RC25669 (WAT1710-006) October 4, 2017
Computer Science

IBM Research Report

A Cloud-Native Monitoring and Analytics Framework

Fabio A. Oliveira, Sahil Suneja, Shripad Nadgowda, Priya Nagpurkar, Canturk Isci
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

Research Division
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

A Cloud-native Monitoring and Analytics Framework

Fabio A Oliveira, Sahil Suneja, Shripad Nadgowda, Priya Nagpurkar, Canturk Isci
IBM T.J. Watson Research
{fabolive,suneja,nadgowda,pnagpurkar,canturk}@us.ibm.com

Abstract

Operational visibility is an important administrative capability and is one of the critical factor in deciding the success or failure of a cloud service. Today, it is increasingly becoming more complex along many dimensions which include being able to track both persistent and volatile system state, as well as provide higher level services such as log analytics, software discovery, behavioral anomaly detection, drift analysis to name a few. In addition, the target endpoints to monitor are becoming increasingly varied in terms of their heterogeneity, cardinality, and lifecycles, while being hosted across different software stacks. In this paper, we present our unified monitoring and analytics pipeline to provide operational visibility, that overcomes the limitations of traditional monitoring solutions, as well as provides a uniform platform as opposed to configuring, installing and maintaining multiple siloed solutions. Our *OpVis* framework has been running in our production cloud for over two years, while providing a multitude of such operational visibility and analytics functionality uniformly across heterogeneous endpoints. To be able to adapt to the ever-changing cloud landscape, we highlight it's extensibility model that enables custom data collection and analytics based on the cloud user's requirements. We describe its monitoring and analytics capabilities, present performance measures, and discuss our experiences while supporting operational visibility for our cloud deployment.

1 Introduction

In cloud environments, operational visibility refers to the capability of collecting data about the underlying system behavior and making this data available to support important administrative tasks. Without visibility into operational data, cloud operators and users have no way to reason about the health and general behavior of the cloud infrastructure and applications.

Traditionally, the operational visibility practices have been limited to resource monitoring, collection of metrics and logs, and security compliance checks on the underlying environment. In today's world, better equipped to manipulate massive amounts of data and to extract insights from it using sophisticated analytics algorithms or machine-learning techniques, it becomes natural to broaden the scope of operational visibility to enable, for instance, deep log analytics, software discovery, network/behavioral anomaly detection, configuration drift analysis, to name a few use cases.

To enable these analytics, however, we need to collect data from a broader range of data sources. Logs and metrics no longer suffice. For example, malware analysis is done based on memory and filesystem metadata, vulnerability scanning needs filesystem data, network analysis requires data on network connections, and so on. At the same time, these data sources are potentially very different in nature. Log events are typically continuously streamed, whereas filesystem data changes are less frequent, and configuration changes normally occur when an application is deployed.

Yet another source of data for modern operational visibility stems from the diverse and prolific image economy (DockerHub, Amazon Marketplace, IBM Bluemix) that we witness as a result of pervasive virtualization. The more the world relies on cloud images, the more important it becomes to proactively and automatically certify them by performing security and compliance validation, which requires visibility into dormant artifacts, in addition to running cloud instances.

Adding to the complexity of dealing with a multitude of data types for modern operational visibility, cloud environments are becoming larger and increasingly heterogeneous. For instance, it is nowadays common for a cloud provider to support deployments on physical hosts, virtual machines (VMs), containers, and unikernels, all at the same time. As a result, for more effective visibility, operational data from this diverse set of runtimes needs to be properly collected, interpreted, and contextualized. Tenancy information, resource limits, scheduling policies, and the like are exposed by different cloud runtime platforms (e.g., Openstack, Kubernetes, and Mesos) in different ways.

As if heterogeneity were not enough, the lighter the virtualization unit (e.g., containers and unikernels), the higher the deployment density, which leads to a sharp increase in the number of endpoints to be monitored. Figure 1 summarizes the complexity of modern cloud environments along multiple dimensions, including deployment types and cloud runtimes, as well as some challenges for which operational visibility is needed.

In this paper, we propose a novel approach to operational visibility to tackle the above challenges. To enable increasingly sophisticated analytics that require an ever-growing set of data sources, we implemented *OpVis*, an **extensible** framework for operational visibility and analytics. Importantly, *OpVis* provides a **unified** view of all collected data from multiple data sources and different cloud runtimes/platforms. *OpVis* is extensible with respect to both data collection and analytics.

We contend that an effective operational visibility platform must decouple data collection from analytics. Old solutions that attempt to mix data collection and analysis at the collection end do not scale, and are limited to localized rather than holistic analytics. We enable algorithms that can uncover data relationships across otherwise separated data silos.

Furthermore, to scale to the increasing proliferation of ephemeral, short-lived instances in today's high-density clouds, we propose an **agentless**, non-intrusive data collection approach. Traditional agent-based methods are no longer suitable, with their maintenance and lifecycle management becoming a major concern in enterprises.

Our implementation of *OpVis* supports multiple data sources and cloud runtimes. We have been using it in a public production cloud environment for over two years to provide operational visibility capabilities, along with a number of analytics applications

IBM Research Technical Report, USA
2017.

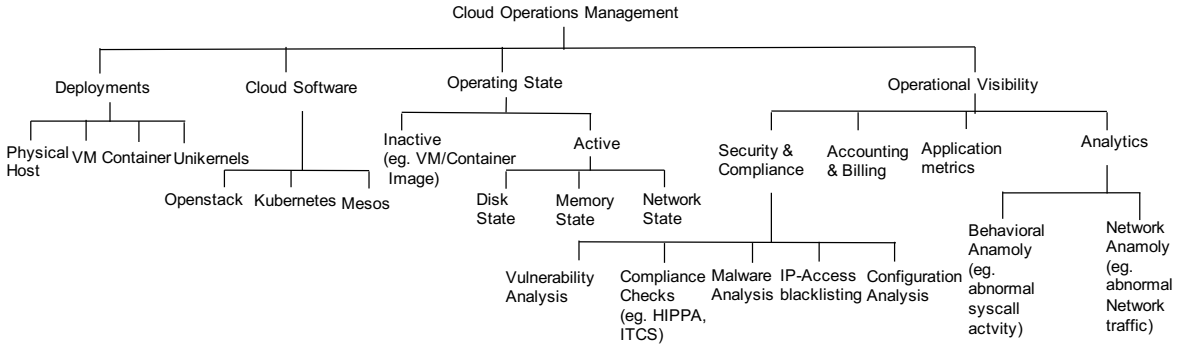

Figure 1. Cloud operation management.

we implemented to, among other things, provide security-related services to our cloud users.

We evaluate *OpVis* with a combination of both controlled experiments and real production data in cases where we were allowed to publicize it.

2 Existing Techniques

To gain visibility inside VMs, most existing solutions usually repurpose traditional monitoring tools, typically requiring installation inside the monitored endpoint's runtime, and thus causing guest intrusion and interference [4]. Other's avoid application-specific agents by installing generic hooks or drivers inside the guest [34, 58–60], requiring VM specialization, leading to vendor locking. The ones that do not require guest modification, can usually provide only few black box metrics, for example by querying the VM management consoles like VMware vCenter and Red Hat Enterprise Management. Yet another approach is to gather metrics by remotely accessing the target endpoints (e.g. over SSH, or via HTTP queries). A combination of one or more of these techniques is also typically seen in some solutions [3, 45].

The landscape is a bit different with containers, since the semantic gap between the guest environment and the management layer (host) is greatly reduced, with containers at their core being just a special packaging of host processes and directories. In addition to container image scanning [5, 17, 48, 49, 57], some solutions are able to provide some level of agentless, out-of-band container inspection by talking to Docker daemon [20, 50], querying kernel's cgroups stats [13], monitoring container's *rootfs* [27], or via syscall tracing [54]. While these are able to provide basic metrics, for deep inspection most resort to installing in-guest components-agents, plugins, scripts, instrumentation libraries, or custom exporters [19, 44, 50], and thus require guest modification.

