2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

StorM: Enabling Tenant-Defined Cloud Storage Middle-Box Services

Hui Lu†, Abhinav Srivastava‡, Brendan Saltaformaggio†, Dongyan Xu†
†Department of Computer Science, Purdue University, ‡AT&T Research
†{lu220, bsaltafo, dxu}@cs.purdue.edu,‡abhinav@research.att.com

***Abstract*—In an Infrastructure-as-a-Service cloud, tenants rely on the cloud provider to provide "value-added" services such as data security and reliability. However, this provider-controlled service model is less flexible and cannot be customized to meet individual tenants' needs. In this paper, we present StorM, a novel middle-box service platform that allows each tenant to deploy tenant-specific security and reliability services – in virtualized middle-boxes – for their cloud data. With such middle-boxes, StorM divides the responsibilities of service creation between tenants and the provider by allowing tenants to customize their own cloud data polices and the provider to offer corresponding infrastructural support. In developing StorM, we address key challenges including network splicing, platform efficiency, and semantic gap. We implement a StorM prototype on top of OpenStack and demonstrate three tenant-defined security/reliability middle-box services, with low performance overhead (< 10%).**

***Keywords*-Cloud Computing; Cloud Storage; Middle-box; OpenStack**

I. Introduction

Infrastructure-as-a-Service (IaaS) clouds (e.g., Amazon EC2) offer "one size fits all" virtual hardware platforms for tenants to deploy their applications in a scalable, cost-effective manner. However, providing all guests the same, non-customizable set of resources (i.e., virtualized block volumes, CPU, and networking) deprives tenants of any control over their in-cloud data. As a result, tenants have to rely on cloud service providers (CSPs) to implement and support each and every cloud data security/reliability service, such as access control, encryption/decryption, and fault tolerance.

Unfortunately, the sharp division of administrative domains in IaaS clouds – tenants can only administer virtual machines (VMs) while relying on CSPs to provide services to their cloud data – raises several problems for tenants. For example, concerns over the security and privacy of the underlying server-side resources have made consumers hesitant to move to public clouds [1], [2]. Sensitive files and proprietary code, stored in the cloud may be leaked, and since tenants are at the mercy of the services offered by CSPs (e.g., full disk encryption), the tenants cannot further enhance the security and reliability of their data.

One (seemingly straightforward) solution may be to deploy such services (e.g., data encryption, access control) by tenants in each of their VMs. However, despite the possibility of doing so, there are limitations in such an "end-point" based approach, which pushes the service deployment and maintenance burden back to the tenants. For example, the same set of services need to be deployed and managed in each VM – by the tenant. Further, such services inevitably compete for resources with the tenant's primary workloads running in the same VM, causing nontrivial performance impact. A recent survey of 500 CIOs at medium-scale organizations confirms that 76% of management issues, such as missing SLA and increasing costs, are due to software agents *running inside guest VMs* [3, Part II]. As a result, organizations tend to favor non-intrusive (relative to the production VMs) but customizable approaches to cloud data management – to achieve both high efficiency and security/reliability.

To this end, we present **StorM**, a novel *middle-box* platform for deploying tenant-defined security/reliability services for the tenant's in-cloud data storage. StorM is inspired by the recent trend in networking research to employ middle-boxes for network security, agility, and scalability [4]–[7]. Specifically, StorM provides virtualized middle-boxes, which operate between tenant VMs and the cloud storage infrastructure. The service logic (e.g., access logging, encryption, and replication) of a middle-box is defined by the tenant but the creation and operation of the middle-box is taken care of by the cloud provider. StorM offers infrastructure-level support to deploy *isolated yet dependable* middle-box services on behalf of tenants, which run *transparently* to both in-VM workloads and back-end cloud storage.

Development of StorM is complicated by several challenges faced by existing cloud platforms. (1) A CSP maintains two separate networks – one for the tenant VM traffic known as the *instance network* and the other for the storage traffic called the *storage network*. The isolation prevents middle-boxes, which must access the instance network to communicate with clients, from operating on the storage data flowing through the storage network. We solve this problem by designing a novel *network splicing* approach to allow middle-boxes to securely process storage traffic without breaking isolation. (2) Middle-box-based approaches require data redirection, which incurs end-to-end latency. To avoid performance degradation, StorM introduces a novel *active-relay* system which allows storage network packets to be acknowledged by middle-boxes – short-cutting the path to storage servers and allowing the data source to continuously send data packets without delay. (3) Tenant VMs operate

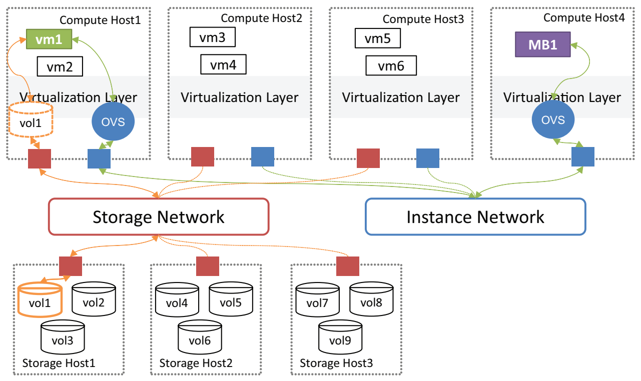

Figure 1: A typical IaaS cloud architecture with separate storage and instance networks.

at a file and directory granularity while middle-boxes must intercept storage network packets (e.g., iSCSI) that carry low-level storage system information such as raw data blocks or metadata blocks. To provide meaningful monitoring and control services, StorM bridges such a *semantic gap* by enabling cloud services to construct high-level file structures *on-the-fly* using each file's metadata accesses.

We have implemented a StorM platform prototype on top of OpenStack. While its current design is tailored for block storage, StorM is equally applicable to other storage systems such as object storage. We have also built three security/reliability middle-box services to demonstrate StorM's practicality: (1) on-the-fly encryption/decryption, (2) storage access monitoring, and (3) data replication. These services reflect a tenant's specific security/reliability requirements, based on the cloud's existing virtualization infrastructure. Our evaluation shows that StorM successfully solves the three challenges in the previous paragraph, while incurring negligible overhead on the storage traffic ($< 10\%$ in all cases). In summary, this paper makes the following contributions:

- We identify and highlight the "one size fits all" dilemma in current cloud systems, and motivate the need for value-added, tenant-specific security/reliability services provided by virtualized middle-boxes.
- We design and develop StorM to enable virtualized storage middle-boxes. StorM addresses three key challenges: *network splicing*, *platform efficiency*, and *semantic gap*.
- We have implemented a StorM prototype on top of OpenStack. We have also deployed three tenant-customized security/reliability services showing the efficacy of StorM.
- Our evaluation shows that StorM is capable of supporting storage security and reliability functions with low overhead.

II. Overview

A. Background

In a typical IaaS cloud architecture, such as OpenStack, large pools of computing, storage, and networking resources are controlled through a cloud controller. On-demand computing resources are provided to tenants by provisioning VMs with configurable CPU, memory, and disk sizes.

