Multitenant Access Control for Cloud-Aware Distributed Filesystems

Giorgos Kappes, Andromachi Hatzieleftheriou and Stergios V. Anastasiadis

Abstract—In a virtualization environment that serves multiple tenants (independent organizations), storage consolidation at the filesystem level is desirable because it enables data sharing, administration efficiency, and performance optimizations. The scalable deployment of filesystems in such environments is challenging due to intermediate translation layers required for networked file access or identity management. First we define the entities involved in a multitenant filesystem and present relevant security requirements. Then we introduce the design of the Dike authorization architecture. It combines native access control with tenant namespace isolation and compatibility to object-based filesystems. We introduce secure protocols to authenticate the participating entities and authorize the data access over the network. We alternatively use a local cluster and a public cloud to experimentally evaluate a Dike prototype implementation that we developed. At several thousand tenants, our prototype incurs limited performance overhead below 21%, unlike a solution from industry whose multitenancy overhead approaches 84% in some cases.

Index Terms—security services, cloud storage, distributed filesystems, datacenter infrastructure, identity management, authorization

1 Introduction

Cloud infrastructures are increasingly used for a broad range of computational needs in private and public organizations, or personal environments. In the datacenter, aggressive systems consolidation is opening up new opportunities for flexible data sharing among the users of one or multiple tenants (e.g., constellations of co-resident applications [1]). For commercial, analytics and social applications, data exchanges at low cost are a key benefit that valuably complements the reduced operational expenses already offered by the cloud. As the cloud landscape is dominated by concerns about the security and backward compatibility of systems software, it remains open problem how to achieve scalable file sharing across different virtualized machines or personal devices serving the same or different tenants.

Existing solutions of cloud storage primarily provide centralized management of entire virtual disks over a common backend (e.g., Ceph RDB, OpenStack Cinder). Although they conveniently provision the capacity elasticity of disk images, they do not facilitate native support of scalable data sharing at file granularity. Similarly, identity management in the cloud selectively grants coarse-grain root authorization to administrative domains, but lacks the fine granularity required to specify the permissions of individual end users. Furthermore, the known file-based solutions face scalability limitations because they either lack support for multiple tenants, rely on global-to-local identity mapping to support multitenancy, or have the guests and a centralized filesystem (or proxy thereof) sharing the same physical host [2], [3], [4].

Multitenant access control of shared files should securely isolate the storage access paths of different users in a configurable manner. Authentication and authorization have already been extensively studied in the context of distributed systems [5], [6], [7]. However, a cloud environment introduces unique characteristics that warrant reconsideration of the assumptions and solution properties according to the following requirements:

1) Each tenant should be able to specify the user identity namespace unrestricted from other tenants. The simple application of identity mapping at large scale adds excessive overhead.
2) The enforcement of authentication and authorization functions should be scalably split across the provider, the tenant and the client. Directory services designed for private environments do not easily support tenants or handle enormous user populations.
3) Unless it is natively supported for flexible and efficient storage access, file sharing requires complex management schemes that are restrictive, error prone and costly.
4) Access control should take advantage of secure hardware (as root of trust) in machines administered by the provider and the tenant, or in personal user devices. Unless the virtualization execution is adequately trustworthy, the tenants will simply refrain from data sharing.

We introduce the Dike multitenant access control for outsourcing the complex functionality of storage authorization and data sharing to cloud service providers. The proposed design reduces the amount of systems infrastructure maintained by the tenant, and provides an effective interface for storage interoperability among co-located administrative domains. Thus, we enable shared data accesses among numerous users, and facilitate distributed data processing over large cluster installations.

- *G. Kappes and S. V. Anastasiadis are with the Department of Computer Science and Engineering, University of Ioannina, Ioannina, Greece.*
- *A. Hatzieleftheriou is with Microsoft Research, Cambridge, UK. Work contributed while the co-author was graduate student at the University of Ioannina.*
- *Corresponding author: S. V. Anastasiadis (email: stergios@cse.uoi.gr)*

We rely on an object-based, distributed filesystem to scalably store the data and metadata of individual files. A guest directly mounts the shared filesystem without the involvement of a local or network proxy server. Each tenant manages the identities and permissions of its own users independently. By maintaining *separately* the access permissions of each tenant, the filesystem securely isolates the identity namespaces of the tenants but also enables configurable file sharing across different tenants and hosts. We provide the Dike prototype implementation of the above approach over the Ceph distributed filesystem [8]. With microbenchmarks and application-level experiments, we quantitatively demonstrate that our design incurs only limited performance overhead.

We can summarize our contributions as follows:

- Description of security requirements in consolidated cloud storage.
- Design of multitenant access control for an object-based storage backend.
- Specification of secure protocols for the authentication of the participating entities and the authorization of the data access over the network.
- Prototype implementation over a production-grade distributed filesystem.
- Experimental performance evaluation of multitenancy overheads across different systems.