Furthermore, most existing solutions address only certain subcomponents of operational visibility, and only a few (seem to) cover all amongst image scanning, as well as out-of-band basic metrics and deep inspection [27, 48, 57]. And amongst those handful, none are opensourced and extensible (although several of the aforementioned ones are).

These properties translate to installation, configuration and maintenance of multiple siloed solutions to cover all aspects of the operational visibility spectrum (Figure 1). And given the above arguments, to the best of our knowledge, no existing solution provides all of OpVis' capabilities of a unified, agentless, decoupled, extensible and opensourced operational visibility framework, that does not enforce guest cooperation or cause guest intrusion, interference and modification.

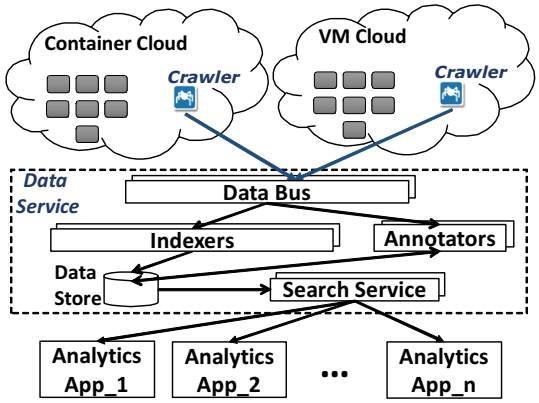

Figure 2. *OpVis* overview.

3 Design and implementation

In this section we describe the design and implementation of *OpVis*, our unified operational visibility and analytics framework. The overall architecture of *OpVis* is depicted by Figure 2, which clearly separates three layers: data collection, data service, and analytics. We refer to *OpVis* data collectors as *crawlers* (see top of Figure 2). They monitor cloud instances and images to take periodic memory-state and persistent-state snapshots, which are then encoded into an extensible data format we refer to as the *frame*. In addition to discrete state snapshots, the crawlers track log files of interest from cloud instances. Snapshots, in the form of *frames*, and streaming log events enter the data service through a scalable data bus from which they are fetched and then indexed on a data store for persistence. A search service makes all collected data available and queriable, enabling a variety of analytics applications, for instance, to diagnose problems experienced by a cloud application, to discover relationships among application components, and to detect security vulnerabilities. The rest of this Section describes the *OpVis* data collectors (*crawlers*) (§3.1), data format (*frame*) for discretized state snapshots (§3.2), and backend data service (§3.3). We also

present a few analytics applications that take advantage of *OpVis* (§4).

3.1 Data collectors: agentless *crawlers*

We take an *agentless* approach to data collection, that is, *OpVis* crawlers collect data in an *out-of-band*, *non-intrusive* manner. Critically, an important role of the crawlers is to enable operational visibility with a *unified* view across different cloud-runtime types and for different forms of application and system state. We implement out-of-band visibility into container runtimes, e.g., plain Docker host, Kubernetes, and Mesos, and into VM runtimes, e.g., OpenStack.

We observe that monitoring live cloud instances (containers and VMs) is important for reactive analytics; however, to enable proactive analytics applications it is equally important to also scan cloud images (Docker images and VM disks). For this reason, *OpVis* crawlers provide visibility into these dormant artifacts as well.

The types of analytics applications that can be written are limited by the collected data. To enable semantically-rich end-to-end visibility and analytics, the crawlers collect in-memory, live system state, e.g., resource usage and running-processes information, as well as persistent system state, e.g., filesystem data and logs. This broad range of data types requires proper manipulation of continuously-streaming data, such logs, as well as state that needs to be taken at discrete snapshots, such as process, network, and filesystem information.

Exposing and interpreting persistent and volatile state of VMs and containers requires techniques tailored for each runtime and state type. Broadly, we apply introspection and namespace-mapping techniques for VMs and containers, respectively. Despite slight differences in the specifics of the techniques, the key tenet of our approach remains unchanged: to provide deep operational visibility in near real time and out of band, with no intrusion or side effects on the running cloud instances. Next, we delve into our implemented techniques, organizing our presentation by runtime (container and VM) and broad state category (memory and persistent).

3.1.1 Containers' memory state

With the advent of Docker [22], Linux containers have become an increasingly popular choice for virtualizing cloud infrastructures. Containers represent a type of OS-level virtualization where related processes can be grouped together into a logical unit (container). Each container is given a unique, isolated view of the system resources. Two Linux kernel features, *namespaces* and *cgroups*, are used to guarantee virtualization, isolation, and controlled resource usage for each container. Different types of namespaces provide isolation for different kinds of resources. For instance, the `pid` namespace controls process virtualization; thus, it makes the processes inside a container to only see each other, as they belong to the same `pid` namespace.

Because container processes are simply host processes with a different view of the system, they are visible from the host. We use two techniques to collect a container's memory state information. The first route is via cgroup accounting stats. Most OS virtualization technologies provide some way of accounting container resource utilization, like cgroups in the Linux Kernel. This is specially needed for performance isolation, where it is necessary to limit the memory and CPU utilization of containers so they do not take over the host.

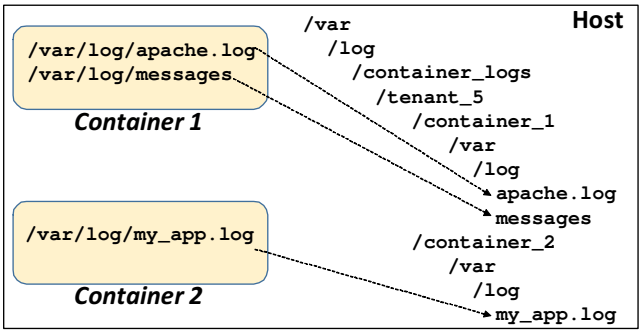

Figure 3. Mapping log files from containers to the host.

These resource utilization stats include: disk IOs, CPU cycles, user memory, and kernel memory (some).

Our second container-monitoring technique relies on the Linux namespace APIs in such a way that the crawler process running on the host "attaches" to the `pid` namespace of each container to collect data from it. Although this is not a pure introspection technique, as the crawler logically "moves" into the container context, it nonetheless gives us the same non-intrusiveness character we seek. It works even if the container is unresponsive or compromised, since the crawler still gets the overall system view from outside the container. Importantly, it does not require any special agent or library inside the containers.

In summary, with the above techniques the crawler collects containers' live memory state, including information on hardware-usage metrics, processes, and network connections.

3.1.2 Containers' persistent state

Besides live memory state, we also want to collect the persistent state of running containers. In contrast to the namespace-attachment technique described previously, we employ a purely host-side monitoring scheme to collect persistent-state data. The Linux kernel maintains a mapping from resources as seen inside containers to the actual host resources. One such mapping pertains to the root filesystem of each container. Our crawlers identify the location of each container's root filesystem in the host filesystem, and then extract containers' filesystem data and metadata, information on configuration files, and installed packages. (Note that each memory and persistent-state discrete data type is described in detail in §3.2.)

Just as importantly, the same persistent-state data can be collected from dormant container (Docker) images. To do so, the crawler creates a container for each image to be scanned, applies exactly the aforementioned technique for extracting persistent state, and destroys the container. Since the time to create and destroy a container is extremely short, this technique is quite reasonable. Another technique is to mount the container image on the crawler host and perform the inspection offline. Image scanning is performed on demand, i.e., when a new image is pushed to the cloud or an existing one is modified.

Thus far we have presented the collection of discretized state, taken as periodic snapshots. An important form of persistent state does not fall into that category. Log events continuously streamed

from containers provide important pieces of evidence for debugging and analytics applications alike. Capturing log streams from containers is therefore of paramount importance. In the next paragraphs, we describe the implementation details of log collection for containers.

Log-file mapping. Two components of the crawler work together to deal with logs. The crawler's *container watcher* constantly polls the host where it runs to discover new containers; when a newly-created container is discovered, the *container watcher* maps log files of interest from the container filesystem to the host filesystem so that the *log handler* can monitor them.

Figure 3 illustrates this logical file mapping. The top directory of all monitored log files in the host is `/var/log/container_logs`, under which the *container watcher* creates one sub-directory per cloud tenant (user). Finally, under a tenant sub-directory, one directory per container is created, and that becomes the new root directory of all log files of that container. In the figure, `container_1` has two log files to be monitored, and `container_2` has one. Both containers are owned by a tenant whose id is `tenant_5`.