Networking is a critical component in cloud infrastructures. In Figure 1, we illustrate the most common datacenter networking setup: an instance network and a storage network. The most complex network architecture lies in the instance network, including advanced network technologies such as software-defined networking (SDN) using Open vSwitch [8], virtual LAN, tunneling technologies, and numerous network security features to allow multi-tenancy, isolation, and security. In contrast, the storage network is relatively simple and usually separated from the instance network, providing basic connectivity which carries storage I/O traffic originating from tenant VMs to the storage servers.

Cloud storage services, such as block storage and object storage, are provided over the network. Object storage (e.g., provided by Amazon S3 and OpenStack Swift) is ideal for cost effective, scale-out storage, and normally used for backup, archiving, and data retention. Block storage (e.g., Amazon EBS, OpenStack Cinder) offers raw volumes which are used as VMs' disks and allow tenants to configure them with the file systems of their choice. In this paper, we mainly focus on block storage, however, StorM's design is equally applicable to object storage.

B. Opportunities and Challenges

The goal of StorM is to support *the deployment of tenant-defined storage security/reliability services via virtualized middle-boxes*. The proposed approach has several benefits over the traditional methods of offering such services, i.e., either installed inside tenant VMs or one-size-fits-all services by cloud providers.

First, StorM enables tenants to implement and customize many services, such as data management, maintenance, and protection, in virtualized middle-boxes without any modifications to their production VMs. Second, StorM-supported middle-boxes are themselves minimal VMs running tenant-specific service programs, and thus these middle-boxes avail the same benefits as conventional VMs. For example, middle-boxes can be dynamically provisioned with more memory, CPU, or even hardware acceleration support. More importantly, StorM facilitates the chaining of services offered by middle-boxes to create a service bundle that would be impossible to achieve via traditional service methods. For example: a tenant concerned about data security and audit logging can request both storage monitoring and encryption service middle-boxes. StorM chains these middle-boxes so that after the storage monitor records the I/O access, the data is passed through the encryption box to encrypt the data written to disk. These services, like VMs, can be scaled up and down, depending upon the traffic load, making them truly elastic. Further, advanced network technologies provided in the VM network are leveraged to protect and isolate storage middle-boxes, belonging to different tenants.

Finally, StorM moves such services out of the tenant's VMs, and by doing so reduces performance overhead due to interference between services and the foreground applications. Our experiments show that *even with sufficient computing resources*, the performance of foreground applications could be significantly affected by "value-added" services, such as encryption/decryption, running in the same VM, confirming the findings of one recent survey [3, Part II]. Our evaluation (Section V) shows that moving these tasks into a dedicated middle-box greatly mitigates such impact.

To provide any meaningful service to tenants, middle-boxes need to access the storage network's traffic. However, the storage network and instance network, as shown in Figure 1, are designed to be isolated. Storage packets from the VMs are immediately put on the storage network, preventing them from being received by middle-box VMs on the instance network. The first challenge of StorM is to splice these two networks and forward storage traffic along a specific path across these two networks. To address this challenge, the StorM platform creates a new forwarding plane to route storage traffic via the middle-boxes using both conventional and SDN-based flow steering techniques. Further, by taking advantage of SDN-enabled networking, StorM provides *on-demand middle-box service scaling* by dynamically adding or removing middle-boxes on the storage traffic path by programming SDN switches.

Tenant-defined middle-box implementations depend upon an API to retrieve block-level I/O data and operations from the received storage packets and present them to services for processing. Afterwards, "manipulated" data packets are forwarded to the next-hop. Such data processing and packet rerouting introduces latency that threatens to offset the benefits of StorM. Therefore, the second challenge of StorM is to reduce this overhead. StorM uses a multi-threaded, high throughput design and also develops a new technique called active-relay that further avoids data processing and forwarding delays. The key idea enabling this approach is that instead of naively relaying packets to their next-hop, the middle-box immediately acknowledges the data source, which could be a tenant VM or another middle-box, once the data packets are received. By doing so, the following packets originating from the data source will not be delayed by the subsequent data processing and next-hop forwarding inside the middle-box. This further separates the production VM's operation from the middle-boxs' because the VM observes its storage packets being handled via a single storage subsystem. Our performance evaluation results show

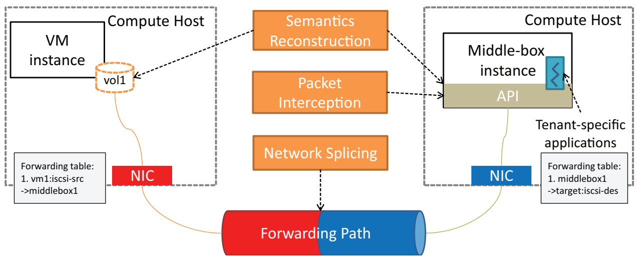

Figure 2: Architecture of StorM.

that such improvements significantly reduce the latency of the storage traffic.

Lastly, the storage packets going into middle-boxes only carry block-level information such as disk sectors, raw data blocks, and inode information. Middle-boxes offering encryption/decryption can work at this granularity, however other services such as access monitoring and intrusion detection require higher-level file semantics which are present in the packet. Storage middle-boxes take advantage of inherent storage subsystem information such as file system types and disk layouts to accurately reconstruct the mapping from low-level block accesses to high-level file operations. Our case study in Section V-B implements a storage monitoring middle-box and demonstrates that this mapping not only recovers the high-level file operations, but also provides more details of the file system internal operations which can be used to analyze suspicious behaviors in the file system.

C. Assumptions

A cloud infrastructure (e.g., OpenStack) typically consists of two types of nodes – controller nodes and compute nodes. Like most efforts in this space [9], we assume that cloud providers (including the controller and physical cloud infrastructure) are trusted. Since the StorM components reside on the physical cloud infrastructure, they are also trusted and protected by cloud providers. Once middle-boxes are deployed, their network connectivity is isolated to only the storage network (controlled by CSPs) and connections from local CSP administrators; and they are transparent to the programs (including malicious programs) running in the tenants' VMs. Finally, the tenants' VMs, which are facing the external network, are not trusted. Attackers may compromise a tenant's VM, and subsequently try to access its in-cloud data storage.

III. Design

Figure 2 shows the three main components of StorM. The first component, network splicing, is part of the cloud infrastructure, and enables the seamless and transparent deployment of virtualized middle-boxes and ensures the isolation and security of any "cross-network" traffic. The last two components, packet interception and semantics reconstruction, are offered as an API to middle-boxes to intercept the storage network traffic from inside the middle-box, and then interpret the iSCSI packets (if required) to build higher-level views from the low-level data. These high-level semantics are required to implement middle-box services such as storage access monitors.