In a preliminary publication [9], we examined the idea of native multitenancy support for the access control of distributed filesystems. The present manuscript extends significantly our prior published work by experimentally motivating it and considering the context of current virtualization environments; it covers comprehensively the entity definition, threat model and security analysis of the proposed solution; it specifies secure protocols for the authentication and authorization operations; and provides insightful details about the comparative performance and resource utilization of Dike over both a local cluster and the Amazon public cloud across a microbenchmark and two application-level workloads.

In the remaining document we first motivate our work (§2) and define the system entities and assumptions (§3). We specify the system design and secure protocols (§4), and describe the prototype implementation as extension of a distributed filesystem (§5). We present our experimentation environment and explain several notable results (§6). We point out previous related research in comparison to our work (§7), and further discuss our results in the context of latest advances in virtualization technology (§8). Finally we summarize our conclusions and plans for future research (§9).

2 Motivation

Consolidation at the filesystem level is increasingly advocated as a desirable multitenancy paradigm because it enables improved storage efficiency across the different users and machines co-located in the datacenter [2], [3], [4], [10]. The block-based interface isolates a virtual disk into a protection domain fully controlled by the guest; this is an attractive approach due to the versioning and migration properties of virtual disks [2]. Instead, a file-based interface supports easy resource administration and optimized performance, but also provides the "killer" advantage of configurable guest isolation and sharing at fine granularity [2], [3], [4], [10], [11].

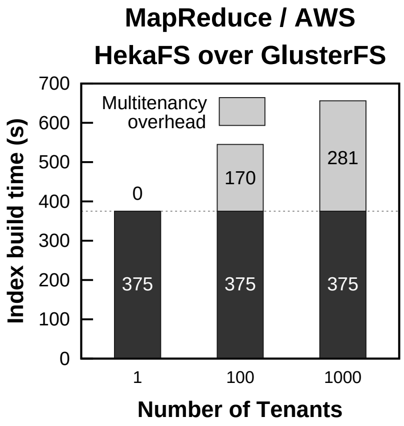

Fig. 1. The overhead of identity mapping added by HekaFS over GlusterFS for multitenancy support. For 1 tenant we use vanilla GlusterFS.

Given the increasing popularity of filesystem storage, the OpenStack Manila is an architecture introduced to integrate filesystem shares (i.e., shared file trees) with guest machines [12]. However, the specification of the Manila protocol that connects the filesystem to the guest is work in progress. Today, cloud storage resources are typically isolated across individual tenants at the network level instead of actually being shared at the filesystem level. The OpenStack Keystone service [13] (or the AWS AIM [14]) federates the identities of different tenants (accounts), but only manages the administrative entities rather than the end users of the OpenStack (AWS) services.

Cloud storage security could be preferably enforced at the infrastructure rather than the application level for enhanced functionality and improved resilience to programming bugs in the application or the guest system [15]. In analytics workloads, a shared distributed filesystem often maintains the accumulated data and allows temporary local caching at compute nodes for processing. Thus, a multitenant filesystem could facilitate the storage consolidation for collaborative analytics jobs of the same or different customers [16]. The trend toward lightweight virtualization through containers or library operating systems can take advantage of secure storage offloading to the provider and relieve the guest machines from unnecessary complexity and overhead [17], [18]. Similarly, a thin filesystem client can efficiently integrate personal devices into cloud shared folders, especially as security features are included in the processor chipset hardware [19].

We experimentally motivate our work by examining the performance overhead of identity mapping currently used by HekaFS [20] to support multitenancy over the GlusterFS distributed filesystem (§ 6.2.2). We consider the Phoenix v2.0 [21] shared-memory implementation of MapReduce, and use the reverse index application to generate the full-text index of a 1.01GB file collection. In Fig. 1, we measure the duration of index build over GlusterFS for 1 tenant, and over HekaFS for 100 or 1,000 tenants. The reverse index application only takes 375s in GlusterFS, but requires 545s (45% more) or 656s (75% more) in HekaFS configured with 100 or 1,000 tenants, respectively. From measurements of the I/O system calls, we found the increased number of tenants to be accompanied by higher latency in specific metadata operations (e.g., opendir, close) of the filesystem.

Below we examine examples of virtualization in which file-based storage consolidation makes sense for (i) fine-granularity access control, (ii) storage efficiency, (iii) data sharing, and (iv) administration flexibility.

Virtual Desktops An enterprise private cloud stores the desktop filesystems of personal thin clients. Each desktop root filesystem is stored as a separate directory with access limited to a single client. As an optimization, the root directory of each client can be branched out from a shared, read-only directory.

Shared Workspace A shared filesystem maintains the home directories of collaborating users. Typical file exchanges of unstructured data (e.g., documents, images) are enabled through shared folders in a Dropbox-like manner.