Following this approach, the *log handler* independently discovers new log files to be monitored by watching recursively the contents of `/var/log/container_logs/**/*`.

Finding log files of interest. In order to identify log files to be monitored, the *container watcher* inspects the environment of a container, looking for a variable named *LOG_LOCATIONS*. When creating a container, the user is expected to set this variable to a string whose value is a comma-separated list of paths to log files of interest. Other variables in a container's environment, automatically set by the cloud, uniquely identify the user owning the container and the container itself.

To implement the file mapping described above and depicted by Figure 3, we rely on the fact that the filesystem of a container is naturally exposed to the host, as explained earlier. The *container watcher* can then find the location of all container log files in the host filesystem and create symbolic links to them so that they can all be found by the *log handler* under `/var/log/container_logs/**/*`.

In addition to allowing users to provide a list of log files to be monitored, the crawler treats the standard output of a container as a log file. In fact, the standard output of a container appears in the host filesystem as a file, which is subjected to the above log-file mapping scheme. The standard output of a container is always monitored, even if the cloud user specifies no log files of interest.

Once the log-file mapping is established, the *log handler* continuously tracks the log files and streams to the data service in near real time the log events as they appear. Our *log handler* implementation is based on Logstash [26].

3.1.3 VMs' memory state

Since VMs have their own OS kernel and therefore keep their internal memory state hidden from the host, they are more difficult to monitor than containers. We use and extend VM introspection (VMI) techniques [29] to gain an out-of-band view of VM runtime state. We have developed solutions to expose VM's memory live with negligible overheads for KVM VMs (access in Xen via hypervisor-exported APIs). Since KVM is part of a standard Linux environment, we leverage Linux memory management primitives and access VM memory via QEMU process' /proc/<pid>/mem pseudo-file, indexed by the virtual address space backing the VM's memory from /proc/<pid>/maps.

Feature Type	Example Feature Elements	Crawl Mode		
		Host	Container	VM
OS	Type, distro, version, platform	Y	Y	Y
Memory	Used, cached, free, utilization %	Y	Y	Y
Interface	Bytes/packets tx/rx	Y	Y	Y
Process	Cmd, created, pid, cwd, openfiles	Y	Y	Y
Metrics	Per process cpu %, rss, io bytes	Y	Y	Y
Connection	Src/dst ip/port. Conn status	Y	Y	Y
CPU	Idle, wait, user, utilization %	Y	Y	N
Load	Short/ mid/ long term sys load	Y	Y	N
Disk	Fstype, mountpoint, free %	Y	Y	N
Config	Parameter name, value, path	Y	Y	Y
Package	Name, size, version	Y	Y	Y
File	Name, path, size, atime, ctime	Y	Y	Y
Docker-ps	Status, image, create_time, ports	Y	N	N
Docker-history	Container image history	Y	N	N
Kernel-module	Name, state	N	N	Y
CpuHardware	Family, vendor, khz, num_cores	N	N	Y
Ruby/python pkg	Package name, version	Y	Y	Y
Netflow	Netflow v1,5,9,10 data	N	Y	N
Apache / Nginx	Workers, connections, bytes/s	Y	Y	N
Redis	Hits, misses, evicted_keys	Y	Y	N
Tomcat	Request count, JVM free mem	Y	Y	N

Figure 4. Examples of features extracted by different crawler plugins. An 'N' in a cell represents currently not implemented or non-applicable functionality for a particular crawling mode. Host crawling uses OS exported functionality, Container crawling uses kernel's cgroups and namespace APIs, and container rootfs traversal, and VM mode uses memory and disk introspection.

The obtained external view of raw VM memory and disk is wrapped and exposed as network-attached devices (FUSE over iSCSI). This way, the actual crawling logic, as well as monitoring and analytics components are completely decoupled from VM execution.

The backend attaches to this raw, byte-array VM memory view exposed by the frontend, and implements the crawl logic that performs the logical interpretation of this raw state into structured runtime VM state. This is achieved via in-memory kernel data structure traversal. Briefly, we overlay the struct templates for various kernel data structures (e.g. `task_struct` for processes. `mm_struct` for memory mapping, etc.) over the exposed memory, and traverse them to read the various structure fields holding the relevant information. Further details can be found in our previous work [53], which also describes several enhancements for selective memory extraction and guest OS-consistent VM memory access.

After introspecting all the relevant data structures, the extracted state (see Figure 4 for the various extracted features) is wrapped into a single structured document (the frame), with its corresponding crawl timestamp and VM ID, which the backend analytics pipeline feeds off of.

3.1.4 VMs' persistent state

Exposing and collecting a VM's persistent state non-intrusively requires VM disk introspection. Our design follows certain key principles: (1) the persistent-state collection of offline and live VMs must be identical; (2) it must have negligible impact on the VM's

runtime; and (3) it must be done from outside the VM and not lead to any runtime or persistent-state change.

For offline VMs, one could implement this simply by leveraging standard device mounting and filesystem utilities. However, for running VMs, this process cannot be accomplished in the same way. As the VM is actively running and accessing its disk, the disk inherently holds a *dirty* state; thus, standard mounting approaches break at this step. Even though there exist methods to circumvent this problem, the solutions they provide violate our key principles: they create side effects on the running VMs and intrude into the VM's operation.

Our approach, implemented for KVM/OpenStack, respects the above key principles and works as follows. First, a special crawler component, the *VM disk introspector*, needs to identify the device configurations of the VMs. We use OpenStack and QEMU APIs to determine disk layers for running VMs. Next, as the VMs are running while the crawler accesses the disks out of band, we expose all identified disk layers as *read-only* pseudo-devices to ensure that no action can alter the device state at the physical level. Moreover, because the disks being accessed out of band are live and hence inherently *dirty*, we use Linux device mapper reverse snapshots to wrap each pseudo-device with a separate Copy-on-Write (CoW) layer. Then, this new device view can be exposed as either a local storage device or a network-attached one (e.g., iSCSI). Finally, the exposed device is mounted on the crawler VM so that it can access the target VM's filesystem over the entire device view to collect the exposed persistent state, namely, filesystem data and metadata, configuration files, and installed packages.

This technique works for both raw and QCoW2 images, and can be applied uniformly to offline and live VMs. During the entire process, a live VM continues its execution and disk accesses normally; there is no need for disk copy.

For log collection, after the *VM disk instrospector* performs all the above actions to expose a VM's filesystem, the *log handler* can track updates to the log files of interest in near real time and stream them to the data service.

3.1.5 Crawler extensibility

Extensibility has always been a core design principle for the OpVis crawler. We envisioned various sources of heterogeneity (Fig. 1) in the operating environment, configurations, monitored end-points. As a result, we adopted plugin architecture to accommodate various extensibility dimensions. Figure 5 shows the overall plugin design described next.

Crawl Mode We support *five* modes in which crawler can be started: (i) INVM: for host or agent-based VM crawling, (ii) OUTVM: VMI-based agentless VM memory crawling, (iii) MOUNTPOINT: for VM disk crawling after mounting its virtual disk image, (iv) OUTCONTAINER: for crawling containers via Docker APIs, container rootfs, and kernel's cgroups and namespaces API, and (v) K8SDS: mode added specifically for kubernetes platform wherein crawler is deployed as a privileged container in a k8s daemonset pod.

Crawl Plugins Logic for every metric (a *feature* in OpVis terminology, see Figure 4) being collected by the crawler is implemented as a separate crawl plugin. This allows flexibility of configuring the crawler to selectively enable different plugins for different environments. Crawl plugins extract both application and system

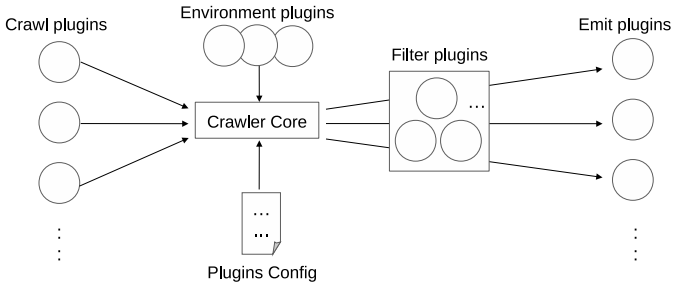

Figure 5. OpVis crawler's plugin architecture

state. Data collection extensibility is easily achievable via a new crawl plugin, by abiding a contract with the crawler core, where the plugin extends an interface class (corresponding to a host, VM or container target), and implements a `crawl()` function that returns a *feature_type* identifier and *feature_elements* dictionary. Similar extensibility models follow for data emitters, filters and environment plugins, as described next.