A. Network Splicing

The storage network and instance network, as shown in Figure 1, are usually isolated in cloud deployments. While this design keeps the storage networking simple, it prevents the storage traffic from using many common networking services. To allow storage traffic to benefit from storage services running inside VM middle-boxes, the storage traffic must be selectively brought into the instance network, where it can be tunneled through the middle-boxes. To solve this problem, we develop a novel solution that splices both storage and instance network. We break the network splicing problem into three sub problems – a) connection attribution, b) selective flow routing from storage to instance network, and c) steering flows through middle-boxes, and describe our solutions below:

Connection Attribution The StorM platform allows tenants to selectively decide which VM's storage traffic should be routed through specific middle-boxes, offering various services such as monitoring, encryption/decryption, and replication. Using each tenants' high-level routing policies, the StorM platform automatically determines which flows should be steered through middle-boxes. To be able to offer fine-grained selection of flows, StorM has to solve the connection attribution problem.

Connection attribution refers to the process of automatically identifying which VM is attached to which persistent storage connection and sending VM I/O data to the storage server. On the tenants' side, the connection attribution allows tenants to specify fine-grained routing policies, e.g., asking for middle-box services for a few select VMs or shared among all VMs. On the provider side, it allows StorM to distinguish one tenant's storage traffic from the other tenants and provide isolation and security to the storage traffic.

However, achieving this on-demand network splicing is difficult due to the way cloud systems such as OpenStack set up storage connections. Cloud systems typically use the iSCSI protocol to communicate between storage clients (called iSCSI initiator) and the server (iSCSI target). Since an iSCSI initiator runs on the host (compute node) instead of tenant VMs, the connection information bears only the host IP and port and destination IP and port, obfuscating the VM details attached to the connection. The mapping of the 4 tuples to the actual VM owning that storage connection information is buried deep inside the iSCSI implementation.

To solve this problem, StorM first gathers the information about the virtual block devices (also known as IQN numbers) that are attached to a tenant VM. This information is stored in the hypervisor. This allows StorM to know which VM is attached to which virtual device. Another mapping exists in the system that glues virtual devices to specific storage traffic source ports. To collect this information, StorM modifies the source code of storage connection software to expose the port number along with the IQN number. In particular, we modified the iSCSI "Login Session" code to expose TCP connection information [10]. These series of mappings enable StorM to identify each storage connection, allowing it to offer fine-grained tenant policies.

Storage to Instance Network Once we identify the storage connections or flows to be routed through middle-boxes, StorM must selectively bring those flows into the instance network, where storage middle-boxes can process the flows. When the middle-boxes are finished processing, StorM moves the flows back to the storage network to go to the storage server. At the surface, this may appear like a typical routing problem, however in practice, StorM must overcome several constraints of cloud networking while guaranteeing the isolation and security of this traffic.

StorM creates a pair of *storage gateways* as the ingress and egress points for a tenant's storage traffic inside the instance network. One storage gateway selectively sends IP-based storage traffic from the storage network to the instance network and the other vice-versa. To ensure the isolation and security of this traffic, these two gateways are created in a tenant's network space (a virtual isolated network domain reserved to a tenant), and are thus invisible to the outside world. StorM operates entirely within a tenant's isolated virtual network. Hence a trusted cloud provider will apply existing network isolation techniques (e.g., namespace, security groups) to secure the tenant's virtual network (inaccessible to eavesdroppers without compromising the hypervisor).

StorM provides storage traffic redirection – from a VM's storage traffic node to the ingress gateway, or from the egress gateway to the storage node – via the conventional network address translation (NAT) rules. StorM sets up these NAT rules on the host running each tenant VM and the host running the storage gateways. Since the storage network and instance network are completely isolated, during redirection, StorM prevents the exposure of IP addresses inside the storage network from appearing in the instance network. To do so, StorM applies a series of IP masquerading rules both at the ingress and egress gateways to make the IP addresses consistent. On the ingress gateway, the source IP address of storage traffic is translated to the ingress gateway's IP address via IP masquerading, while the destination IP address is translated to the egress gateway's IP address. As a result, the middle-box VMs will only see the redirected storage traffic coming from the ingress gateway and going to the egress gateway (or the reverse).

The storage gateways can be created on any compute nodes as long as these nodes have both storage and instance

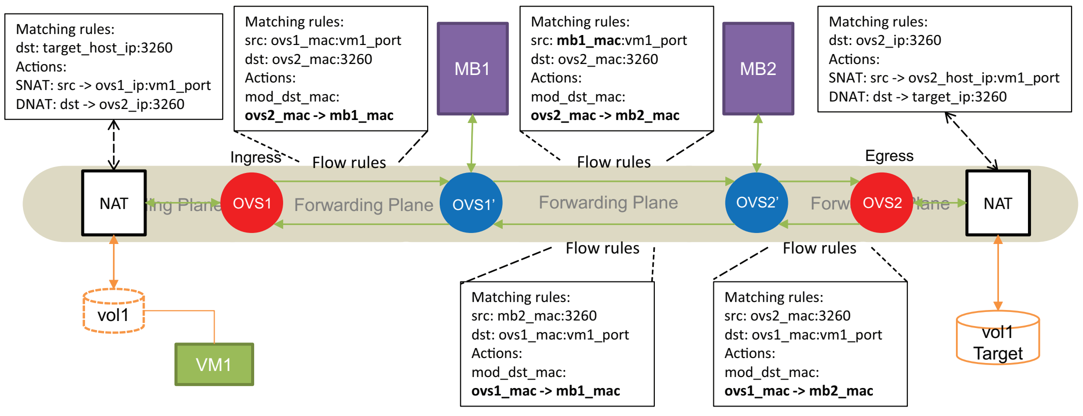

Figure 3: An example of StorM's new forwarding plane.

network interfaces installed. To reduce the routing latency, it is preferable to place the ingress gateway physically close to the VM's storage traffic node, and the egress gateway close to the storage server node.

Notice that, for all VMs on a single host, 3 fields out of the 4-tuples of the iSCSI TCP connection are the same (except the source port). Connection attribution determines which VM is bound to which iSCSI TCP connection established on the host. However, the source port field can only be known after the connection is established. Since StorM applies NAT rules to route flows towards the storage gateway, NAT rules which match only the 3-tuples will route all storage flows on the host towards the storage gateway.

To solve this problem, StorM introduces an *atomic attachment operation* to VM storage volumes using storage middle-boxes: Before attaching a volume, StorM installs the NAT rules on the VM's host; thus during the volume attachment operation, the very first (and the subsequent) packets of that storage connection follow the established NAT rules. After the connection is established, StorM removes the NAT rules to ensure that they will not influence any following volume attachments. Note that the removal of NAT rules does not impact established flows, which still follow their existing NAT rules (if any). StorM uses a mutex lock to ensure the atomic operations across all above steps. As volume attachment (and detachment) occurs less frequently on a compute node and is fast to finish, this mutex lock has little impact on the overall performance of the system.