Software-as-a-service A software-as-a-service provider supports business customers with disjoint end users [22]. Each business customer is treated by the filesystem as a tenant with separate application files in writable mode (e.g., databases) and shared system files in read-only mode (e.g., libraries of executable code).

Software Repository A public cloud provides a shared software repository that can be forked into separate branches by different groups of developers. A group obtains writable access to its own branch, and read-only access to branches of other groups. A similar scheme can be used to share scientific datasets.

3 System Requirements

We outline the general requirements of our system through the goals, assumptions, system trust and threat model.

3.1 Goals

In the proposed access control, we set the following goals:

1) **Isolation** Each tenant is free to choose identities for its users. Thus we isolate the identity space and access control of different tenants to prevent namespace collisions.
2) **Sharing** Provide flexible access control to enable secure file sharing within a tenant or among different tenants.
3) **Efficiency** Natively support multitenant access control to achieve the required performance and scalability for enormous numbers of users or files.
4) **Interface** Leverage the architectural characteristics of widely-adopted filesystems for backward compatibility with existing applications.
5) **Manageability** Maintenance support at the file level allows the cloud provider to uniformly and flexibly manage the storage resources of different tenants.

Domains

D: set of domains in the system

P: set of providers in the system ($=\{p\}$ by default)

T: set of tenants in the system

$D = P \bigcup T$

$s(d)$: set of all domains sharing any file with domain $d \in D$

Users

U: set of users in the system

U_d: set of all users belonging to domain $d \in D$

$U = \bigcup_{d \in D} U_d$, with $U_i \bigcap U_j = \emptyset,\ \forall i, j \in D$

Files

F: set of files in the system

F_d^i: all files in domain $d \in D$ accessible by domain $i \in D$

F_d: set of all files owned by domain $d \in D$ (i.e., $F_d = F_d^d$)

$F = \bigcup_{d \in D} F_d$, $F_i \bigcap F_j = \emptyset,\ \forall i, j \in D$

F^i: set of all files accessible by domain $i \in D$

$F^i = \bigcup_{d \in s(i)} F_d^i$, with $s(i)$ as defined above

Permissions

$P_d^i(u, f)$: permissions of user $u \in U_i$ to file $f \in F_d$, $i, d \in D$

$P_d^i(f)$: per-user permissions of file $f \in F_d$ in domain $i \in D$

$P_d^i(f) = \bigcup_{u \in U_i} P_d^i(u, f)$

$P_d^{\bar{d}}(f)$: permissions of file $f \in F_d$ in domains $i \in D,\ \forall i \neq d$

$P_d^{\bar{d}}(f) = \bigcup_{u \in U_i,\, i \neq d} P_d^i(u, f)$

$P_d(f)$: per-user permissions of file $f \in F_d$ in all domains

$P_d(f) = P_d^d(f) \bigcup P_d^{\bar{d}}(f)$

Fig. 2. Summary of basic entities and their properties in the Dike system.

3.2 Definitions and Assumptions

The *user* is an entity (e.g., individual or application) that receives authorizations and serves as unit of accountability in the system. We call *tenant* an independent organization whose users consume networked services from a cloud *provider* [22]. The *domain* generally refers to any organization, including the provider itself, that accesses the cloud resources for consumption or administration purposes. The *directory* refers to a registry of users and their attributes, while *folder* is a catalog of files in the filesystem.

A *server* implements service actions, and a *client* provides local access to a service over the network. We collectively refer to the clients or servers of the system as *nodes*. Through their local client, the users of a tenant access storage services at file granularity. The user who creates a file in a tenant is the *owner user* of the file and assigns access permissions with a discretionary model. A file is owned by the tenant to which the owner user belongs. In case of a file shared across multiple tenants, the assignment of access rights in an non-owning tenant is undertaken by the administrator user of that tenant.

In Fig. 2 we summarize some basic entities with their properties in the Dike system. The set of domains D is derived from the union of the tenant set T with the provider singleton P (extension for multiple providers left for future work). The sets of users belonging to different domains are disjoint and disjoint are also the sets of files owned by different domains ($\forall i, j \in D,\ U_i \bigcap U_j = F_i \bigcap F_j = \emptyset$). The permissions of file f owned by domain d is derived from the union of the per-user file permissions across all the domains in the system ($P_d(f) = P_d^d(f) \bigcup P_d^{\bar{d}}(f)$).

We adopt the architecture of a distributed object-based filesystem because it is scalable and typically used in cloud environments. The system consists of multiple metadata and object servers: a *metadata server* (MDS) manages information about the file namespace and access permissions, and an *object server* (or object storage device, OSD) stores file data and metadata in the form of objects. A client can only access an OSD after an MDS communicates to the OSD the necessary authorization capability (signed token of authority) for the requested access.