Emitter Plugins The OpVis crawler supports combinations of different data formats (csv, json, graphite) and emitter endpoints (stdout, file, http(s), fluentd, kafka, mtgraphite). This spectrum of data emitting capabilities enables the crawler to cater to various types specialized data stores for analytics and monitoring.

Environment Plugins Environment information is an orthogonal dimension for data collection but is important to establish the context for the entity being crawled. For example, multiple containers from different subnets could have same ip-address, therefore, in addition to collecting network state of a container the tenancy information and network topology is important to resolve and further analyze collected data. Environment plugins serve to provide this context information, by adding a `metadata` dictionary to the emitted frame containing custom namespace, owner, and tenancy information amongst other elements. For example, a kubernetes environment plugin adds k8 metadata such as container labels and pod IDs to containers' frames.

Filter Plugins These provide data aggregation and filtering capabilities atop the extracted features. Examples include metrics aggregator (average/min/max) plugins, as well as `diff` plugins to send only frame deltas to the backend.

3.2 Frame: state snapshot

A *frame* is a structured representation of a snapshot of a container or VM, encompassing memory and persistent state, taken by the crawler using the data collection techniques previously presented. Log events are not part of a frame, as they are streamed rather than discretized.

We refer to each element in a frame as a *feature*, which in turn embodies a collection of key-value pairs where the keys are *feature attributes*. The set of attributes of a feature depends on the *feature type*. Each feature type corresponds to a type of memory or persistent state collected by the crawler. The feature types we define include: `os`, representing general information on the operating system, `process`, corresponding to OS processes, `connection`, encapsulating information on network connections, `file`, associated with metadata of filesystem objects, `config`, representing the

contents of filesystem objects identified as configuration files, and `package`, for metadata on OS-level and programming-language-level packages. We also define types for discretizing resource-usage metrics, e.g., for CPU and memory. Our framework can be extended with a new type declaration and the corresponding crawler's data-collection plugin implementation.

As a concrete frame example, imagine a running container. Now, suppose the crawler is about to take a snapshot (collect state) of the container. If, at that time, the container has four running processes, the frame representing the snapshot will contain four features of the type `process`. The attributes of each feature will identify each process, e.g., name, pid, parent id, and command line. Similarly, the frame will contain one `file` feature for each filesystem object inside the container at data-collection time; and so on, for all other feature types.

In addition to features, a frame has metadata to capture important aspects of the snapshot. In particular, a *timestamp* indicates when the snapshot was taken, allowing analytics applications to run temporal queries to reason about state evolution. Also, associated with a frame is a *namespace*, which is used to identify the cloud instance in question, typically as a combination of a cloud-assigned id and a user-provided string with a name and version of the cloud application/service. Finally, a *group* identification is used to allow the aggregation of frames pertaining to related cloud instances, e.g., those that are part of the same auto-scaling group. Other pieces of metadata provide provenance information to identify image versions and cloud users.

3.3 Data service backend

The *OpVis* data service is illustrated in the middle part of Figure 2. It comprises a data pipeline whose entry point is a scalable, replicated, and fault-tolerant data bus. To realize our data-bus cluster we use Apache Kafka [6]. One key function of the data bus is to provide buffering, which is critical when the data-ingestion rate exceeds the data-consumption rate. Kafka allows data producers to *publish* data to different *topics*. Thus, frames and log events emitted by the crawler enter the data pipeline through two different Kafka topics.

In the next stage of the data pipeline, clusters of indexers fetch data from Kafka. Frame indexers *subscribe* to the frame topic and store the incoming frames on Elasticsearch [25], a data store based on the index-and-store Apache Lucene [7] engine. Once indexed on Elasticsearch, a frame representing the system state of a container or VM becomes a searchable document. We refer to this general management paradigm as *state as documents*. Using the Elasticsearch query language, logically corresponding to the search service in Figure 2, users, operators, or analytics applications can execute semantically-rich queries to find frames and frame features. Every attribute of every feature of every indexed frame can be used as a key query. Similarly, the frame's metadata fields can also be query keys. This notion of applying search to manipulate operational data (system state) made visible, derived from the *state-as-documents* paradigm, is extremely powerful, as the analytics applications presented in §4 demonstrate.

We used Logstash [26] to implement our frame indexers. In particular, we relied on two Logstash plugins: Kafka *input plugin* and Elasticsearch *output plugin*. To process the frames as emitted by the crawlers, add some pieces of metadata to them, and convert them into a proper Elasticsearch document, we implemented a new Logstash *filter plugin*.

Backend extensibility. The *annotators* (see Figure 2) are also an important element of our data pipeline. An annotator's key goal is to read frames of interest to create and index a different type of document. This is done to support certain analytics applications that might need to search for these special, curated documents.

We distinguish between two categories of annotators: privileged and regular. Privileged annotators typically originate from the cloud provider to support analytics applications that have general applicability, e.g., to detect vulnerabilities in all cloud-user images (see §4.1). These annotators fetch frames directly from Kafka, like the indexers. To help cloud providers implement privileged annotators, we provide a Python framework with wrappers for reading frames from Kafka and indexing new Elasticsearch documents. Also, since the frame format is well defined, we provide functions to facilitate frame manipulation. Reading frames from Kafka saves cycles from Elasticsearch, which is the portion of the data pipeline exposed to end users (with multi-tenancy controls).

Regular annotators, on the other hand, can be created by cloud users if there is a need for querying a document type not supported by the existing privileged annotators. User-provided regular annotators are deployed on a sandbox environment provided by our serverless cloud infrastructure, and they read frames from Elasticsearch, not Kafka. To help end users create annotators, we provide well-defined, high-level APIs to transparently read frames from Elasticsearch, manipulate retrieved frames, and trigger the indexing of annotator-specific Elasticsearch documents.

Log indexing. Log indexers subscribe to the log topic and store incoming log events on Elasticsearch. Like the frame indexers, we used Logstash to implement them. Unlike frames, which have a well-defined structure, cloud application logs can have any format, since many of them are application-specific. This implies that performing semantically-rich Elasticsearch queries on logs (using attributes in log records as keys) might not be possible, unless the log indexers know what log format to expect. Before emitting logs, the crawler's *log handler* tries to apply the Logstash JSON filter plugin; thus, if logs coming from cloud applications are in the JSON format, our log indexers will guarantee that log records can be queried by log attributes, whatever they might be. Note that free-text queries are still possible, even if logs are emitted in an unknown format. Those queries, however, will be no better than grep-style searches.

4 Analytics applications over *OpVis*

Our operational visibility pipeline has been running in our production cloud for over two years. It has been providing our clients several security oriented operational insights derived from their images, containers and VMs. We highlight four such services in this Section, focusing specifically on Docker images and containers, while the analytics presented are independent of the target runtime.

To run their containers, users typically download images from public repositories like Docker Hub [21]. Studies have highlighted the various security vulnerabilities that such images tend to contain [10]. In a pure container cloud like ours, where containers are hosted directly on the host OS, security of client images and containers becomes paramount since a vulnerability in the client runtime (or a malicious container itself) can lead to a malicious intrusion, as well as an escape to host, enabling exploits on co-located container instances as well as the cloud infrastructure itself.

The common feature across all our security services built over the *OpVis* pipeline is that they do not require any setup prerequisites to be built in to the user's runtime, and start working immediately as soon as a user brings in their images or runs a container or VM. These services feed off of systems frames from the kafka pipe, perform analytics-specific 'annotations' on the frames, generating user-visible 'reports' highlighting the security posture of their images and systems.