SDN-enabled Flow Steering After passing through the storage gateways, storage traffic flows traverse one or more middle-boxes selected by the tenant, offering various storage services. StorM's job is to steer the flows through middle-boxes and provide on-demand services which can dynamically add or remove middle-boxes for an existing storage traffic flow. To this end, StorM designs and develops an *SDN-based flow steering method* on the instance network. StorM relies on a centralized SDN controller that controls a set of virtual switches, to which middle-box VMs are connected. These virtual switches as well as their associated middle-boxes constitute the forwarding chain of a storage traffic flow. StorM dynamically configures the forwarding chain by inserting flow rules in the corresponding SDN-enabled virtual switches that match the storage traffic 4-tuples. No additional configurations are required in the middle-boxes except enabling the IP forwarding function.

As shown in Figure 3, the basic forwarding unit of the chain consists of three components: the middle-box, its previous hop (a virtual switch, where storage traffic comes from), and the next hop (another virtual switch, where storage traffic goes after leaving the middle-box). For example: for a flow that has to traverse through two middle-boxes MB1 and MB2, the first chain {OVS1, MB1, OVS1′} brings the flows to MB1 and the second chain {OVS1′, MB2, OVS2′} takes it to MB2 and then to the egress gateway. Note that the OVS1′ is the destination hop in the first chain, but becomes the source hop in the second chain – this is how StorM combines chains to setup arbitrary network paths. The reverse path follows the same idea. By installing these flow rules in the SDN-enabled virtual switches, StorM can route storage traffic to any number of virtualized storage middle-boxes.

B. An Efficient API

With the help of network splicing, StorM brings the storage traffic to the storage middle-boxes. At this point, the middle-boxes have to intercept and process these flows to offer various services. To this end, StorM designs highly efficient packet interception and interpretation APIs. These APIs provide meaningful storage I/O data to services executing inside middle-boxes while avoiding as much overhead as possible.

Since storage flows passing through middle-boxes do not target any local processes inside the middle-box, their packets are directly sent to the FORWARD chain of the middle-box kernel to be transmitted on the outgoing network interface. To retrieve these packets, an intuitive but costly approach is to setup a hook (with a callback function) along the packets' kernel path. We refer to this method as the *passive-relay* approach.

Though simple, the passive-relay approach leads to non-trivial overhead in intercepting flows, especially for flows with heavy I/O loads. This is due to the frequent system calls to copy packet data from the kernel to user mode – one per packet. To optimize this process, StorM proposes an *active-relay* approach. Instead of simply relaying received data to the next hop, the active-relay approach acknowledges the data source actively.

Essentially, the active-relay approach breaks the original single TCP connection into two: one connects the middle-box to the ingress gateway, and the other connects the middle-box to the egress gateway. Any received packets of one connection are acknowledged immediately by the middle-box, and then sent to the other connection for the next-hop forwarding after the packets have been processed by the storage services. By doing so, the active-relay approach shortens the packet acknowledgment path, and thus accelerates the packet transmission rate, which is not delayed by the subsequent data processing and forwarding.

As the active-reply breaks one connection into two, state inconsistency between them could happen if packets are not properly delivered by any one of the connections. We solve this problem by storing a copy of the received packets in the middle-box's *non-volatile* memory, until the packets are delivered and acknowledged by their next hop(s). Some situations (TCP connection failure, storage VM crash, and storage media error) could also result in a data transmission failure. While such failures may impact a tenant VM's storage service, they are handled by existing fault-tolerance techniques readily deployable alongside StorM.

To establish the connection from the ingress gateway to the middle-box, NAT rules are deployed in the middle-box to redirect storage traffic flows (coming from the ingress gateway) to a local port, where the *pseudo-server process* listens. The TCP connection from the middle-box to the egress gateway is created by the *pseudo-client process*, which simply connects to the egress gateway with the corresponding destination IP and port. Both processes provide iSCSI parsing logic, and *read and write interfaces* to the storage service processes. Note that the active-relay approach leverages the kernel's TCP stack for data copying between the kernel and the user space. This approach is more efficient, because the TCP handler packs several packets together for each copy.

C. Semantics Reconstruction

Tenant VMs operate at a granularity of files and directories, but middle-boxes which are processing raw storage data packets (e.g., iSCSI) can only observe low-level information such as disk sectors, blocks, and inodes information. However, many middle-boxes offering services such as monitoring and IDS require access to higher-level views for their operation. This is an instance of the semantic gap problem, and thus StorM must reconstruct the high-level file structures using the file metadata present in the storage packets.

StorM generates an initial high-level system view of a file-system and supplies it to the middle-boxes when the block device is attached to its tenant VM. This system view describes the organization of the specific file system (e.g., Ext4 or NTFS) on that disk, including the layout of metadata and raw data and the mappings between directories and files to their data locations. Metadata accesses, such as file creation, deletion, and renaming, may update this high-level system view. By keeping track of metadata accesses, StorM is able to maintain an up-to-date file system view. This allows the middle-boxes to convert low-level data accesses into high-level file operations, which is essential for fine-grained data reliability services, such as the data replication middle-box developed in Section V-B3.

D. Policies

StorM's high-level policies allow tenants to deploy virtualized storage middle-boxes in a highly-customizable manner. The following policies must be specified by tenants prior to using middle-boxes: (1) which VMs and their associated volumes will use the middle-box services, (2) the middle-boxes' storage service types and virtual resources (e.g., vCPU number and memory size), and (3) the organization of these middle-boxes (i.e., how middle-box VMs are chained for each volume).

StorM provides an interface for tenants to submit these policies to the cloud provider, and the StorM platform, accordingly, parses the policies and deploys the middle-box services. Specifically, the platform first provisions the required middle-box VMs on the compute hosts with the specified VM templates. Then, the platform retrieves the connection attributions for each volume and generates and installs the forwarding rules according to the organization specification. Lastly, StorM connects the volumes to their VMs with the middle-box services enabled.

IV. Implementation

We have implemented a prototype of StorM (~10,000 LOC) on top of OpenStack Icehouse. The new forwarding service consists of a centralized SDN controller and monitors installed on each compute host. The SDN controller first gathers related information (e.g., connection attribution), and then generates the flow rules for the virtual switches on the flow path. Next, the flow rules are installed in the virtual switches of the corresponding compute nodes by the monitors.

The packet interception API runs inside each storage middle-box. We have implemented the active-relay approach for StorM's highly efficient API. The active-relay leverages Linux's SCSI target framework to implement the pseudo-server process. The pseudo-client process is built with the help of the Open-iSCSI framework. The iSCSI parsing logic of Open-iSCSI is reused to decapsulate and encapsulate iSCSI packets, when either process receives or sends iSCSI packets.

We have implemented StorM's semantics reconstruction functions for Linux Ext series (2, 3, and 4) file systems on iSCSI-based storage networking. StorM uses Linux's *dumpefs2* tool to construct an initial file-system view. The reconstruction function updates the mappings between the low-level data blocks and its high-level files using metadata accesses, thus maintaining an up-to-date system view. The mappings are further extracted and stored in a Hash table for fast searching, such as in IDS or monitoring services.