3.3 System Trust

An independent provider operates the datacenter nodes at which the filesystem clients and servers run. Multiple virtual nodes generally share a physical host. A client runs on a virtual node inside the datacenter or on an external mobile device with sufficient hardware security or virtualization support (e.g., ARM TrustZone [23], cTPM [19]). A secure protocol for clock synchronization keeps the time synchronized across the nodes of the system.

Standard cryptographic primitives ensure the confidentiality, integrity and freshness of the network communication. Secure hardware (e.g., Trusted Platform Module) at each physical machine applies static measurement to certify the integrity of the system software stack [24], [25]. A cryptographically signed statement of authenticity serves as certificate of node integrity [5]. A central monitor (*attestation server*) inside (or outside [26]) the datacenter applies static remote attestation to build up the infrastructure trust.

The distributed filesystem protects the confidentiality and integrity of stored data and metadata by permitting networked accesses to authorized users and handling the revocation or delegation of permissions. The filesystem infrastructure enforces the access policy at the granularity of individual files, although possible extension to file collections or byte ranges within a file is compatible with our design [27], [28].

The co-located tenants may not trust each other. Each tenant is responsible to specify the file access permissions of its individual users. A tenant can share data with other tenants under the access permissions specified to the provider. In the terminology of the attribute-based access control model with tenant trust (MT-ABAC [29]), we follow the type-γ trust that lets the trustor tenant control the existence of the trust relation, and the trustee tenant control the assignment of cross-tenant attributes to its own users.

3.4 Threat Model

The adversary can be an external attacker or authenticated user remotely attempting unauthorized filesystem access by trying to steal or generate certificates, keys and tickets, or using network attacks to eavesdrop, modify, forge, or replay the transmitted packets of the filesystem protocols. The unauthorized requests may attempt to read or tamper with the stored data and metadata of the filesystem, including the access policy that grants file operation permissions within and across tenants or the provider. The adversaries may also exchange certificates through an out-of-band channel in an attempt to bypass the sharing policy enforced by the filesystem.

In general, an employee of the provider (*honest-but-curious*) may attempt to read the filesystem data of the tenants, but does not have incentive to corrupt the filesystem state or collude with adversaries. A tenant may choose to activate data sharing with the provider for enabling security services supported by the system (e.g., anti-virus scanning). For transparency reasons, the file owner can always use the tenant view to inspect the applicable access policy including possible file sharing with other tenants and the provider.

The privileged software (e.g., hypervisors) is measured before being trusted to run system services and isolate the processes of different tenants. We do not consider runtime integrity monitoring (e.g., to prevent zero-day attacks), given the lack of established methods for dynamic remote attestation in virtualization environments [30]. We also leave out of scope the malicious manipulation through physical access or side-channel attacks (e.g., power analysis, cache timing, bus tapping). We target filesystem access control without any explicit attempt to provide general solutions to denial of service, key distribution, traffic analysis, and general multitenant sharing of resources other than storage [31].

3.5 Encryption and keys

Public keys are used as identities that uniquely identify the entities of the system, such as users and services. A tenant is identified through the key of its tenant authentication service. For privacy protection, a privacy certification authority ensures the unlinkability of multiple keys referring to a single user (this technique is not currently supported in our prototype implementation) [23]. The private keys of an entity can only appear in plaintext form at the volatile memory of an attested node. They are stored persistently in encrypted form, or partitioned across multiple nodes for improved protection [32]. The interacting entities securely communicate over symmetric keys, which are agreed upon with public-key cryptography dynamically. We leave outside the scope of the present work the consideration of data encryption by the filesystem servers on the storage media.

4 System Design

Next we introduce the Dike architecture of multitenant access control for networked storage shared at the file level.

4.1 Identity Management

Identity management refers to the representation and recognition of entities as digital identities in a specific domain [33]. A multitenant environment complicates the secure operation of a shared filesystem due to potentially conflicting needs from multiple independent organizations. Below we consider possible schemes of identity management for multitenant filesystems.

Centralized A common directory administered by the provider maintains the identities of users for all the tenants [34]. Such an approach lacks the required scalability

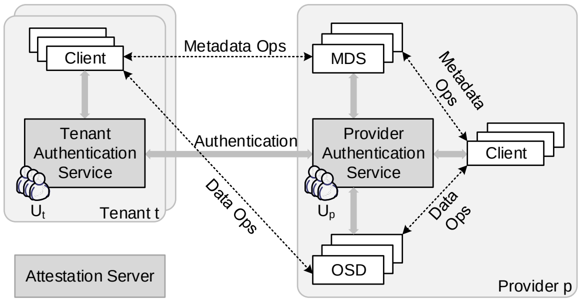

Fig. 3. The hierarchical architecture of Dike. The interaction of the attestation server with all the nodes is omitted for readability. We use dark gray background for the new entities added in Dike with respect to an existing object-based filesystem.

and isolation because it centrally manages all the identities, and restricts the tenants from making free identity choices.

