimizes the notification latency for file-level mutations, while maintaining or exceeding the fidelity of prior state-of-the-art snapshotting techniques? Can we obtain our desired real-time view of the cloud by transforming executing VM instances' disks into big data?

2 Solution Strategy

The essence of our idea is to generate a stream of inferred file-level mutations from the virtual disk write stream of an executing guest VM. Our approach builds upon techniques pioneered in virtual machine introspection (VMI) research. VMI traditionally infers file-level mutations synchronously by slowing the guest for analysis in real-time. To achieve high IO performance, our approach is asynchronous which lags real-time, but we maintain a near-real-time stream of file-level mutations. To achieve scale, we envision a layered architecture allowing filtering of file-level mutations at each layer. In addition we provide a fidelity of state change that is, at a minimum, as good as, and, in some cases, better than state-of-the-art snapshotting. Our novelty lies in adapting VMI to cloud-scale by achieving three goals: (1) low IO overhead for VM guests, (2) low latency for file-level mutation notifications, and (3) scalability.

We offer better fidelity than snapshotting because our capture of file-level mutations occurs continuously as a stream of inferred mutations, whereas snapshotting occurs at distinct points in time. As mentioned in §1, snapshotting provides a quantized view of disk state over time. There are events between snapshots that are lost—a file created and deleted—that our stream contains, but no snapshot has any record of ever occurring. Creating a stream of inferred file-level mutations requires duplicating virtual disk block writes to an external process we call *inference engine* as described in §3. We only require capture of disk block writes and do not trap system calls or interpret executing guest instructions. We explore many applications in §4 generalizing beyond the normal scenarios where snapshotting is usually applied. We consider continuous inference at the virtual disk level within the cloud and have explicit support for streaming file-level mutations inferred by the inference engine over a network for centralized analytics in near-real-time—centralized virus scanning, centralized anomaly detection, centralized configuration change detection, etc. The rest of this section discusses VMI (§2.1), compares and contrasts our method with snapshotting (§2.2), and outlines the prerequisites needed for a file system or application to work with our method (§2.3).

2.1 Virtual Machine Introspection

The VMI [8] research community unwittingly invented a solution to the problem of snapshot latency. VMI is a method of introspecting executing virtual machine state at a semantic level used historically for intrusion detection and other security applications. Attackers compromising a virtual machine can not hide from intrusion detection systems based on VMI because of the strong isolation provided by the virtual machine monitor (VMM) between VMI applications and the running VM. Techniques for disk-based VMI take advantage of the fact that every disk block write must traverse through the VMM, because disks are emulated hardware. Thus, the disk block write stream already passes under the inspection of the VMM and VMI simply adds more layers of inspection.

However, the historic setting of VMI—security—has not focused on the IO performance of the guest VM. Because they do not focus on maximizing IO performance, disk-based VMI techniques often interpose on the critical IO path examining each block write. On top of this interposition, intrusion detection systems check each write's file-level interpretation against a database of rules—synchronously—and allow, disallow, or report the write depending on current VM state and the set of active rules. This synchronous nature, although lowering the latency for file-level mutation

notification, causes IO slowdown in the guest VM, with one system reporting 33% [32] overhead. Thus, VMI offers a partial solution, but it is not a panacea because the first issue we need to address, high IO overhead, remains.

Cloud computing adds another dimension of complexity: scale. VMI is traditionally implemented in a single system design without considering scale. Asynchrony solves the performance issue that a single VM guest observes on a single system, but it is not enough to scale to tens of thousands of running VM instances. Centralized virtual disk write monitoring across an entire cloud requires a scalable architecture. To achieve scale we add layers to our approach, *filtering* file-level mutations as early as possible to reduce the flow of traffic over networks to centralized monitoring points. A layered approach creates flexibility by allowing selective monitoring of file-level mutations by higher layers. Of course, higher layers must also be designed with scale in mind, and the central monitoring point must scale with the size of the cloud. We focus on scaling by minimizing communication via filtering, and distributing inference computation.

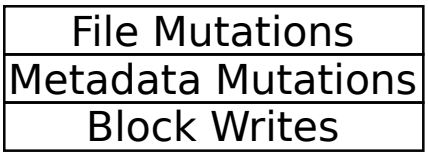

Figure 1: Three levels of abstraction in capturing state from a disk. Inferring file-level mutations is the highest level.

Figure 1 shows a spectrum of possible block write introspection, with three points in the semantic space highlighted as a stack. Considering introspection at each of the three highlighted points reveals tradeoffs for our three constraints—we desire low IO overhead, low latency file-level mutation notification, and scalability in a cloud setting. We could replicate the virtual disk block writes without inferring any higher level activity, corresponding to the bottom of Figure 1. This is equivalent to block replicated storage devices. Simple block write replication has low IO overhead, but it has high latency to file-level mutations because the block writes must be interpreted, and it does not scale well because it blindly copies all disk write traffic. We could track metadata providing metrics such as space used by the file systems, depth of paths, number of files, etc. corresponding to the middle of Figure 1. Tracking metadata has low IO overhead, and also decreases the latency for file-level mutation notification; however, more interpretation is still required for understanding file-level mutations. Scalability increases because we only track metadata and do not blindly copy every disk block write. We could introspect disk block writes at the highest semantic level corresponding to the top of Figure 1 by inferring entire file-level mutations. This final form of introspection gives us low IO overhead, low latency to file-level mutation notification, and also scalability—only "interesting" file-level mutations need to be passed on as defined by higher layers. This last method minimizes network traffic to processing layers which may be centralized, and also distributes initial inference computation and filtering amongst all hosts executing guest VMs. Our method performs the last form of inference at the highest semantic level, and because of this satisfies all three constraints while also enabling more applications as discussed in §4.