V. Evaluation and Case Studies

We have deployed StorM in a cloud testbed based on OpenStack. Our test cloud cluster contained 10 physical machines each with two Intel Xeon quad-core processors and 32 GB memory. Each machine was installed with two 1 Gigabit Ethernet cards, connecting to the storage network and instance network, separately. We used OpenStack Icehouse and deployed compute services as well as StorM on each physical machine and one block storage volume service (OpenStack's Cinder) on one of the physical hosts. A 1 TB SATA disk was used for creating a physical volume, and multiple volume groups were created from the physical volume through OpenStack's Cinder service. The instance network used GRE tunneling for inter-host VM traffic, and tenant networking was provided by OpenStack's Neutron service.

A. Performance Evaluation

The goal of our performance evaluation is to determine how much overhead (both latency and throughput overhead) StorM incurs on the storage traffic. This would be caused by the extra level of indirection StorM has introduced on the tenant's storage traffic and the data processing (for security and reliability) within middle-boxes. To measure the overhead of StorM, we used the I/O micro-benchmark called Fio [11]. Fio generates and measures a variety of file and block operations, and it can vary both I/O request sizes (the amount of data read/written in a given transaction) and parallelism (the number of threads issuing I/O requests simultaneously).

Traffic Redirection Overhead First, we measured the *redirection overhead* by comparing two cases – LEGACY with MB-FWD. Specifically, in LEGACY, we ran all tests without the StorM platform, and allowed the tenant VM to communicate with the storage target node directly. In MB-FWD, we used StorM platform to direct the storage traffic to the middle-box, but the middle-box did not perform any processing on the storage packets.

We ran all Fio tests on the tenant VM (2 vCPUs and 4 GB memory). A 20 GB volume was created and attached to this VM. We deployed one middle-box for this VM's storage volume, and the configuration of the middle-box VM was the same as the tenant VM – with 2 vCPUs and 4 GB memory. To measure the routing impact in the *worst case*, we placed the middle-box VM and the tenant VM as well as the ingress and egress storage traffic gateways on *different physical nodes*.

We then measured this setup for two cases: For the LEGACY case, a direct path from the tenant VM to the storage target server was used (i.e., the baseline without StorM). In the MB-FWD case, StorM introduced 3 extra hops – the traffic traveled from the tenant VM to the ingress storage traffic gateway, the middle-box VM, the egress storage traffic gateway, and the storage target server. Note that the middle-box did not perform any processing, allowing us to measure only transmission overhead.

We varied the I/O request size of Fio from 4 KB to 256 KB to measure the performance (in IOPS) and latency (in milliseconds). A representative I/O operation pattern was chosen — 50% write and 50% read mixed in a random access manner. One Fio thread was used for each test case – any routing overhead will be directly reflected by the single thread's I/O performance and latency. We ran each test 10 times and took an average to avoid any variability in the network. The variation among the 10 repetitions is less than 5%.

In Figure 4, we observed that the workload's performance under MB-FWD was lower than LEGACY. This is to be expected as the packet routing does cause some additional overhead (a common problem for all middle-box based solutions). Particularly, as the I/O size increased, the performance gap increased — from 7% (4 KB) to 18% (256 KB). This is because an I/O request with a larger size contains more packets, and the latency of this I/O request aggregates the routing delays of all packets. The I/O latency in Figure 7, shows similar results – MB-FWD resulted in slightly increased I/O latency due to the longer forwarding path. Recall that this is the worst case for *traffic redirection* as all forwarding hops of MB-FWD were distributed across different physical hosts. We find that the routing overhead can be reduced by ~20% by placing the ingress traffic gateway close to the tenant VM and the egress gateway close to the storage target server (e.g., in the same physical host). Also, (because the middle-box is not performing any processing) StorM's active-relay approach is not in use, and as we will show next, this effectively avoids most routing overhead.

Another key observation is that the intra-host packet transfer contributes more to the routing overhead than the inter-host packet transfer. The main reason for this behavior is due to the fact that the virtualization driver, for copying network packets (from hypervisor to guest VMs), is less efficient — it uses a single thread per VM's virtual interface and usually causes high CPU utilization. A hardware solution (e.g., SR-IOV) could greatly reduce this overhead.

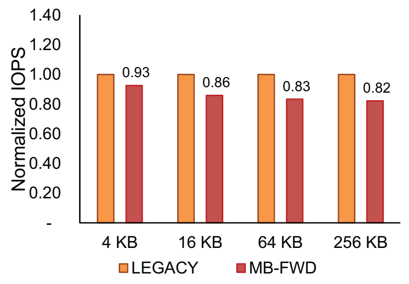
Figure 4: The routing overhead comparison (IOPS) with various I/O sizes (one thread). Higher is better.

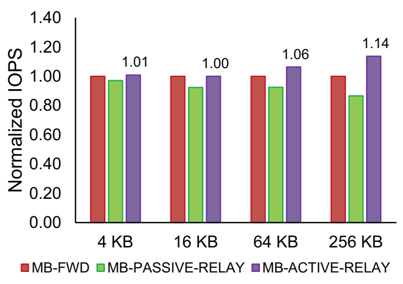
Figure 5: The processing overhead comparison (IOPS) with various I/O sizes (one thread). Higher is better.

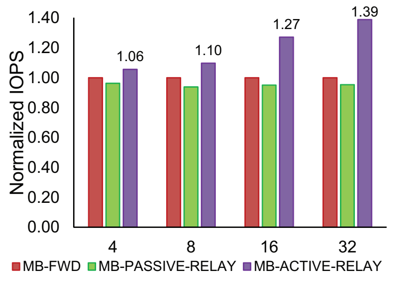
Figure 6: The processing overhead comparison (IOPS) with various I/O threads (16 KB size). Higher is better.

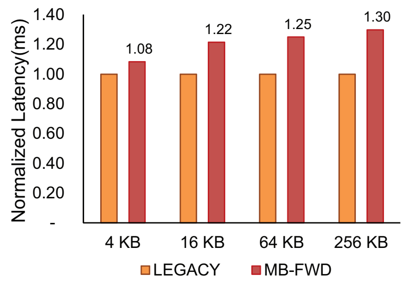
Figure 7: The routing overhead comparison (latency) with various I/O sizes (one thread). Lower is better.

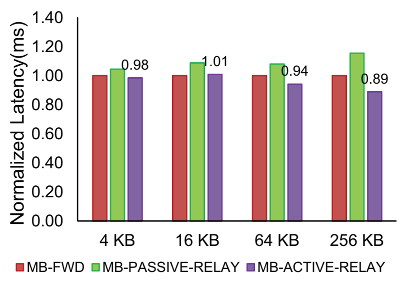
Figure 8: The processing overhead comparison (latency) with various I/O sizes (one thread). Lower is better.