Peer-to-peer Assuming globally unique identities, different tenants communicate directly with each other to publicize the identities of their users and groups [35]. Cross-tenant file sharing is possible through direct inclusion of remote users into the access policy of a file. A drawback is the need to periodically keep up to date the contributed users and groups across the collaborating tenants.

Mapped A shared filesystem maps the identities of users to globally unique identities [4]. For instance, the global identity space is partitioned into disjoint ranges, and each range is assigned to a different tenant. Identity mappings incur extra runtime overhead because they are applied dynamically whenever a server receives a request. Moreover, granting the permissions of local-user classes to remote users opens up the way for violating the principle of least privilege.

Hierarchical In order to isolate its identity space, each tenant maintains a private authentication service to manage locally the identities of its users. The authentication services of legitimate tenants are registered with the underlying common filesystem of the provider. Requests incoming from the clients of an approved tenant are processed by the filesystem according to the stored access permissions of each file.

4.2 Authentication

In Dike we adopt the hierarchical identity management because it offers the isolation and scalability properties required for multitenancy. An attestation server is used to bootstrap the system trust (§3.3). We partition the task of filesystem entity authentication among the provider and the tenants (Fig. 3). Each tenant uses a separate *tenant authentication service (TAS)* to authenticate the local clients and users. Additionally, the provider operates a *provider authentication service (PAS)* to authenticate the metadata servers, object servers and tenant authentication servers of the system. The clients (users) are distinguished into the *tenant clients* (*users*) that can only access the resources of the filesystem belonging to a particular tenant, and the *provider clients* (*users*) that are trusted to access the entire filesystem.

The user identities and the file permissions are explicitly visible to the filesystem infrastructure. This is necessary in order to run the enforcement of the access-control policy as a service of the provider rather than the tenant. For accesses of provider users the authentication process remains similar with the only difference that both the clients and users are directly authenticated by the PAS rather than a TAS.

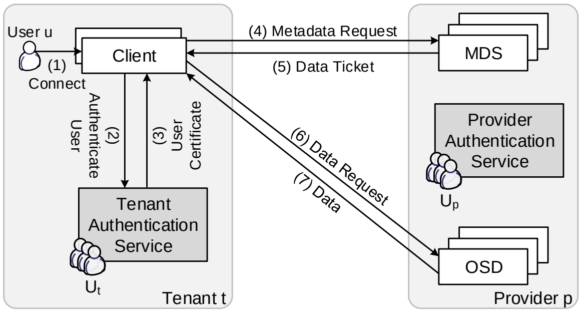

Fig. 4. Over dashed lines in the parentheses we enumerate the steps of file access by a user in the Dike multitenant access-control system. Important steps required for the attestation and authentication of the system nodes are omitted for readability. The new entities of Dike are denoted with dark gray background.

4.3 Operation Example

In Fig. 4, we show the interaction between a tenant client and the filesystem nodes in order to allow a user data access from the filesystem. We follow the involved steps at a relatively high level before introducing next the secure protocols (§4.4). A client is initially authenticated by the TAS (not shown). The tenant user connects to the client (step 1). On behalf of the user, the client contacts the TAS (2) and receives back a user authentication certificate (3) to request filesystem access (4) from the metadata server (MDS). The MDS validates the received user authentication using a PAS-issued certificate of TAS. Then, the MDS issues to the client a *data ticket* (5) to access an object server (steps 6-7). The data ticket securely specifies the user and permissions of the authorized operation over a file.

4.4 Secure Protocols

Next we use secure protocols to describe the authentication and authorization steps in more detail. In the protocol definitions, the participating entities include the privacy certification authority v, the attestation server a, the user u, the client c, the TAS t, the PAS p, the metadata server m, and the object server o. The notation $x \rightarrow y : z$ denotes the transmission of message z from entity x to entity y.

The private and public keys of entity x are denoted as K_x^{-1} and K_x respectively. We use the symbol K_{xy} for the secret key established between entities x and y. The notation $\{z\}K$ denotes the message z encrypted with the secret or public key K, or signed with the private key K. The operation $H(z)$ refers to the hashing of message z. We use the letter N for referring to a nonce, and the symbols T_s and T_e for referring to the starting and ending time of a ticket validity.