2.2 Correctness Relative to Snapshotting

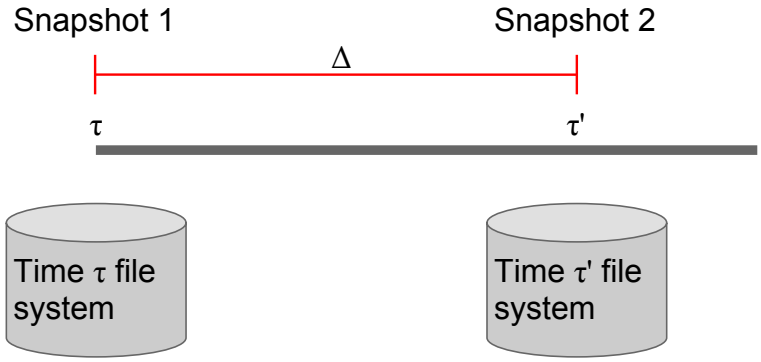

Figure 2: Combining file-level mutations recorded as Δ with the file system at time τ yields a file system equivalent to the file system at time τ'. The file systems may be mutating state on disk when the snapshots occur.

Our technique offers the same fidelity at a file-level as current state-of-the-art snapshotting methods. We never report a file-level mutation 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 mutation 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 mutations we report are consistent with file-level changes observable in that snapshot. For example, if a tool such as Tripwire [29] runs on the snapshot and runs on a virtual disk maintained with updates from a stream of file-level mutations, the results are identical. This does not mean that our technique provides the equivalent of a snapshot at a block-level.

Figure 2 superimposes our technique over snapshotting, and we use this figure to develop an example illustrating the difference between inferring file-level mutations and snapshotting. Figure 2 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, our technique provides a stream of file-level mutations in between these two time points which we refer to as Δ. Thus, Δ is a *well-ordered set* of inferred file-level mutations. Applying our stream of file-level mutations 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 mutations 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 mutations 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 inferring file-level mutations would not contain disk blocks associated with the file because its creation did not complete.

2.3 File System and Application Requirements

What file systems are amenable to this approach? Sivathanu et al. [26] derive the properties a file system must exhibit for this type of file-level inference. The guarantee a file system must maintain, from [26], is,

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

In words, the type (k) of a block (A) on disk (D)—generally 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 for a file can not be mistaken for a metadata block for the same file. For this guarantee to hold, a file system must exhibit, "a strong form of reuse ordering," and metadata consistency [26]. *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 [26] (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, any inference might incorrectly believe that their type, and their contents, have not changed upon future writes.

What class of applications are amenable to this approach? Any application that can resume or operate on data flushed to disk works with both whole disk snapshotting and file-level mutation inference. Recent research [10] reports that modern desktop applications frequently flush data to disk which means many common applications already work with snapshotting and by extension file-level mutation inference. Abstractly, a disk flush requires buffered IO 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* [26] 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 IO operations. Our technique preserves flushed file-level mutations, which means applications properly flushing critical data to disk can safely use data obtained via file-level mutation inference.

3 Architecture

In this section we begin by giving an overview of the abstract architecture and technical challenges behind inferring file-level mutations from virtual disk block writes. We describe an asynchronous architecture outside of the critical IO path that makes near-real-time inference feasible. We continue with a concrete example of how this technique works with KVM [16] and `ext2` [5]; however, KVM, a Type 2 VMM, is only an example. This technique works with both Type 1 and Type 2 VMMs. For a Type 1 VMM the technique could be implemented within the VMM or execute as an extra VM managed by the VMM. Our example with KVM illustrates a potential

implementation path for a Type 2 VMM. We finish with challenges posed by features of modern file systems and guest OS kernels.

3.1 Inferring File-Level Mutations

Figure 3 shows the basic idea of inferring file-level mutations from disk writes. Each layer is reverse mapped in order to understand the file or directory that a disk write modifies. Creations and deletions of files and directories are detected via inference based on metadata manipulations; this is file system specific and may need monitoring of a journal, inodes, or other file system data structures laid out on the disk. Each layer needs a non-trivial amount of engineering to reverse—we are essentially running the portions of a kernel devoted to handling disk writes in reverse outside of the kernel initiating those writes. We introduce the term *inference engine* to refer to the process responsible for performing this full stack reverse mapping. We also introduce the term *monitoring agent* to refer to processes that register interest in certain file-level mutations with the inference engine.

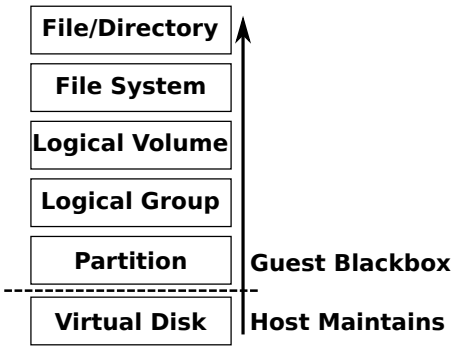

Figure 3: Reverse mapping of block writes observed by the host, to file writes in the guest.