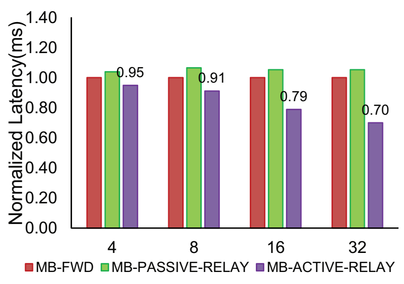
Figure 9: The processing overhead comparison (latency) with various I/O threads (16 KB size). Lower is better.

Middle-box Processing Overhead Next, we measured the overhead resulting from *data processing* inside storage middle-boxes. Data processing overhead can be further broken down into two parts: data extraction overhead (by StorM's API), and data processing overhead (by security/reliability service logic). In this section, we focus on the performance overhead of StorM's API, and discuss the data service processing overhead in Section V-B. To demonstrate a measurable storage service overhead, we ran a "stream cipher" service inside the middle-box, which operates on each bit of the raw data (the details of the service are described in Section V-B). We then used the Fio to generate I/O as described previously. Here, we compared three approaches: MB-PASSIVE-RELAY refers to as the passive-relay approach (as described in Section III-B), MB-ACTIVE-RELAY refers to the (default) active-relay approach, and MB-FWD as described previously (with no processing inside the middle-box).

Confirming our previous intuition, Figure 5 shows that MB-PASSIVE-RELAY added additional overhead on top of the MB-FWD overhead, ranging from 3% to 13% as the I/O size increased from 4 KB to 256 KB. This was caused by the extra computation added to the packets delays, resulting in the low performance. Larger I/O size results in more overhead for of the same reason — the performance degradation aggregates from all packets of the large I/O request. These results further justify the need for StorM's active-relay approach.

In contrast, our proposed MB-ACTIVE-RELAY approach led to the same performance as MB-FWD when the I/O size was small (e.g., 4 KB and 16 KB), and *better performance* when the I/O size became larger. Figure 5 shows a 14% performance improvement when the I/O size was 256 KB. The reason for this improvement in performance is because the active-relay approach shortens the packet acknowledgment path — reduced from four hops to only one hop. Compared with MB-FWD, Figure 8 shows the average I/O latency of MB-ACTIVE-RELAY reduced by 11% with the I/O size of 256 KB.

In Figure 6, we increased the Fio thread number from 4 to 32 to simulate parallelism in the tenant's application. We observed that, compared with MB-FWD, the IOPS number of MB-ACTIVE-RELAY increased by 39% when the workload had 32 threads. Likewise, Figure 9 shows that the average I/O latency of MB-ACTIVE-RELAY reduced by 30%. In this case, even compared with LEGACY, the overhead caused by MB-ACTIVE-RELAY was much less than 10%.

In summary, the packet routing for storage middle-boxes introduces up to 18% performance overhead in the worst case, but this overhead can be mitigated using StorM's active-relay approach, which shortens the packet acknowledgment path. Compared with the LEGACY case, the overall performance overhead under MB-ACTIVE-RELAY is *less than 10%* in all of our measurements.

I/O Access ID	Operation	File	Size
1	read	/mnt/box/.	4096
2∼34	read	META: inode_group_90	4096
35∗	read	/mnt/box/name1/.	4096
...			
71∗∗	read	/mnt/box/name9/.	4096
72∗∗	write	/mnt/box/name9/7.img	16384
...			
287∗	write	/mnt/box/name1/1.img	32768
...			
294∗	write	mnt/box/name1/1.img	4096
295∗	write	/mnt/box/name1/1.img	24576
296	write	inode_group_90	4096

Table I: Reconstructing the high-level file operations from the block-level accesses.

B. Security/Reliability Service Case Studies

In this section, we provide the detailed design and evaluation of three storage services that we have built to offer as middle-boxes. These services show the efficacy of StorM that allows seamless development and transparent deployment of tenant-defined storage security/reliability services.

1) Case 1: Storage Access Monitor: The goal of the storage access monitor is to allow tenants to set an alert on sensitive files and directories, and the middle-box will log all accesses made to these marked resources. Tenants can either periodically request the logs created by the middle-box to see if any unauthorized attempts are made on it or set the policies inside the middle-box to be directly notified on any access. This middle-box provides a crucial service to tenants: even if tenant VMs are compromised and malware attempts to access the sensitive data, those accesses will be logged by the storage monitor inside the middle-box. The access logs can be used to perform post-attack investigation, and access pattern can be used to detect such malware in future.

A storage service executing inside a middle-box can only observe low-level block I/O accesses, hence the storage access monitor must reconstruct the high-level operations from the low-level block accesses in order to enforce tenant policies expressed in files and directories. As described in Section III-C, StorM provides the middle-boxes with semantic reconstruction APIs. Using this service, we built a monitoring engine, a multi-threaded daemon running inside the middle-box, which performs three steps: The first step is *Classification*, where the classification process determines whether an access is to a file's content or metadata using the file system view provided by StorM. The *Update* phase asks StorM to update its file system view from any intercepted metadata. Lastly, the *Analysis* phase logs (or raises an alarm) accesses to monitored files or directories.

Synthetic Attack Scenario To demonstrate the accuracy and usefulness of the storage access monitor, we first present a synthetic use case. An iSCSI volume was attached to a tenant VM, and mounted under "/mnt/box". The volume

ID	Operation	File	Size
1∗	write	/mnt/box/name1/1.img	4096
2∗∗	read	/mnt/box/name9/7.img	4096

Table II: File operations in the tenant VM

was formatted to Linux Ext4, where 10 folders, "name0" to "name9", were created. Each folder contained ten files from "1.img" to "10.img". We then attached the monitoring middle-box to the tenant VM, and file operations were issued in the tenant VM; two of them are shown in Table II. With the help of the monitoring engine, these file operations are successfully reconstructed and logged in the middle-box as shown in Table I.

We observed that a high-level file operation in the tenant VM may generate several block-level accesses captured by the monitoring middle-box. For example: when the tenant VM reads "7.img" in the directory "name9", the file system first reads the inode metadata of directory "name9" and the data blocks under this directory. Then, the content of "7.img" is read from the data blocks. We also observed that the written messages could be cached in the VM's local buffer for some time. As a result, the block-level I/O access sequence is different from the file I/Os — the write operations may delay all the read operations. Notably, in addition to logging sensitive file accesses, this monitor provides detailed file system level activities. It can further be utilized to debug abnormal system behaviors (e.g., the write blowup issue in certain file systems [12]) and optimize file system performance.

Real-world Malware Scenario We also applied the storage access monitor to perform a real-world malware behavior study. We chose *HEUR:Backdoor.Linux.Ganiw.a*, a Linux backdoor Trojan detected by Kaspersky [13] in 2015. We mainly studied the installation process of this malware with the help of the storage access monitor.