The symbols C, P, T, M and O stand for the client, PAS, TAS, MDS and OSD sets respectively. In our design a TAS serves as proxy for a tenant and the PAS as proxy for the provider. The attestation server knows the public key K_v of the privacy certification authority, and all the nodes receive

$\mathcal{P}_0$: Attestation by server a of entity $x \in C, T, P, M$, or O

$x \to a : \{\{K_x, H(w), N\}\, K^{-1}_{AIK(x)}\}K_{xa},$

$\{K_{xa}\}K_a,\ \{K_{AIK(x)}\}\, K_v^{-1}$

$a \to x : \{\mathcal{C}^a_x, N+1\}K_{xa}$

$\mathcal{C}^a_x = \{K_x, T_s, T_e\}\, K_a^{-1}$

$\mathcal{P}_1$: Authentication by PAS p of entity $x \in C, T, M$, or O

$x \to p : \{\mathcal{C}^a_x, K_{xp}, N^{(1)}\}K_p$

$p \to x : \{\mathcal{C}^p_x, N^{(1)}+1\}K_{xp}$

$\mathcal{C}^p_x = \{K_x, T^{(1)}_s, T^{(1)}_e\}\, K_p^{-1}$

$\mathcal{P}_2$: Authentication of client c by TAS t

$c \to t : \{\mathcal{C}^a_c, K_{ct}, N^{(2)}\}K_t$

$t \to c : \{\mathcal{C}^p_t, \mathcal{C}^t_c, N^{(2)}+1\}K_{ct}$

$\mathcal{C}^t_c = \{K_c, T^{(2)}_s, T^{(2)}_e\}\, K_t^{-1}$

$\mathcal{P}_3$: Authentication of user u by TAS t

$c \to t : \{\{K_u, N^{(3)}\}K_u^{-1}\}K_{ct}$

$t \to c : \{\mathcal{C}^t_u, N^{(3)}+1\}K_{ct}$

$\mathcal{C}^t_u = \{K_u, T^{(3)}_s, T^{(3)}_e\}\, K_t^{-1}$

Fig. 5. Protocols for the authentication of a tenant user in Dike.

the public key K_a of the attestation server. Additionally, the TAS, MDS, and OSD know the public key K_p of the PAS, and each client securely obtains the public keys K_p, K_t, K_m, and K_o.

The privacy certification authority v signs the public part of the attestation identity key $K_{AIK(x)}$ previously provisioned for entity x through a standard protocol [30]. Then the remote attestation server a verifies the authenticity of a node (software/hardware) configuration w through protocol $\mathcal{P}_0$. If the verification succeeds, the requesting entity obtains an attestation certificate signed with the private key K_a^{-1}. Then each entity x, standing for t, m or o, uses the protocol $\mathcal{P}_1$ to get authenticated by the PAS and receive the certificate $\mathcal{C}^p_x$. A tenant client c uses the protocol $\mathcal{P}_2$ to get authenticated by the TAS t, establish a secure channel over secret key K_{ct}, and receive the certificate $\mathcal{C}^t_c$.

On behalf of user u, an authenticated client c applies the protocol $\mathcal{P}_3$ to receive a user certificate $\mathcal{C}^t_u$ signed by the TAS. The client c ensures its authenticity by holding the secret key K_{ct}, and the user u proves her authenticity by having the client sign the request with the private key K_u^{-1}. Using protocol $\mathcal{P}_4$, the client c agrees on a secret key K_{cm} with the metadata server m. Then, the client applies protocol $\mathcal{P}_5$ to request the data ticket $\mathcal{T}^m_{c(u)o}$ on behalf of user u to access file f. In addition to the public keys K_c and K_o that specify the client and the object server, the data ticket contains the handle ($handle$) of file f and the file permissions ($perms$) applying to user u.

With protocol $\mathcal{P}_6$, the client transmits to OSD o the request for operation op on the $handle$, encrypted with $K_{c(u)o}$, and receives back the reply ($opreply$). The secret key $K_{c(u)o}$ is encrypted with the public key K_o. The above steps are similar in the case of a provider user accessing the filesystem, although both the user and client are directly authenticated by the PAS rather than the TAS.

$\mathcal{P}_4$: Secret K_{cm} shared between client c and server m

$c \to m : \{\mathcal{C}^p_t, \mathcal{C}^t_c, K_{cm}, N^{(4)}\}K_m$

$m \to c : \{N^{(4)}+1\}K_{cm}$

$\mathcal{P}_5$: Data ticket for file f to client c on behalf of user u

$c(u) \to m : \{\mathcal{C}^t_u, f, N^{(5)}\}K_{cm}$

$m \to c(u) : \{\mathcal{T}^m_{c(u)o}, N^{(5)}+1\}K_{cm}$

$\mathcal{T}^m_{c(u)o} = \{K_c, K_o, perms, handle, T^{(5)}_s, T^{(5)}_e\}\, K_m^{-1}$

o may be replaced by object server group g for efficiency.

$\mathcal{P}_6$: Run op on $handle$ for client c on behalf of user u

$c(u) \to o : \{\mathcal{T}^m_{c(u)o}, op, N^{(6)}\}K_{c(u)o}, \{K_{c(u)o}\}K_o$

$o \to c(u) : \{opreply, N^{(6)}+1\}K_{c(u)o}$

Fig. 6. Protocols for authorizing a filesystem request in Dike.