We assume that the inference engine begins with a complete and consistent view of the virtual disk before the VM instance begins running. If the VM disk is stored in a central repository, this repository can maintain data structures for later inference. For example, it could store the partition table, positions of key inodes, position of the journal for each partition, and the metadata of LVM [28] partitions in a compact representation for initializing an inference engine at runtime.

When a VMM starts a VM guest on a host, it also starts an inference engine before the VM guest is allowed to write to its virtual disk. Once virtual disk writes begin, they are duplicated as a stream to the inference engine process running on, presumably, a separate core. Using a separate core minimizes resource contention between the inference engine and the running VM. However, if no extra cores are available, the inference engine can run on the same core as the VM guest impacting its performance more than if dedicated cores were available. No host kernel modifications or kernel modules are needed if the inference engine is a userspace application. An optimization might be to move the inference engine to within the VMM, or enable configurable filtering of the disk block write stream by the VMM. For example, inference engines could configure the VMM to only pass along disk block writes within certain block ranges on the virtual disk. The inference engine would receive only disk block writes of interest which require further filtering or interpretation.

An inference engine begins by identifying the partition a write affects as specified in the partition table. Then it passes the write off to a *partition handler*. The partition handler may or may not pass the write on. For example, if the partition is swap, it may not be deemed important or useful as the memory pages are not reverse mappable into files or directories. However, another inference engine that understands memory could find a use for such writes and reverse map them into pages in use by the guest kernel and processes within the guest.

Assuming that the partition contains a valid file system or is an LVM volume, the write passes to an appropriate LVM handler or *file system handler*. This next layer of handlers must determine if the write is *metadata* or *data*. In the case of metadata, the write is inspected by the inference engine and appropriate data structures for reverse mapping updated. For example, the metadata manipulation might specify the creation of a new directory, or the resizing of an LVM volume. Both must be retained by the inference engine, otherwise future inferences become out of sync with the actual contents of the virtual disk. If a write is data, then the inference engine performs a full stack reverse mapping to understand which file was modified. The full architecture just described is shown graphically in Figure 4.

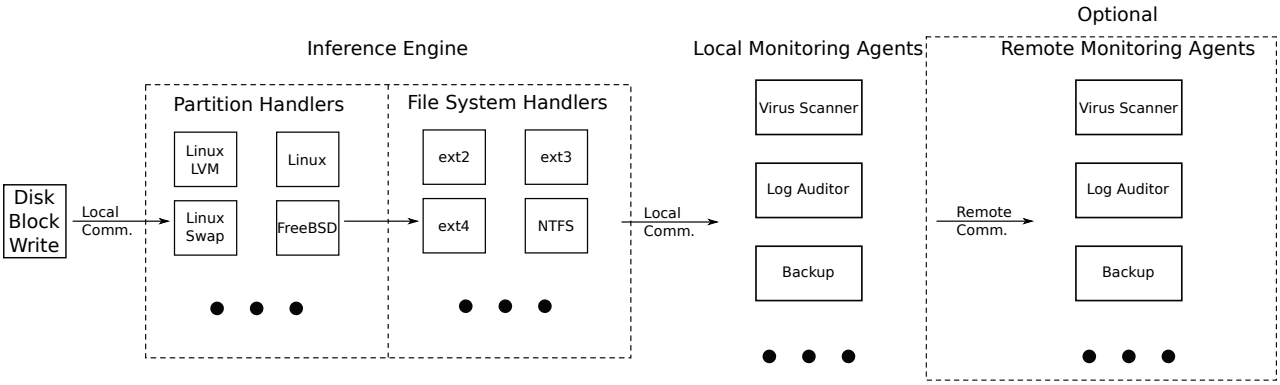

Figure 4: Block write traversal through the layers of the inference engine to registered monitoring agents.

Depending on which monitoring agents have registered with the inference engine, the modification passes to them. Monitoring agents also run on other cores, if available, and do not cause contention for CPU cycles with the running VM instance. This enables external virus scanners, external log file auditors, external synchronization of specific files and folders, and any other file-based agent to run external to the VM instance. The monitoring agents can operate across a network, or split between a local presence on the host and a remote presence in a presumably more centralized location as shown in Figure 4. A local copy of the monitoring agent could filter file-level mutations from the virtual disk and pass on important file-level mutations to the remote agent. Cooperation between the inference engine, local monitoring agents, and remote monitoring agents provides a consistent view of the virtual disk in near-real-time for analysis. The inference engine provides a stream of file-level mutations, and filtering at each level slows this stream to a manageable trickle such that a centralized system could maintain a near-real-time consistent view of thousands of VM instances.

3.2 KVM with `ext2`

KVM's design is ideal for our proposed inference engine as Figure 5 shows that all emulated IO goes to userspace by default. We only need to copy the write stream to a userspace inference engine from the emulator of the disk. In KVM's case, the emulator is generally Qemu [4]. Figure 5 shows the portion of KVM that we are interested in—the IO subsystem. We are not interested in all IO operations, only disk write operations which are separated from other IO operations by the Qemu process performing emulation.