When the malware was executed with root privileges, the storage monitor observed the creation of several files and directories. The main steps are listed in Table III. Not surprisingly, this malware installed its startup scripts in "/etc/init.d" to ensure the malware would be launched automatically at the system startup. The malware also linked the start scripts to different system run levels (1-5). Moreover, the malware replaced "selinux" with its own copy and tried to launch the fake one at the system startup. To hide from checks, the malware replaced several system tools such as "netstat", "ps", "lsof" and "ss" with their trojan version.

In addition to detecting created files during the malware's installation process, the storage monitor also observed several important files read by the malware. For example, the malware accessed the GeoIP database by reading the file "/usr/share/GeoIP/GeoIPv6.dat". Later, it called the SAX (Simple API for XML) driver by reading the Python file "/usr/lib/python3.4/xml/sax/expatreader.py" (a Python mod-

Step 1	`cp "#!/bin/bash\n<path_to_malware>" /etc/init.d/DbSecuritySpt`
Step 2	`ln -s /etc/init.d/DbSecuritySpt /etc/rc[1-5].d/S97DbSecuritySpt`
Step 3	`cp <path_to_malware> /usr/bin/bsd-port/getty`
Step 4	`cp "#!/bin/bash\n/usr/bin/bsd-port/getty" /etc/init.d/selinux`
Step 5	`ln -s /etc/init.d/selinux /etc/rc[1-5].d/S99selinux`
Step 6	`cp <path_to_malware> /bin/netstat` `cp <path_to_malware> /usr/bin/lsof` `cp <path_to_malware> /bin/ps` `cp <path_to_malware> /bin/ss`

Table III: File system accesses by the backdoor malware.

ule in C). We concluded that this malware called the Python functions from its C (or C++) code for parsing XML documents, and it tried to get the machine's location information using its IP or hostname. We note that the revealed file access patterns of malware can then be used by the middle-box for future detection of the same malware.

2) Case 2: Data Encryption: To allow tenants to keep their data confidential, we have implemented a storage encryption middle-box. The goal of this middle-box is to encrypt the tenant data before it is written to the disk and decrypt it when the data is requested. Implementing this functionality inside a middle-box offers additional flexibility to tenants to decide when encryption should be performed and what algorithm should be used (as opposed to depending on the cloud provider for this service). Further, instead of deploying encryption services for each VM, multiple VMs belonging to the same tenant can share the same encryption middle-box (improving performance with less management overhead).

We have implemented a widely used block cipher in the encryption middle-box by leveraging *dm-crypt* — a well-known disk encryption subsystem in the Linux kernel. We deployed it in the kernel space of the encryption middle-box. By passing tenants' storage flows to the encryption middle-box, data encryption and decryption for the corresponding tenant VMs' volumes was easily achieved.

We compared the middle-box encryption solution with a traditional tenant-side encryption solution (by installing the encryption system in the tenant VM). The same AES cipher with 256 bits keys were used for both solutions. A 20 GB volume was created and attached to the tenant VM for both scenarios. Note that the client-side encryption requires configuring the volume's format to enable the encryption approach. However, the middle-box solution does not have this requirement; it is *transparent* to the tenant VM.

To test the decryption and encryption, we ran a simple FTP server in the tenant VM to download/upload a large file from/to the attached volume, respectively. We observed that both tenant-side and middle-box encryption solutions nearly reach the maximum storage bandwidth. The average bandwidth (both read and write) was around ∼88 MB

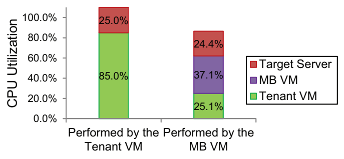

Figure 10: CPU utilization breakdown (with FTP).

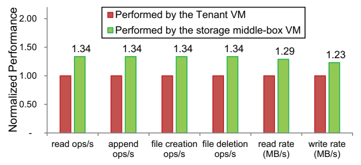

Figure 11: The application-level performance comparison (with PostMark).

for the tenant-side solution and ∼84 MB for the middle-box solution. Interestingly, the middle-box solution led to much lower overall CPU resource utilization as shown in Figure 10. Note that the overall CPU utilization of the middle-box solution was the sum of the CPU utilization from the tenant VM, the middle-box VM, and the storage target. On the contrary, the overall CPU utilization of the tenant-side solution only involved the tenant VM and the storage target.

The middle-box solution *reduced* the overall CPU utilization by 20% due to the CPU savings in the tenant VM. To confirm this, we used a more realistic application workload, PostMark, which generates many small file operations to simulate an email server. With the same configuration as above, Figure 11 showed the performance (operations per second) of the decomposed components of PostMark. Compared with the client-side solution, the middle-box solution improved the performance of each component, ranging from 23% to 34%. Upon deeper investigation, we found that this was because outsourcing encryption to a middle-box shortens blocking time for application threads. Dm-crypt may hold application threads on spinlocks (wasting CPU cycles) while encrypting/flushing writes blocks to disk. However, the middle-box makes this application-side process much faster: once data is acknowledged by the active-relay, the application threads continue.

3) Case 3: Data Reliability: We have implemented a tenant-defined replica dispatch service in a storage middle-box. Data storage replication provides data redundancy that can be used for improved performance and fault tolerance (if the main storage backup system fails). Our data replica dispatch middle-box service can also be highly customized. For example, tenants can selectively replicate important files rather than the whole array with customizable replication levels (e.g., two or three replicas). In addition, tenants can leverage multiple replicas to achieve enhanced read throughput.

For write I/O operations, in addition to forwarding the data to the original volume, our replication service copies exactly the same I/O data in advance to other backup volumes

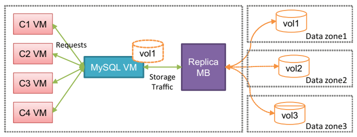

Figure 12: The replication test configuration.

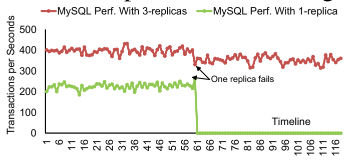

Figure 13: The running status of MySQL before and after a replica fails.

that are attached to the middle-box. Importantly, to keep the state consistent across all replicas, we strictly ensure that a same sequence ordering of write I/O operations on all volumes. On the other hand, for read I/O operations, the replication service alternatively chooses one of the available replicas for reading data. As a result, the read throughput aggregates from all available replicas. Once a replica is not responsive for some reason, it will be eliminated from future operations. The unfinished reads of that failed replica are served from one of the other active replicas.

To evaluate the tenant-defined replication service, we set up a realistic environment, shown in Figure 12, with one VM hosting a database server and four tenant VMs sharing the server (all belonging to the same tenant). We ran MySQL on the server VM associated with a 20 GB volume attached as the database disk. Each tenant VM ran Sysbench — an OLTP benchmark accessing the MySQL database with six requesting threads in complex mode (both *read* and *write*). We set the replication factor to three, where the replication middle-box was attached to two replicas (20 GB each). At runtime, we injected an error (at the 60th second) to one of the replicas to make it unavailable (by closing the iSCSI connection). Figure 13 shows the running status of MySQL before and after that replica fails. We observed that once the replica failed the replication service removed the failed replica and ensured the database server continuously worked properly using the two other available replicas. The average performance of MySQL after the failure dropped a little due to lower read parallelism with less replicas, but compared to the non-replication case using only one store, we observed 80% performance improvement using three replicas because of aggregated read throughput.