We use timestamps to ensure the freshness of the issued tickets and certificates. We detect replay attacks in the exchanged messages using nonces and having them increased by one before they are returned. Additionally, in several protocols ($\mathcal{P}_0$, $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_4$) we use nonces to detect potential unauthorized modification of the request and confirm the authenticity of the recipient server.

4.5 Multiview Authorization

In the *multiview* authorization methodology that we introduce, the filesystem selectively makes the metadata accessible to different entities in the form of views (Fig. 7). The filesystem administrator of the provider uses the *provider view* to specify permissions for entire tenants or individual users. Instead a tenant administrator uses a *tenant view* to configure the metadata made accessible by the provider to the respective tenant. A user obtains filtered access to a subset of the provider or tenant view according to the applicable permissions.

The authorization policy of the filesystem is specified in the permissions maintained for each file by the MDS. We support two types of access permissions for a file or folder, the Unix and the Access Control List (ACL). The MDS isolates on distinct data structures (e.g., linked lists) the policies of a file that apply to different tenants. It additionally stores separately the policy for the provider users.

A Dike access policy can configure a file as private or shared across the users of a single or multiple tenants and let the filesystem natively support cross-tenant accesses. The certification hierarchy in general-purpose public-key cryptography can have an arbitrary number of levels. In the past, this possibility has received negative criticism due to the potential complexity that it introduces [35]. Instead, Dike only uses a two-level hierarchical structure to let the TAS of each tenant be certified by the PAS. Dike also differs from the multi-realm Kerberos protocol, in which remote accesses require tickets of the remote realm to be granted either directly, or hierarchically through a common ancestor [36].

4.6 Inheritance and Common Permissions

Cloud storage systems generally handle an enormous number of files. Managing several permissions for each file in-

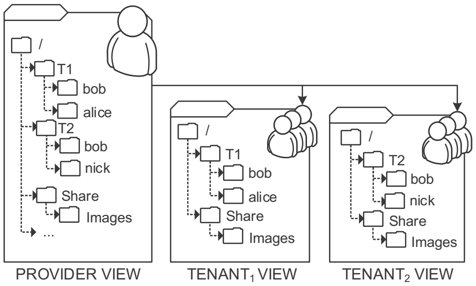

Fig. 7. Provider and tenant view of the filesystem metadata in the multiview authorization methodology of the Dike system.

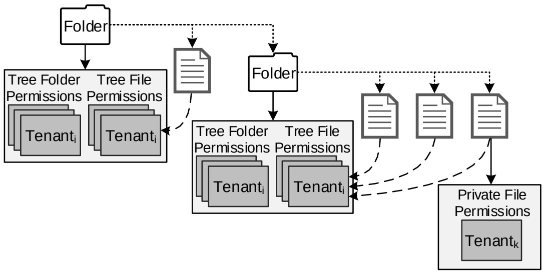

Fig. 8. Inheritance and common permissions in Dike with tree file and folder permissions. At the bottom right corner, we also show an example of private file permissions that are limited to individual files.

volves considerable space and time requirements. A related study found that users rarely change the access rights of single files [37]. As users prefer to create new files with permissions inherited from the parent folder, inheritance in storage access control is generally recommended.

Dike supports the inheritance of access permissions as a convenience to the user. In order to reduce the load of the object and metadata servers, we also allow common permissions to be shared among the different files of the same folder (Common type in Table 1). From our isolation goal, it follows that the inheritance and common permissions are enforced separately within each tenant.

The *tree folder permissions* refer to the permissions of the folder itself, and the *tree file permissions* refer to the permissions of the files directly contained in the folder. We collectively call *tree permissions* the folder and file permissions of the folder. By default the tree permissions are initialized according to the environment of the user (similar to the Unix `umask` semantics). Inheritance applies trivially in this case in the sense that all files and folders are created with the same default settings (Default type in Table 1).

As a second option, our design allows an authorized user to explicitly specify the settings of the tree permissions in a folder (Custom type in Table 1). Then, the specified permissions are inherited into the files and folders of the subtree rooted at the folder. For implementation simplicity, we allow the tree folder permissions to be physically copied to the underlying folders. Instead, the tree file permissions are common across the children files without being copied separately to each child.

Additionally, we can explicitly set the permissions of an individual file to *private file permissions* (Custom in Table 1). These are distinct from the tree file permissions inherited from the parent folder. In Fig. 8 we present an example of two folders and multiple files adopting the tree file permissions or the private file permissions. We summarize the characteristics of the different permissions types in Table 1.

TABLE 1
Permission types supported in Dike. The Custom type refers to user-specified permissions potentially different from the Default.

Permission Type	Inherited	Common	Default	Custom
Tree Folder	✓		✓	✓
Tree File	✓	✓	✓	✓
Private File				✓

The file and folder permissions are *statically* specified either manually by an authorized user or automatically with the default settings. Alternatively, our design lets the system determine *dynamically* the tree permissions of a folder based on the permissions of the respective subtree. For instance, the tree file permissions of a folder can be automatically set identical to the permissions of the first file created in the folder. Over time, the tree file permissions can be adjusted according to the permissions that apply to the majority of the files in the underlying subtree. A similar approach can be followed for the tree folder permissions.