4.1 Vulnerabilty Analyzer (VA)

VA discovers package vulnerabilities in a user's Docker images and containers hosted on our public cloud, and guides them to known fixes in terms of relevant distribution-specific upgrades. Once a system frame enters the *OpVis* analytics pipeline, the VA annotator extracts the package list from it and compares its against publicly available vulnerability databases (e.g., NVD). We currently target Ubuntu security notices while support for other distributions is in the process. Listing 1 shows a sample VA report, containing CVEs and a url to the security notice corresponding to the vulnerable package found during scans. VA has been integrated as part of the DevOps deploy pipeline where, based upon user-specified deployment policies, images tagged as vulnerable by VA can be blocked against deployment as containers.

Recently, we have also added support to scanning vulnerabilities of application runtime-specific packages brought in by application-level package installers like Ruby gems and Python pip packages. An interesting security dimension this feature addresses is defense against typosquatting attacks on application libraries [55]. In this case, malicious packages similar in names to legitimate/intended packages make their way into a user's system, when the user inadvertently makes a typo while executing an installation command, for example, `pip install reqeusts` instead of `requests`. By comparing installed packages against permutations of whitelisted ones, such malicious packages can be detected and protected against.

We are also currently converting VA to an as-a-service model, and shall soon release a version enabling easy extensibility in terms of supported environments beyond just our own cloud, as well as custom client analytics.


```
  {
  "vulnerability-check-time":
"2017-05-18T08:45:02.696Z",
  "os-distrbution": "ubuntu",
  "os-version": "xenial",
  "vulnerable": true,
  "crawled-time": "2017-05-18T08:45:00",
  "execution-status": "Success",
  "namespace": "container:10.0.2.15/ubuntu",
  "vulnerabilities": [
  "cveid": [
  "CVE-2016-6252", "CVE-2017-2616" ],
  "summary": "su could be made to crash or stop
programs as an administrator.",
  "url": "http://www.ubuntu.com/usn/usn-3276-1",
  "usnid": "usn-3276-1"
  "type": "vulnerability"
  }
```


Listing 1: Sample Vulnerability Analyzer report

4.2 SecConfig

Software misconfiguration has been a major source of availability, performance and security problems [63]. For an application developer using third-party components shipped as standard Docker images, it is non-trivial to ensure optimal values for various configuration settings spread across the different components. To aid in this process, SecConfig scans applications and system configuration settings to test for compliance and best practices adherence from a security perspective.

SecConfig gives container developers a view into their runtimes' security properties, and gives guidance on how they should be improved to meet best practices in accordance with certain industry guidelines lik HIPAA, OWASP, PCI, and CIS [40], mixed with own internal deployment standards. We currently have a small pool of 'policies' (135 rules spanning 11 different applications and system services), which we test 'compliance' against as part of SecConfig. Examples configuration rules include passwords to be greater than 8 characters, and set to under 90-days expiration, secure ciphers to be used in apache's SSL/TLS settings, etc. A sample report snippet can be seen in Listing 2.

Current efforts include making SecConfig SCAP-validated [39] which can then enable clients to confidently gauge if their systems are in full compliance with authorized agency-specific security checklists like USGCB, DISA STIG, etc.


```
  {
  "compliance-check-time":
"2017-05-17T02:23:45.804772Z",
  "compliance-id": "Linux.9-0-a",
  "compliant": true,
  "crawled-time": "2017-05-17T02:23:41.145098Z",
  "description": "checking if ssh server is
installed",
  "execution-status": "Success",
  "namespace": "container:10.0.2.15/mysql:5.7",
  "reason": "SSH server not found",
  "type": "compliance"
  }
```


Listing 2: Sample SecConfig report

4.3 Drift Analysis

One of the key tenets of DevOps automation is enforcing container immutability. Once a container goes into production, the expectation is that it would never be accessed manually (login or ssh) and thus its contents or behavior at day zero, tightly controlled with deploy scripts to adhere to an overall architecture, *should* be the same thereafter. However, it has been observed that systems inevitably 'drift' [1]- deployed containers change over time and show unexpected behavior, with the change being either persistent, as in disk-resident, or existing solely in-memory. Such drift can introduce unexpected exposures and side effects on deployed applications and can go unnoticed for a long time with image-centric validation processes.

With our *OpVis* pipeline, we are able to detect such drift by tracking evolution across time for all monitored systems. Specifically, by *diff*ing system frames across time, we can discover 'which' systems violated the immutability principle, and then narrow down

on 'what' caused that drift by uncovering potential causes ranging from package level changes to even the granularity of the suspect application configuration settings.

In one particular instance, in an analysis of our production cloud over a 2 week period, we found that almost 5% of the hosted containers exhibited drift between containers and their corresponding images, in terms of difference in their vulnerability and compliance counts (as described in the previous two subsections). Although one reason could be the change in vulnerability database itself, our analysis uncovered that 'in-place' updates to the containers, both benign [1] and undesirable, were indeed taking place in our cloud via SSH, docker exec, automated software updates, and software reconfiguration via web frontends.

4.4 Malware Analysis

Malwares typically manifest themselves in various forms including rootkits, viruses, botnets, worms, and trojans that infect and compromise the operating environment for your applications. Rootkit in particular is a malicious software or program intended to enable unauthorized access to your compute platform and often tries to hide its existence. There are various open-source tools available for detecting rootkits[12][38][41]. On a standard Linux box, rootkit checks involve various system forensics, including but not limited-Filesystem scan to check existence of known offending files or directories, scanning kernel symbol table, process/network scanning.

Scope of vulnerabilities exposure by rootkits is smaller than that for a VM or physical server. For example, there are certain kernel-mode rootkits developed as loadable kernel modules in Linux. Application containers do not generally have such high privileges (Linux capabilities) to load kernel modules. Therefore, containers are less susceptible to kernel-mode rootkits. Bootkits are a variant of kernel-mode rootkits that infect startup code, like Master Boot Record (MBR), Volume Boot Record (VBR), or boot sector. Since container start up does not involve the traditional OS start up paradigm, these rootkits also become less relevant for containers.

Therefore, in the current capability of Malware detection, we focused on detecting file-based malware in container images and instances. A repository of known malwares and their corresponding offending file-paths is maintained by pulling their definitions from available open-source targets[41][12]. Then crawled file-paths from images and instances are validated against this repository to identify any potential malwares.

5 Experiences

In this section we discuss some experiences and lessons we learned while running *OpVis* in our public production cloud environment for over two years.

Visibility vs side-effects tradeoff. Different systems provide different levels of visibility; interestingly, more visibility requires more privileges and has higher chances of side effects (i.e., a data collector affecting the state during state collection).

This tradeoff becomes apparent with the two types of systems that we monitor: containers and VMs. Containers are easy to monitor as they share the kernel with the host. Everything that happens in the container (at the POSIX level) is visible from the host. This means that asking things like "what processes are running in a container?" can be easily answered by taking a look at the host kernel. On the other hand, VMs have their own kernel; thus, they keep their own state, forcing an agentless monitor to peek into the kernel, which is more difficult and requires introspection techniques.

The consequence of easy visibility is higher chances of leaving side effects. Our implementation presents some data points for this, when comparing containers to VMs. One such case is reading files. Typically, when reading a Unix file, the filesystem keeps track of the access time in the file metadata. Doing this from the host for a container leaves this access-time trail. On the other hand, doing this for a VM is not as easy; it requires disk introspection techniques previously discussed.

Choosing the right APIs for monitoring. A critical aspect related to monitoring is deciding what layer of the stack to look at. This affects the data to be obtained, but even if the data is the same, different layers may pose different challenges. One such challenge is the stability of the APIs.

One of the many crawler functions is to collect information from containers. Doing so can be achieved at different layers and with different methods. For example, if we were interested in collecting the CPU usage of a container, we can do it by using either Docker APIs, or *cgroups* in the host. We observed that the Docker APIs have been changing rapidly, compared to the *cgroup* APIs, which are provided by the Linux kernel. Using more stable monitoring APIs will require fewer changes to *OpVis* as a result of updates to the underlying systems.