Our case studies demonstrated that many storage security and reliability services can be offered via middle-boxes to cloud tenants. Moreover, StorM is capable of providing the development and deployment platform for these middle-boxes.

VI. Related Work

Offering security and system services via middle-boxes has been around for almost a decade. However, the arrival of software-defined networking (SDN) and the possibility of deploying middle-boxes dynamically, instead of a static chain, has renewed researchers' interest in middle-boxes. In recent works [14], [15], researchers have identified challenges in deploying middle-boxes in a SDN-enabled network and shown why straightforward deployment of middle-boxes in SDN networks would be problematic. Our work on designing and implementing StorM is along the same direction: we investigated the possibility of offering tenant-defined services in the cloud storage network via middle-boxes and discovered that existing cloud systems lack mechanisms to support them.

Previous research has proposed various security mechanisms as cloud services to protect VMs, applications, and security groups. SSC [16] did allow the deployment of tenant-specific storage security services, but their approach required these protections to be installed inside of tenant VMs running on a modified cloud platform. In contrast, we proposed the first storage security platform that allows the deployment of tenant-specific storage services via virtualized middle-boxes in the cloud.

Cryptographic solutions [17] have been proposed to protect the data in clouds. One common cloud data encryption solution involves service providers encrypting customers' data [17], [18] — the approach that major cloud service providers, like Microsoft, Google, and Yahoo have adopted. EMC provides its Encryption-as-a-Service (EaaS) cloud service [19], which enables client-side encryption. To complement these existing solutions, our StorM provides a flexible platform, where various encryption techniques (among other storage services) can be built upon. This allows tenants to flexibly choose the cryptographic algorithms that they want to implement inside a middle-box, depending on their security/storage needs.

In addition to cryptographic solutions, previous research has also looked into disk monitoring and logging solutions, such as the host-based IDS solutions Tripwire [20] and FWRAP [21]. Host-based disk IDS solutions required a trusted OS, but advanced kernel rootkits can break that assumption. To overcome this problem, Virtual Machine Introspection (VMI) based techniques were proposed such as XenAccess [22] and other systems [23], [24]. These introduced a set of monitoring libraries running in the privileged domain (dom0 or the VMM itself) and tracked guest-level activities such as virtual disk accesses. In contrast to these services that required access to the privileged domain or changes inside the tenant VM, StorM's monitoring service requires no support from the tenant VM and instead executes monitoring code in a separate, isolated VM (the middle-box).

To ensure storage reliability at the block level, existing vendor-specific solutions depend on hardware adapters (e.g., EMC's SRDF, and NetApp's SnapMirror) to replicate entire storage arrays. Further, network-based replication (e.g., EMC's RecoverPoint), using an appliance that sits at the edge of the network, takes multiple arrays and servers into account. CYRUS [25] and CDStore [26] provide user-controlled file-level data reliability by dispersing users' backup data across multiple clouds. However, each of these solutions requires client-side software, and in contrast, StorM requires no software inside the tenant VM, and allow tenants to flexibly choose the data replication services on demand in a transparent and seamless manner.

VII. Conclusions

In this paper, we have presented StorM, a storage security/reliability service platform for the multi-tenant cloud systems. StorM allows each tenant to deploy tenant-defined storage security/reliability services in the form of virtualized middle-boxes that transparently reside between the tenant VMs and storage servers. To enable this storage service platform, we addressed three main challenges — network splicing, platform efficiency, and semantic gap. We implemented a prototype of StorM on top of the popular OpenStack cloud system. We also built three middle-box services to demonstrate the efficacy of StorM— storage access monitor, data encryption, and data replication. Our evaluation results demonstrate that StorM provides tenants with customized, value-added storage services with low performance overhead.

VIII. Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by NSF under Award 1219004.

References

[1] R. Sharma and R. K. Trivedi, "A case of multilevel security application for ensuring data integrity (prevention and detection) in cloud environment," *International Journal of Computer Applications*, 2014.

[2] K. Ren, C. Wang, and Q. Wang, "Security challenges for the public cloud," *IEEE Internet Computing*, 2012.

[3] "Virtualization Data Protection Report," http://www.dabcc.com/documentlibrary/file/virtualization-data-protection-report-smb-2013.pdf.

[4] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *ACM SIGCOMM Computer Communication Review*, 2011.

[5] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, 2008.

[6] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: enabling innovation in middlebox deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.

[7] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," in *ACM SIGCOMM Computer Communication Review*, 2011.

[8] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[9] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, 2009.

[10] "Open-iscsi," http://www.open-iscsi.org/.

[11] "Fio," http://linux.die.net/man/1/fio.

[12] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: a facebook messages case study," in *Proceedings of the 12th USENIX conference on File and Storage Technologies*, 2014.

[13] "Kaspersky: HEUR:Backdoor.Linux.Ganiw.a," MD5:ef5d928cab15a54d33209510818f5c72http://malwaredb.malekal.com/.

[14] Z. Qazi, C. C. tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *SIGCOMM*, 2013.

[15] S. Fayazbakhsh, V. Sekar, M. Yu, and J. Mogul, "Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *HotSDN*, 2013.

[16] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-service cloud computing," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[17] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Financial Cryptography and Data Security*, 2010.

[18] S. Kamara, P. Mohassel, and B. Riva, "Salus: a system for server-aided secure function evaluation," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[19] "Encryption as a service," http://www.cloudlinktech.com/wp-content/plugins/download-monitor/download.php?id=133.

[20] G. H. Kim and E. H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994.

[21] B. Y. M. Cheng, J. G. Carbonell, and J. Klein-Seetharaman, "A machine text-inspired machine learning approach for identification of transmembrane helix boundaries," in *Foundations of Intelligent Systems*, 2005.

[22] B. D. Payne, M. De Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, 2007.

[23] Y. Zhang, Y. Gu, H. Wang, and D. Wang, "Virtual-machine-based intrusion detection on file-aware block level storage," in *Computer Architecture and High Performance Computing, 2006. SBAC-PAD'06. 18TH International Symposium on*, 2006.

[24] F. Tsifountidis, "Virtualization security: Virtual machine monitoring and introspection," *Signature*, 2010.

[25] J. Y. Chung, C. Joe-Wong, S. Ha, J. W.-K. Hong, and M. Chiang, "Cyrus: Towards client-defined cloud storage," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[26] M. Li, C. Qin, and P. P. Lee, "Cdstore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal," *arXiv preprint arXiv:1502.05110*, 2015.