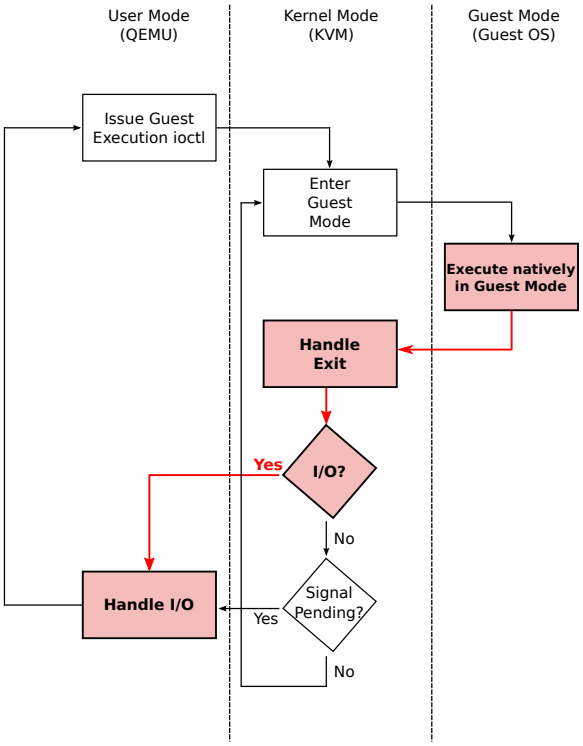

Figure 5: KVM architecture showing userspace, kernel, and physical boundaries. Figure reconstituted from [16].

For the rest of this section, imagine that we are running an inference engine with a single registered agent: an agent monitoring the file `/home/monitorme/clock.jpg`. Assume that this file is stored within an `ext2` formatted partition without LVM. Imagine that the `clock.jpg` file gets updated every 5 seconds by a webcam pointed at a clock with a second hand. Thus, our agent sees modifications to this file every 5 seconds. The inference engine discards the rest of the virtual disk IO because no other registered agents exist.

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 5 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 IO operation. Because it is an IO operation, the KVM kernel module invokes the userspace process emulating IO devices for the guest VM—in this case a Qemu process. The steps described here are highlighted in Figure 5.

Before issuing the ioctl to the KVM kernel module to return to guest mode, the write is copied to the inference engine process running as another userspace process not shown in Figure 5. At this point the inference engine takes over analyzing the write to determine what action to take. The first step requires reverse mapping the partition table. The inference engine identifies that the write is within a partition of interest—specifically the `ext2` formatted partition containing the `clock.jpg` file.

The write is then passed to an `ext2` specific handler that was initialized with reverse mapping data for this partition. 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 `ext2` includes the superblock, the block group descriptor table, inode bitmaps, block bitmaps, and inode tables. 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 `/`.

The inference engine has performed four reverse mappings: one for the initial data block to the responsible inode, and three for the three parent directories. The inference engine 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 agents are interested in this file-level mutation. In this example, there is a registered agent monitoring this file and the inference engine uses interprocess communication to notify the monitoring agent process that there is new data to consume. The monitoring agent 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 the inference engine until it associates them with a file. For exposition, we assumed the data block was immediately associable with a file.

3.3 Technical Challenges

Beyond the technical challenges in performing the reverse mappings required as described earlier in this section for multiple partitions and multiple file system types, there are several features of modern file systems and OS kernels that stymie efforts to perform this type of inference.

Encrypted File Systems or Full Disk Encryption: In the case of encryption, some activities can be inferred; however, the usefulness in terms of monitoring agents for tasks such as virus scanning or log file auditing evaporates. Data is no longer readable without the keys used to encrypt it, or a password enabling retrieval of those keys from the disk. The privacy-preserving way around this is to obtain the password or keys via an escrow service or directly from the guest owner. Only the owner of a running VM instance can judge if the benefit of external agents outweighs the risk of divulging all data contained within a virtual disk. Introspecting VM memory also reveals the encryption keys, but this clearly violates privacy without involving the VM instance owner. This challenge can be mitigated if the owners of running VMs only encrypt, for example, `/home/*`, but not other folders and files. They would still benefit from external agents without divulging the contents of every file on disk.

Transparent Compression: Compression adds an extra layer of indirection within the inference engine. If a file system has enabled transparent compression, the inference engine must decompress data blocks if they are needed for further processing. Modern file systems cluster data blocks for files and compress the clusters. For example, a file with 24 data blocks and a cluster size of 6 has 4 compressed clusters on disk. A write to any of these clusters must be decompressed with all other 5 blocks in its cluster. There are other schemes such as per-file compression schemes, but they just vary the cluster size parameter to the number of data blocks in individual files. Thus, the inference engine must also implement various decompression algorithms used by the file systems it supports.

Page Cache: The page cache of modern kernels acts as a buffer for writes to a disk in order to increase the performance of the overall system. Flushing every write to disk synchronously would cause systems to perform unnecessarily sluggish. Thus, it is unlikely that any modern kernel or user would be willing to operate without the page cache in a production environment. In addition, the page cache allows the kernel to coalesce and reorder writes for the most efficient ordering. This means that we can only infer file-level mutations that have been flushed to disk and never have a full record of all file-level mutations. Some writes may only go to the page cache, be overwritten, and lost before they ever make it to disk. Some writes may be reordered and we will not know their true sequence as initiated by system calls from writing processes.