Burst and sampling bias. The number of crawler processes created by the main thread can be one or more and is provided as an option. When multiple threads are created, their cumulative activity may result in spikes for CPU utilization. By staggering the activity of the threads, the overall consumption can be amortized leading to an average lowering of CPU activity.

Watching out for starvation. To monitor the crawler's behavior with respect to log collection, one of the things we did was to deploy on each cloud host a test container emitting log events at a low frequency: 2 log events per minute. We created a dashboard to verify whether and when the logs from our test containers were being indexed on the data store. We noticed that, occasionally, some containers generating logs at an extremely high frequency were deployed to a few hosts and, when that happened, our low-frequency test logs from those hosts were significantly delayed.

Our investigation revealed that the root cause was not in the data service; the indexers were working properly and there was no backlog of outstanding data on the data bus. To corroborate our suspicion of starvation caused by the crawler behavior, we experimented with throttling high-frequency logs, which indeed mitigated the problem.

Operational visibility systems need global, distributed admission-control policies in place to allow fair and timely visibility into all systems across all monitored cloud runtimes.

6 Contribution

OpVis has made both internal (production cloud department) and external (github) contributions. OpVis crawler, with python-based host and container inspection and VM introspection capabilities, has been opensourced [18] for over an year now, and has seen contributions from 19 developers internationally. The latest wave

[1] An example of benign update is an honest fix of SSH-related violations, albeit breaking container immutability principles

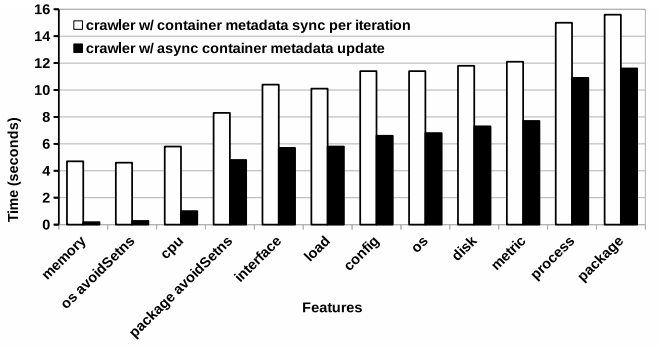

Figure 6. Time to crawl different features for 200 containers.

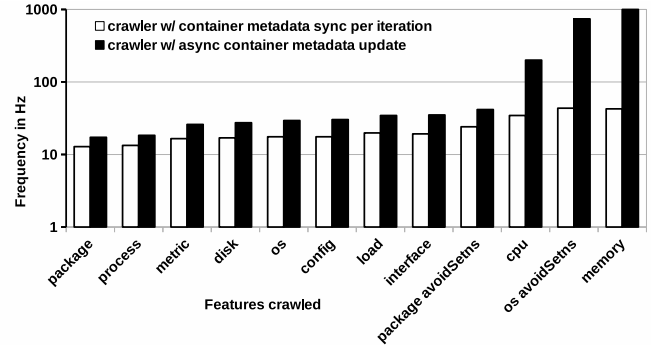

Figure 7. Effective crawl frequency per container.

of commits has been after the crawler restructuring to enable a plugin-based extensibility.

To maintain stability with an active multi developer contribution, we have integrated our github repository with Travis CI. On each code Pull Request (PR), over 250 unit and functional tests are fired, and tested across 5 docker versions. 90% code coverage is enforced, together with `pylint`, pep and `flake`-based style enforcement (python code-style tools). A PR merge triggers an automated Docker image build available publicly at DockerHub (further enabling continuous delivery via webhooks).

THe backend is closed-sourced and offers analytics services to customers of our production container clouds. The full OpVis pipeline (crawler + backend) has been active in our production cloud for over two years now [2]. VM monitoring service was previously part of our Openstack cloud deployment that supported over 1000 OS versions. Multiple instances of the crawler alone have been deployed as data collectors for our other internal departments.

7 Evaluation

In this section we evaluate the efficiency and scalability of the *OpVis* framework. We first showcase the monitoring frequency that can be realized with our out-of-band crawler. Next, we compare it with agent-based in-band monitoring in terms of performance impact on guest workload. Given a disaggregated processing model of *OpVis*. we also measure space and network overhead for data curation between crawler and annotators. Finally, we demonstrate the feasibility of *OpVis*' log streaming in terms of being able to successfully process production-scale log events. We focus our experiments on container clouds, with the corresponding benefits for VM-based cloud deployments having being proven in our previous work [53]. The latter also features other evaluation metrics including high state extraction accuracy, frame storage scalability as well as supremacy over in-guest monitoring by virtue of holistic knowledge (in-guest + host-level resource use measures).

Setup: The host machines have 16 Intel Xeon E5520 (2.27GHz) cores, and 64G memory. The host runs CentOS 7, Linux kernel 3.10.0-514.6.1.el7.x86_64, and Docker version 1.13.1. The guest containers are created from the Apache httpd-2.4.25 DockerHub image.

7.1 Monitoring latency and frequency

In this experiment, we measure the maximum monitoring frequency with which the crawler can extract state from the guest containers. Figure 6 shows the time it takes to extract different runtime state elements (features) from 200 webserver containers. Shown are two sets of bars for each feature representing two modes of crawler operation. The left bar represents the case where the crawler synchronizes with the docker daemon on every monitoring iteration to get metadata for each container, whereas the right bar represents the optimization where the crawler caches the container metadata after the first iteration, and subscribes to docker events to update its cached metadata asynchronously based upon container creation or deletion events. The optimization yields an average improvement of 4.4s to crawl 200 containers.

Another point to note is the improvement in crawl times when namespace jumps are avoided as can be seen in Figure 6 . For example, 11.6s vs 4.8s to crawl packages with and without namespace jumping respectively. Finally, while the crawl latencies shown in Figure 6 are for extracting individual features, as we also verified by experiments, crawl times for feature combinations can be calculated by adding the individual components together. For example, for the performance impact experiments in the next subsection, we simultaneously enabled the CPU, memory and package crawler plugins yielding a combined base crawl latency of 6s (~ 0.18 + 1 + 4.8s for respective plugins).

Scaling these crawl latency numbers across 200 containers then yields the effective monitoring frequency per container. As can be seen in Figure 7 , the crawler is easily able to support over 10Hz of monitoring frequency per container. This is with a single crawler process consuming a single CPU core (for many feature plugins, actually only 70% of a core, with time spent waiting either for (i) crawling the containers' *rootfs* from disk, or (ii) the kernel while reading cgroups stats and/or during namespace jumping).

7.2 Performance impact

In this experiment, we measure the impact of monitoring the guests with our *OpVis* agentless crawler, and compare it with the impact of agent-based in-guest monitoring. We ran 200 webserver containers on the host, and configured the crawler to extract CPU, memory and package information from each container. The idea being that the first two plugins provide resource use metrics data, while the

[2]We cannot divulge the actual scale of the endpoints scanned/processed by the OpVis pipeline

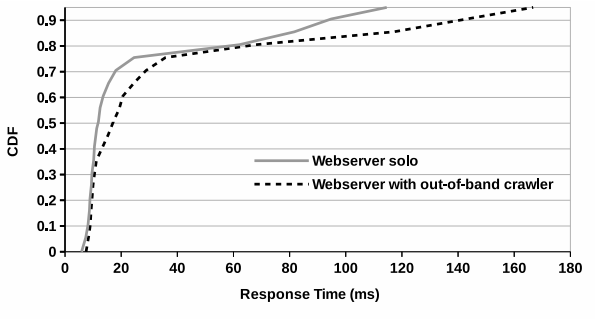

Figure 8. CDF of webserver response times.

third plugin allows periodic vulnerability analysis for the guests, as described in Section 4.1.

Setup: Each container was given equal CPU shares and 256 MB memory. The http containers were pinned to 10 cores on the host, while pinning the docker daemon and its helper processes (like docker-container shims, xfs/devicemapper etc.) to the other 6 cores to minimize interference. Further, various httpd and kernel parameters were tuned appropriately to enable high throughput operations, such as maximum open files, maximum processes, maximum available port range for connections, etc. The workload consisted of random 64-byte static files, requests for which were made to each webserver container from another host via httperf [37] workload generators (file size selected so as to avoid network bottleneck, verified by experiments). Throughputs and response times were recorded after a warm-up phase to ensure all data was brought in and served from memory in subsequent test runs, so as to put maximum stress on the system.