4 Applications

In this section we assume that the inference engine and the architecture described in §3 exists and we imagine the new possibilities that arise from having the capability to infer file-level mutations in near-real-time. We describe applications of this technique in three domains: cloud, transient PC, and mobile. Remember that most applications described below are now possible through our work without running an agent inside the guest or modifying the guest kernel. The only exceptions are applications that require modifying guest disk state. For example, if cloud users wish to retrieve a deleted file, they will have to modify their virtual disk state. This is impossible to do with a black box operation because of the kernel page cache which would not have knowledge of disk changes occurring outside its purview. Thus, applications which cause modifications of virtual disk state require a cooperating agent or actions by the guest administrator.

4.1 Cloud

Cloud has the most applications as it is the setting first envisioned for the techniques described in §3; however, many of these applications are also applicable in the transient PC and mobile cases. The cloud is convenient as the use of VM technology in clouds is widespread and clouds centralize management and maintain virtual disks in VM libraries.

In general, cloud operators today can not pierce the black box of a running VM instance without installing an agent inside the instance that reports back or using modified guest kernels for reporting fine-grain information such as installed software. With an inference engine, they could inspect the virtual disks of their users in near-real-time to identify what software they are running and how they are using their instance at a disk level—what type of data they manipulate, if they install security updates etc.

Fine-Grain Monitoring: Both public and private cloud operators are left in the dark as to what software their users are running. VM images that are used as instances in clouds often have descriptive names and tags; however, once they begin executing as an instance within the cloud they are easily modified. Software that was initially installed in the original VM image can be jettisoned for newer versions, competing software, or never replaced. For example, a user might start with a known Apache [21] web server VM image. Thus, the cloud operator assumes that this user uses Apache. However, unknown to the cloud operator, the user has uninstalled Apache and installed Nginx [7] as a replacement. The cloud operator could register a monitoring agent with inference agents to track the installation and removal of software packages. This does not imply actual execution of software, but provides a view over what software interests cloud users.

System Administration: Managing large numbers of VM instances has become standard practice for many system administrators. Challenges include coalescing logs and searching them for important events, checking if security updates are installed, intrusion detection, monitoring critical configuration files for changes, and ensuring that pushed configuration changes are applied. Before, these administrators used a mix of agents that report based on scans of the file system, and tools that log in remotely to check specific files and install updates. Now, system administrators have the capability to register monitoring agents with an inference engine at each VMM to monitor log files in near-real-time, observe security update installations, watch configuration files for any modifications, and check if pushed configuration changes occurred. This is without needing to log into instances which are not necessarily used by them, or using CPU cycles of an instance they manage on behalf of another entity.

Compliance Monitoring: Enterprise clouds, and to a lesser extent public clouds, are interested in software license compliance, the proliferation of proprietary data, and securing systems through the use of approved and mandated software. Using an inference engine they attach monitoring agents to each VM instance in their cloud and monitor what software licenses are being used by inspecting known license files, what software is being used by tracking installations, if any proprietary data is leaking and how widely by scanning for known signatures, and if approved best practices are being followed by inspecting the software stack in use. For example, users might be operating with a version of software that has unpatched security vulnerabilities. This also helps in capacity planning—determining how many more licenses for software might be needed in the future based on the number currently in use.

Mirrored Instances: Using a management interface, users create two identical instances and specify that user data kept in `/home/*` be synchronized between the two instances. The cloud operator uses a monitoring agent registered at the inference engine assigned to one of the user's instances to monitor `/home/*` and replicate all observed file-level mutations to the other running instance.

In near-real-time, the cloud maintains on behalf of this user two nearly identical VM instances. If the user switched to the back up instance, they would not observe any difference beyond network configuration—although IP addresses could also be reassigned in this case.

Virus Scanning: With inference engines in place across a cloud, virus scanning is now performed outside of the running VM instance. Virus scanning can become centralized in that all file-level mutations are sent back to a central location for checks against a virus database. If the whole file is required, mutations may need to be buffered until the full file is written to disk, or if it is being updated the incoming mutation would be merged with the original file as stored in the central VM library and then scanned by the virus scanner.

Continuous Snapshotting: The frequency of snapshotting in the past has been on the order of minutes, to hours, to days depending on how paranoid users were and how much they were willing to pay. The benefit of high frequency snapshotting is that rolling back implies less loss of data if the nearest snapshot is close in time. With an inference engine, individual file-level mutations can be recorded and rolled back.

For example, with continuous snapshotting a user can revert permissions changes to a folder by finding the appropriate mutations recorded by an inference engine and instructing their cloud to revert those mutations on their virtual disk. They could also recover deleted files from their virtual disk by reverting the mutations that led to their deletion. The entire chain of mutations from VM instance start to the current time could be presented as a timeline of events recorded by an inference engine via a cloud management interface.

Efficient Backup: As in the mirroring instance case, users of a cloud specify which portions of their virtual disk they want backed up. Thus, whole virtual disk snapshots are no longer necessary cutting down on storage costs for both users and the cloud operator. In addition, backups of the specified directories or files may be kept in near-real-time.

Semantic-Based Prefetching: He et al [11] note that the iSCSI protocol benefits from caching disk blocks. Semantic knowledge about disk blocks could be used to further boost performance by prefetching blocks. For example, the cache might be prepopulated with all metadata blocks. As metadata block accesses occur, data blocks associated with them could be cached to speed reads across the network. Our scheme maintains semantic knowledge about disks in near-real-time which could optimize caching schemes inside a cloud when disks are exported between hosts.

4.2 Transient PC

Transient PC [25] is the space of computing which migrates a desktop VM between different host machines to maintain a cohesive environment for computing as a user travels—perhaps between work and the home. Many of the benefits mentioned for the cloud also apply here—system administration, compliance monitoring, virus scanning, and backup. Because transient PC users are naturally tied to a central server storing their VM while they roam, this central point acts as the ideal location for remote monitoring agents registered with inference engines. They also need to transmit all virtual disk changes back to this central point before resuming elsewhere. This is an Achille's heel for transient PC systems, because disk state change can be large—in the gigabytes—and therefore takes time to transmit.

Prioritized File Synchronization: Imagine an overburdened transient PC user working 100-hour work weeks. They are working on a document and decide it is finally time to go home at 3 AM. Unfortunately, their transient PC system needs to synchronize before they can go home to continue working on the document (naturally, they are hard working and not going home to just sleep). This delay is both annoying and unnecessary. The user stops the synchronization and then expresses to the transient PC system to only synchronize the file-level mutations from `/home/*`—where the document lives. This means that the cause of the slowdown—a recent update to their kernel and other system packages—will not be synchronized; however, from the comfort of the home those updates may be reapplied and synchronized whenever they do finally decide to sleep.

Of course, with an inference engine in place, the user can specify priorities for synchronizing data: user data first, system data second, temporary data third and so on. When they are ready to change locations, the transient PC system can synchronize data based on the user's time

constraints—the user is no longer constrained by their transient PC system. If there isn't enough time to synchronize system data, the system can stop after the user data is synchronized. The synchronization of file-level mutations could occur in near-real-time as in the mirrored instance cloud case to further reduce the wait time when moving to a new location. In addition, when resuming at home, the transient PC system can selectively prefetch disk blocks in a prioritized order based on whether they contain file system metadata or important user data.

4.3 Mobile

VM technology has already come to mobile devices [3, 13], and presumably will continue into the future for separation of business tasks from personal tasks and security reasons. Inference engines enable synchronization of file data and transfer of data from inside mobile VMs to the cloud or the home in near-real-time if energy and bandwidth permit. Mobile devices are continuously gaining more and more cores as progress by Nvidia, Qualcomm and others continues. In mobile computing transmission costs dominate CPU cycle costs. Thus, synchronizing whole VM virtual disks or even binary deltas could be prohibitively costly.

Cloudlets [24] assume that VM's are synthesizable at infrastructure close to mobile devices for low latency applications. Prior methods of VM synthesis treated virtual disks as black boxes and created opaque overlays to apply to original VM images to synthesize a VM running the desired application. The same technique, discussed above, for synchronizing user data with the cloud also aids VM synthesis: individual files and folders could be uploaded as a more compact overlay into the cloudlet for VM synthesis rather than an opaque set of bytes which might include more data than is necessary. For example, OS updates which are irrelevant to the application running inside the synthesized VM need not be uploaded to the cloudlet. The cloudlet synthesizes a VM by applying file-level mutations to a virtual disk before booting the VM, or the VM could be booted, file-level mutations uploaded, and applied from within the VM. If these files or folders are manipulated either in the cloudlet or inside a VM on the mobile phone, the overlay of folders and files can be directly updated in near-real-time.

Cloud Synchronization: Inferring file-level mutations from VM disk writes enables the same data to be synchronized to a cloud without the full overhead of binary diff-ing the VM with a cloud stored original. For example, imagine that OS updates need not synchronize with a cloud version of data, but user added music and photos must synchronize. The VMM can now be instructed to backup individual folders within running VMs on the mobile device using an inference engine.

The mobile case is not as clear, because mobile cores and IO are not as powerful as cloud servers—their focus is on the conservation of energy. However, the possibilities offered by an inference engine could provide a more secure environment by having the capability of inspecting virtual disk writes from the VMM as well as more efficiently synchronizing data by multiplexing communication across multiple VMs on the mobile device.

5 Related Work

Building blocks for our described architecture in §3 that satisfy our constraints of low IO overhead, low latency for file-level mutation notification, and scale exist scattered amongst three research communities: storage, smart disks, and VMI; however, no prior work offers all three

of our properties. The storage community [1, 9, 17, 18] tends to solve the performance issues of snapshotting, and ignores semantic understanding of disk block writes. The smart disk community [2, 27] has invented methods of semantically understanding file systems and file-level mutations in order to increase performance via intelligent prefetching and reorganizing block layout on disk. The VMI community [6, 12, 14, 15, 19, 31, 32] provides techniques for introspecting running VM guests including semantically understanding their disk write stream and disk blocks for security applications at the cost of IO performance. We gain an edge by combining the efficiency sought after by the storage community with the live semantic understanding of disk block writes developed within the smart disk community, paired with the virtual expertise of the VMI community.