The base webserver capacity (maximum number of serviced requests per second without any connection drops) aggregated across all 200 webserver containers was observed to be 28000 requests/second (140 req/s per container), with an average response time of 30ms per request. With continuous out-of-band crawling, no impact is recorded on the sustainable request rate across the webserver containers. However, the average response time degrades by 50% to 45ms. Not all webserver containers see a hit in their response times, as can be seen in Figure 8 which plots the CDF of response times for the 200 containers with and without out-of-band monitoring. The crawl latency itself increases from 6s to 8s to extract CPU, memory and package information across all 200 containers, even though the crawler process is running on a separate core than the webserver containers.

To mimic an agent-based monitoring methodology, we next ran the crawler process inside each webserver container, configured to run after every 8s as per the above out-of-band crawling experiment. With such in-guest monitoring, the aggregate sustainable request rate sees a 14% hit with response times degrading by 65%.

In the in-guest monitoring mode, the monitor process competes for resources with the user workload, whereas in the agentless monitoring mode, the monitor process (the crawler) had its own dedicated core (taskset to run on one of the 6 cores not running the webserver containers, see 'setup' above). For completeness, we ran another experiment where the out-of-band crawler was restricted to use only the cores that were running the webserver containers. This still lead to a lower impact than in-guest monitoring in that

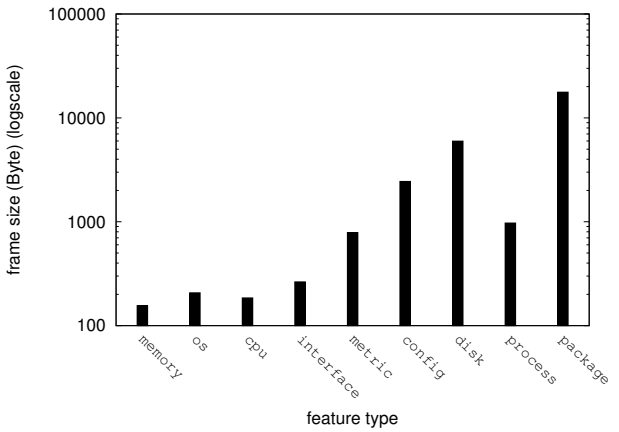

Figure 9. Size accounting for each crawled feature.

the webserver(s) base request rate could still be achieved but with a 75% response time degradation (up from 50% when the crawler had a dedicated core). Alternatively, base response times could be achieved but with a 7% lower sustainable request rate (still better than a 14% hit with in-guest monitoring, going as high as 21% to realize the base response times). *But, being able to use separate core is in fact desirable and indeed a benefit of* OpVis*' decoupled execution-monitoring framework, that enables offloading monitoring tasks outside the critical workflow path, thereby minimizing interference.*

7.3 Space and network overhead

Disaggregated delivery of analytics functions is one of the core features of *OpVis* enabled through separation of *data collection* by crawler and *data curation* by backend annotators. This also implies the need to transfer the data between these two processing endpoints which are typically separated by low-latency high-bandwidth local-area-network (LAN) connections. In this set of experiments our objective was to measure the overhead of transferring and storing crawled data. Fig.9 shows size of crawled data for each individual feature type from a single *httpd* container. During this experiment a more common emitter format, *json* was used and it is important to note that size overhead would differ for different emitter formats. This size overhead can further be reduced by using *data compression* which has its own performance implications of adding data curation latency during decompression.

7.4 Performance study of annotators

Annotators	Avg. "Vis" latency (sec)
Remote Login Check	1 - 15
Vulnerability Analyzer	1
Compliance Check	7

Table 1. Data curation latency measured at the annotator

From cloud users perspective, what is important is how quickly any non-conformity to security policies can be discovered for their hosted containers. In certain cases it depends on external factors,

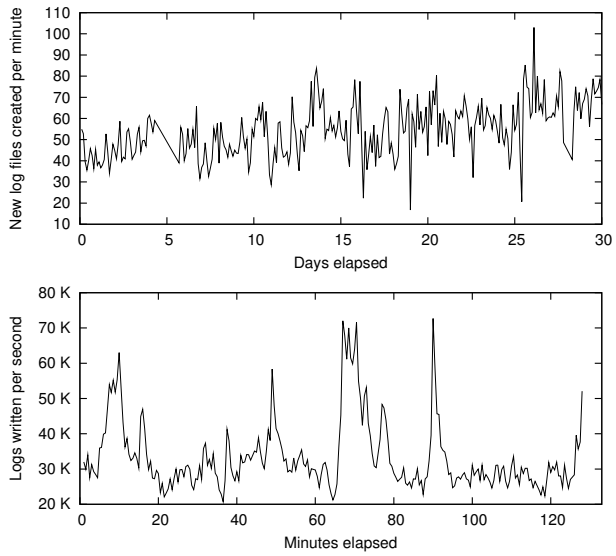

Figure 10. New log files per minute (top) and logs processed per second (bottom).

for instance, how quickly any newly-discovered package vulnerability is formally added to standard CVEs. In the *OpVis* context, we measure *vis* latency, which translates into the time taken by individual annotators to produce their respective verdict. Given that all annotators operate in parallel, their *vis* latencies are also overlapping.

In Table 1, we report average *vis* latencies for 3 *OpVis* applications. In the *Remote Login Check* application, all account passwords inside containers are checked for weaknesses. For containers with weak passwords, the security report is produced quickly (within a second), and for containers with strong passwords it can take up to 15 seconds. Intuitively, this aligns with the common security expectation wherein any potential violation is reported on priority. During *Compliance Check*, close to 21 standard security rules are validated within 7 seconds. Finally, for *Vulnerability Analyzer*, respective packages from containers are cross-checked with any known vulnerabilities, all within 1 second.

7.5 Log streaming in production

We now present data on the crawler's log-streaming behavior observed during a period of 1 month, while it was exercised by external users and internal core services of our production public cloud. Over this period, the crawler was tracking several hundred containers per host and thousands of containers, collecting approximately 250, 000 log events per minute per host on average. Figure 10 shows the actual data for one of our cloud hosts. The top plot shows the rate of new log files created per minute, exhibiting the level of dynamism and live activity in the cloud, with many new log files discovered every minute as instances come and go. It also shows a trend of increasing adoption and scale at the macro level as the month progresses. The bottom plot shows the number of log events processed per second.

8 Related Work

While Section 2 differentiates OpVis with existing operational visibility solutions for VMs and containers, here we discuss other work related to OpVis' VMI use, as well as systems analytics focus.

Memory Introspection Applications. Most previous VM introspection work focuses on the security and forensics domain. It is used by digital forensics investigators to get a VM memory snapshot to examine and inspect [14, 15, 24, 30, 47]. On the security side, VMI has been employed for kernel integrity monitoring [9, 33, 43], intrusion detection [29], anti-malware [11, 23, 28, 35, 42], firewall solutions [52], and information flow tracking for bolstering sensitive systems [31]. Outside the security domain, IBMon [46] comes closest to OpVis' approach, using memory introspection to estimate bandwidth resource use for VMM-bypass network devices. Other recent work employs VMI for information flow policy enforcement [8], application whitelisting [32], VM checkpointing [2], and memory deduplication [16].

Systems Analytics. While Section 4 discusses a few analytics applications developed over OpVis framework, others have explored several related systems analytics approaches. Examples include (i) Litty and Lie's out-of-VM patch auditing [36] that can detect execution of unpatched applications, (ii) CloudPD [51] cloud problem management framework which uses metrics correlations and to identify problems, (iii) EnCore [62] that detects misconfigurations by inferring rules from configuration files, (iv) DeltaSherlock [56] that discovers system changes via fingerprinting and machine learning methodologies, (v) PeerPressure [61] which identifies anomalous misconfigurations by statistically comparing registry entries with other systems running the same application. There exist quite a few other works in literature in the systems analytics domain. With the entire system state, ranging from metrics, to packages, to configuration files, exposed to the backend, OpVis provides a uniform and extensible framework to enable such analytics to be performed across-time and across-systems.

9 Conclusion

In this paper, we presented our unified monitoring and analytics framework- *OpVis*- to achieve operational visibility across the cloud. We described the various techniques employed to enable agentless extraction of volatile and persistent state across VM and container guests, without enforcing guest cooperation or causing guest intrusion, interference and modification. We emphasized the extensible nature of our framework enabling custom data collection as well as analysis. We described 4 of the analytics applications we've built atop the OpVis pipeline, which have been active in our public cloud for over 2 years. We highlighted OpVis' high monitoring efficiency and low impact on target guests, as well as presented production data to demonstrate its usability. We described our opensource contributions, as well as shared our experiences while supporting operational visibility for our cloud deployment.

10 Acknowledgments

Over the years, several researchers have contributed to the various *OpVis* components and hence deserve credit for this work. These include- Vasanth Bala, Todd Mummert, Darrell Reimer, Sastry Duri, Nilton Bila, Byungchul Tak, Salman Baset, Prabhakar Kudva, Ricardo Koller, James Doran.

References

[1] Understanding security implications of using containers in the cloud. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, 2017. USENIX Association.

[2] Ferrol Aderholdt, Fang Han, Stephen L. Scott, and Thomas Naughton. Efficient checkpointing of virtual machines using virtual machine introspection. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 414–423, May 2014.

[3] Amazon. Cloudwatch. http://aws.amazon.com/cloudwatch/.

[4] Amazon. Summary of the october 22,2012 aws service event in the us-east region. https://aws.amazon.com/message/680342/.

[5] Anchore. Open source tools for container security and compliance. https://anchore.com/.

[6] Apache Kafka. https://kafka.apache.org, 2016.

[7] Apache Lucene. https://lucene.apache.org, 2016.

[8] Mirza Basim Baig, Connor Fitzsimons, Suryanarayanan Balasubramanian, Radu Sion, and Donald E. Porter. CloudFlow: Cloud-wide Policy Enforcement Using Fast VM Introspection. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, IC2E '14, pages 159–164, 2014.

[9] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Trans. Dependable Secur. Comput.*, 8(5):670–684, September 2011.

[10] Banyan. Over 30% of official images in docker hub contain high priority security vulnerabilities. https://banyanops.com/blog/analyzing-docker-hub/.

[11] Antonio Bianchi, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[12] Michael Boelen and John Horne. The rootkit hunter project. http://rkhunter.sourceforge.net/.

[13] Google cadvisor. Analyzes resource usage and performance characteristics of running containers. https://github.com/google/cadvisor.

[14] Andrew Case, Andrew Cristina, Lodovico Marziale, Golden G. Richard, and Vassil Roussev. Face: Automated digital evidence discovery and correlation. *Digit. Investig.*, 5:S65–S75, September 2008.

[15] Andrew Case, Lodovico Marziale, and Golden G. RichardIII. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7, Supplement(0):S32 – S40, 2010.

[16] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory de-duplication and migration. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 51–62, New York, NY, USA, 2013. ACM.

[17] Clair. Automatic container vulnerability and security scanning for appc and docker. https://coreos.com/clair/docs/latest/.

[18] cloudviz. agentless-system-crawler: A tool to crawl systems like crawlers for the web. https://github.com/cloudviz/agentless-system-crawler.

[19] Collectd. The system statistics collection daemon. https://collectd.org/.

[20] Datadog. Modern monitoring and analytics. https://www.datadoghq.com/.

[21] Docker. Docker hub: Dev-test pipeline automation, 100,000+ free apps, public and private registries. https://hub.docker.com/.

[22] Docker. http://www.docker.com, 2016.

[23] B Dolan-Gavitt, B Payne, and W Lee. Leveraging forensic tools for virtual machine introspection. Technical Report GT-CS-11-05, Georgia Institute of Technology, 2011.

[24] Josiah Dykstra and Alan T. Sherman. Acquiring forensic evidence from infrastructure-as-a-service cloud computing: Exploring and evaluating tools, trust, and techniques. *Digital Investigation*, 9:S90–S98, 2012.

[25] Elastic. Elasticsearch. https://www.elastic.co/products/elasticsearch, 2016.

[26] Elastic. Logstash. https://www.elastic.co/products/logstash, 2016.

[27] Tenable Flawcheck. Expanding vulnerability management to container security. https://www.flawcheck.com/.

[28] Yangchun Fu and Zhiqiang Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *IEEE Security&Privacy'12*.

[29] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[30] Brian Hay and Kara Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008.

[31] J. Hizver and Tzi cker Chiueh. Automated discovery of credit card data flow for pci dss compliance. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 51–58, Oct 2011.

[32] Jennia Hizver and Tzi-cker Chiueh. Real-time deep virtual machine introspection and its applications. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 3–14, New York, NY, USA, 2014. ACM.

[33] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with OSck. In *ASPLOS*, pages 279–290, 2011.

[34] VMWare Inc. Vmware vmsafe security technology. http://www.vmware.com/company/news/releases/vmsafe_vmworld.html.

[35] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In *CCS '07*, pages 128–138.

[36] Lionel Litty and David Lie. Patch auditing in infrastructure as a service clouds. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 145–156, New York, NY, USA, 2011. ACM.

[37] David Mosberger and Tai Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[38] Nelson Murilo and Klaus Steding-Jessen. chkrootkit: locally checks for signs of a rootkit. http://www.chkrootkit.org/.

[39] NIST. Security content automation protocol (scap) validation program. https://scap.nist.gov/validation/.

[40] OpenSCAP. Security compliance. https://scap.nist.gov/validation/.

[41] OSSEC. Open source hids security. https://ossec.github.io/.

[42] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 233–247, 2008.

[43] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[44] Prometheus. Monitoring system & time series database. https://prometheus.io/.

[45] Dell Quest/VKernel. Foglight for virtualization. http://www.quest.com/foglight-for-virtualization-enterprise-edition/.

[46] Adit Ranadive, Ada Gavrilovska, and Karsten Schwan. Ibmon: monitoring vmm-bypass capable infiniband devices using memory introspection. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 25–32, 2009.

[47] Andreas Schuster. Searching for processes and threads in microsoft windows memory dumps. *Digit. Investig.*, 3:10–16, September 2006.

[48] Aqua Security. Container security - docker, kubernetes, openshift, mesos. https://www.aquasec.com/.

[49] Tenable Network Security. Nessus vulnerability scanner. http://www.tenable.com/products/nessus-vulnerability-scanner.

[50] Sensu. Full-stack monitoring for today's business. https://sensuapp.org/.

[51] Bikash Sharma, Praveen Jayachandran, Akshat Verma, and Chita R Das. Cloudpd: Problem determination and diagnosis in shared dynamic clouds. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[52] Abhinav Srivastava and Jonathon Giffin. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 39–58, 2008.

[53] Sahil Suneja, Canturk Isci, Vasanth Bala, Eyal de Lara, and Todd Mummert. Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 249–261, New York, NY, USA, 2014. ACM.

[54] Sysdig. Docker monitoring, kubernetes monitoring & more. https://sysdig.com/.

[55] Nikolai Tschacher. Typosquatting programming language package managers. http://incolumitas.com/2016/06/08/typosquatting-package-managers/.

[56] Ata Turk, Hao Chen, Anthony Byrne, John Knollmeyer, Sastry S Duri, Canturk Isci, and Ayse K Coskun. Deltasherlock: Identifying changes in the cloud. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 763–772. IEEE, 2016.

[57] Twistlock. Docker security & container security platform. https://www.twistlock.com/.

[58] VMware. Vmci overview. http://pubs.vmware.com/vmci-sdk/.

[59] VMware. Vmware tools. http://kb.vmware.com/kb/340.

[60] VMware. vshield endpoint. http://www.vmware.com/products/vsphere/features-endpoint.

[61] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, volume 4, pages 245–257, 2004.

[62] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. *ACM SIGPLAN Notices*, 49(4):687–700, 2014.

[63] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. Automatic configuration of internet services. *SIGOPS Oper. Syst. Rev.*, 41(3):219–229, March 2007.