Petal [17] is an early example of a system that creates copy-on-write snapshots rapidly—within 650 milliseconds. The Olive [1] distributed block storage system provides snapshotting with IO overhead in the tens of milliseconds. In addition, Olive also creates snapshots within tens of milliseconds which enables a high frequency of snapshotting. Parallax [18] provides virtual storage via a layer of storage VMs in a cloud. Parallax was explicitly designed to support "frequent, low-overhead snapshot[s] of virtual disks." Indeed, Parallax supports snapshotting within less than 30 milliseconds under heavy IO load. Snapshotting at a high frequency of 100 times per second caused only 4% IO overhead to the guest OS using the virtual disk served by Parallax. A design goal of Lithium [9], like Parallax, is to provide instant snapshots. Hansen et al. [9] do not report IO overhead while snapshotting, but presumably they have low IO overhead similar to Parallax. The storage community leads to two conclusions: (1) rapid snapshotting with low IO overhead is valuable, and (2) rapid snapshotting with low IO overhead is possible.

Semantically-smart disk systems [27] interpret metadata and the type of a block on disk as well as associations between blocks, but generally do not maintain a full reverse mapping to the file-level for all disk blocks. Thus, they are efficient because they are implemented in hardware; however, they do not have full semantic knowledge of every block write. IDStor [2] implements inference for disk block writes for iSCSI with the `ext3` file system and their application area is disk-based intrusion detection. Thus, their interface is not general, but their implementation is very close in spirit to our described architecture as it is asynchronous and out of the critical write path. Zhang et al. [32] describe a VM-based approach by leveraging smart disk technology they call, "file-aware block level storage." Their application area is also intrusion detection and they require a guest agent to run to populate their data structures for reverse mapping. In addition, their implementation blocks the critical write path of the guest OS causing a 33% overhead for write-heavy workloads. The smart disk community leads to two conclusions: (1) semantic knowledge of disk block writes is valuable, and (2) semantic interpretation of disk block write streams efficiently is possible.

Garfinkel and Rosenblum [8] coin the term *virtual machine introspection* and develop a VMI architecture that focuses on analyzing memory and requires an OS-specific introspection library. Pfoh et al. [20] develop a formal model for describing VMI techniques and the technique we present is an *out-of-band* monitoring method according to their terminology. The VMI library XenAccess [19] provides introspection of both memory and disk on the Xen platform and uses the term inference engine. Their architecture approaches the generality we intend; however, their inference engine is limited only to inferring file creations and deletions. By architecting their solution outside of the critical write path as we do, they show no statistically significant difference in performance. Zhang et al. [31] use the Extensible Firmware Interface (EFI) to implement

an inference layer between guest virtual disk writes and physical writes. Their focus is on implementing access control rules for file reads and writes outside the guest. They are in the critical IO path and presumably incur overhead similar to the 33% observed in [32]. VMWatcher [15] implements a technique called "guest view casting" to interpret memory and disk operations within a guest instance. This technique assumes that the kernel data structures and device drivers of the guest are open source or well known. Thus, they do not support proprietary file system formats such as those used by the Windows platform. VMScope [14] uses binary translation to deeply inspect and capture events such as system calls, but do not provide a framework for virtual disk write interpretation. Virtuoso [6] automatically generates introspection tools based on programs run inside of a guest. Their technique works well for operations involving the memory image of a guest, but they do not support disk operations. Hildebrand et al. [12] describe a method of performing disk introspection to the point of identifying disk blocks as metadata or data. However, inference of file-level mutations is not described. The VMI community leads to two conclusions: (1) disk-based VMI is important for security applications, and (2) efficiently introspecting disk block writes originating from VMs is possible.

CloudVisor [30] argues for a security model guaranteeing secrecy and integrity of guest VMs including disk IO. They provide an implementation based on nested virtualization that, from the guest's viewpoint, transparently encrypts and decrypts all disk IO operations. This is anathema to the VMI philosophy for monitoring and security. However, they assume that customers in a public cloud do not want introspection. This is not true of all customers and also not true for enterprise clouds where there is a single administrative domain.

6 Conclusion

We proposed an architecture for a new cloud computing mechanism enabling near-real-time monitoring of virtual disk writes in clouds without requiring guest cooperation. Our solution has low IO overhead, low latency for file-level mutation notification, and a layered design for scalability. We expect trends of cloud computing growth to continue into the foreseeable future, which increases the need for cloud-wide monitoring tools. We also anticipate a future with ubiquitous VM technology where multiple VMs run on clouds, desktops, and mobile devices for individual users. In this future, our lightweight near-real-time inference of file-level mutations offers a powerful capability aiding in seamless and quick transitions between the cloud, desktop, and mobile devices by helping synchronize important file-level mutations. The generality of our technique allows inference to operate independent of the guest VM—only understanding file system metadata on disk is needed. Near-real-time inference of file-level mutations continues the tradition of centralizing services in clouds by pushing cloud computing towards a paradigm shift to centralized system monitoring.

7 Acknowledgments

We thank our IBM collaborators specifically Vasanth Bala and Glenn Ammons for providing us inspiration, ideas, applications and support. We also thank our draft readers who provided invaluable feedback during the production of this technical report: Yoshihisa Abe, Benjamin Gilbert, Kiryong Ha, Hyeontaek Lim, Iulian Moraru, Stacy Ribeiro, Oliver Richter, and Ilari Shafer.

This research was supported by the National Science Foundation (NSF) under grant number CNS-0833882, an NSF Graduate Research Fellowship, an IBM Open Collaborative Research grant, and an Intel Science and Technology Center grant. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily represent the views of the NSF, IBM, Intel, or Carnegie Mellon University.

References

[1] AGUILERA, M. K., SPENCE, S., AND VEITCH, A. Olive: distributed point-in-time branching storage for real systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3* (Berkeley, CA, USA, 2006), NSDI'06, USENIX Association, pp. 27–27.

[2] ALLALOUF, M., BEN-YEHUDA, M., SATRAN, J., AND SEGALL, I. Block storage listener for detecting file-level intrusions. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (may 2010), pp. 1 –12.

[3] ANDRUS, J., DALL, C., HOF, A. V., LAADAN, O., AND NIEH, J. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 173–187.

[4] BELLARD, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 41–41.

[5] CARD, R., TSO, T., AND TWEEDIE, S. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux* (1994), pp. 1–6.

[6] DOLAN-GAVITT, B., LEEK, T., ZHIVICH, M., GIFFIN, J., AND LEE, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on* (may 2011), pp. 297 –312.

[7] FIELDING, R. T., AND KAISER, G. The apache http server project. *IEEE Internet Computing 1* (July 1997), 88–90.

[8] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium* (2003), pp. 191–206.

[9] HANSEN, J. G., AND JUL, E. Lithium: virtual machine storage for the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 15–26.

[10] HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 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* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 71–83.

[11] HE, X., ZHANG, M., AND (KEN) YANG, Q. Stics: Scsi-to-ip cache for storage area networks. *J. Parallel Distrib. Comput. 64* (September 2004), 1069–1085.

[12] HILDEBRAND, D., TEWARI, R., AND TARASOV, V. Disk image introspection for storage systems, US Patent Pending 2011.

[13] HWANG, J.-Y., SUH, S.-B., HEO, S.-K., PARK, C.-J., RYU, J.-M., PARK, S.-Y., AND KIM, C.-R. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE* (jan. 2008), pp. 257 –261.

[14] JIANG, X., AND WANG, X. "out-of-the-box" monitoring of vm-based high-interaction honeypots. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2007), RAID'07, Springer-Verlag, pp. 198–218.

[15] JIANG, X., WANG, X., AND XU, D. 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* (New York, NY, USA, 2007), CCS '07, ACM, pp. 128–138.

[16] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the Linux virtual machine monitor. In *Ottawa Linux Symposium* (July 2007), pp. 225–230.

[17] LEE, E. K., AND THEKKATH, C. A. Petal: distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1996), ASPLOS-VII, ACM, pp. 84–92.

[18] MEYER, D. T., AGGARWAL, G., CULLY, B., LEFEBVRE, G., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. Parallax: virtual disks for virtual machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 41–54.

[19] PAYNE, B., DE CARBONE, M., AND LEE, W. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (dec. 2007), pp. 385 –397.

[20] PFOH, J., SCHNEIDER, C., AND ECKERT, C. A formal model for virtual machine introspection. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security* (New York, NY, USA, 2009), VMSec '09, ACM, pp. 1–10.

[21] REESE, W. Nginx: the high-performance web server and reverse proxy. *Linux J. 2008* (Sept. 2008).

[22] REIMER, D., THOMAS, A., AMMONS, G., MUMMERT, T., ALPERN, B., AND BALA, V. Opening black boxes: using semantic information to combat virtual machine image sprawl. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 111–120.

[23] RICHTER, W., AMMONS, G., HARKES, J., GOODE, A., BILA, N., DE LARA, E., BALA, V., AND SATYANARAYANAN, M. Privacy-Sensitive VM Retrospection. In *Proceedings of the Third USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud '11, USENIX Association.

[24] SATYANARAYANAN, M., BAHL, P., CACERES, R., AND DAVIES, N. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE 8*, 4 (oct.-dec. 2009), 14 –23.

[25] SATYANARAYANAN, M., GILBERT, B., TOUPS, M., TOLIA, N., SURIE, A., O'HALLARON, D. R., WOLBACH, A., HARKES, J., PERRIG, A., FARBER, D. J., KOZUCH, M. A., HELFRICH, C. J., NATH, P., AND LAGAR-CAVILLA, H. A. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing 11* (March 2007), 16–25.

[26] SIVATHANU, M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND JHA, S. 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, pp. 1–1.

[27] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), USENIX Association, pp. 73–88.

[28] TEIGLAND, D., AND MAUELSHAGEN, H. Volume managers in linux. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 185–197.

[29] TRIPWIRE. Open source tripwire, January 2012.

[30] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. 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* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 203–216.

[31] ZHANG, X., ZHANG, S., AND DENG, Z. Virtual disk monitor based on multi-core efi. In *Proceedings of the 7th International Conference on Advanced Parallel Processing Technologies* (Berlin, Heidelberg, 2007), APPT'07, Springer-Verlag, pp. 60–69.

[32] ZHANG, Y., GU, Y., WANG, H., AND WANG, D. 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, pp. 185–192.