4.7 Security Analysis

Next we analyze the properties of the Dike architecture by explaining the secure steps followed by the protocols and describing the potential implications of tenant and provider attacks.

Permissions granted In Dike, the provider cooperates with the tenants to enforce the filesystem access control. Only authenticated clients and users are allowed to access the filesystem. The data is only disclosed to or modified by users authorized by the system according to the permissions specified by the owner user. A user can only access the data belonging to or shared with her authenticating tenant. Accordingly, a user operation is restricted to the minimum of the file access permissions specified in the policy of the file owning tenant and the user authenticating tenant.

Authentication enforcement The client and user are authenticated by the TAS (or PAS) before they can receive the authorization data ticket from the MDS to securely access a particular OSD. The MDS verifies the authenticity of the requesting client and user before it signs the data ticket with its private key to make any unauthorized tampering detectable. Then, the authenticated user accesses a file according to the policy specified at the MDS. The OSD uses the public key of the MDS to verify the authenticity of the data ticket and returns the requested data encrypted with the secret key agreed with the client.

Authorization enforcement The data ticket is securely transferred from the MDS to the OSD over the K_{cm} and the $K_{c(u)o}$ secret keys through the client. The OSD is responsible to check the requested operation against the permissions securely contained in the data ticket and enforce the access control specified at the MDS. The authorized file access is restricted to a specific tenant and user as requested by the protocols that establish the key K_{cm} and ticket $\mathcal{T}^{m}_{c(u)o}$. Direct filesystem accesses from the provider users similarly require user authentication from the PAS before they can establish communication with the MDS.

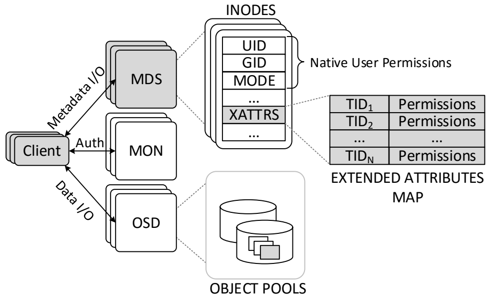

Fig. 9. Prototype implementation of the Dike multitenant access-control system, based on the extended attributes of the filesystem. We use gray background for the entities that we added to or modified from the Ceph object-based filesystem.

Sharing violation The support for file sharing between different tenants is enabled through activation of the necessary access permissions at the MDS. The clients or users from different tenants cannot collude to bypass the MDS policy enforcement because an issued data ticket specifies the authorized client and user belonging to a specific tenant. Potential cross-tenant policy violation is prohibited by design because the filesystem access is restricted through the tenant view, and the permissions of different domains are stored and managed separately by the MDS.

Tenant attacks An attacker is unlikely to penetrate the client of a tenant and impersonate a legitimate user, because a user certificate cannot be requested without access to the user's private key within an attested client. The harm from a tenant user impersonation is limited to the private or shared files that are accessible by the compromised tenant, and cannot affect the system-wide access policy. Depending on the severity of the attack, possible steps to isolate the attacker include the revocation of access permissions to the compromised user by the tenant and disabling of file sharing to the penetrated tenant by the provider.

Provider attacks The attack is more challenging in the unlikely case that it compromises an administrator account of the provider and exposes the system permissions. The implications of such an attack can be contained if the tenants apply external protection techniques or the provider supports remote monitoring from outside the cloud to detect the incident [26], [38]. More generally, in lack of trust to the provider, a tenant may externally apply techniques of encryption, hashing, auditing and multi-cloud replication to strengthen end-to-end confidentiality and integrity, or ensure data restoration in case of a provider compromise [39].

5 System Prototype

Next we describe our implementation of the Dike multitenant access control over a distributed filesystem. The prototype development is based on Ceph, a flexible platform with scalable management of metadata and extended attributes [8].

5.1 Outline of Ceph

There are four components in Ceph: the clients provide access to the filesystem, the metadata servers (MDSs) manage the namespace hierarchy, the object storage devices (OSDs) store objects reliably, and the monitor (MON) manages the server cluster map [8]. Although data and metadata are managed separately, they are both redundantly stored on OSDs.

A registered client shares a secret key with the MON. When a user requests a filesystem mount, the respective client receives a session key after being authenticated by the MON. The session key is encrypted with a secret key shared between the client and the MON. The client with the session key securely requests the desired Ceph services from MON and receives an authenticating ticket for the MDSs and OSDs. The authenticating ticket is encrypted with a secret key that is shared among the MON, MDSs and OSDs [40].

The MDS maintains in an entry of a session map the state of communication with a client. Unless it fails, a session remains active until the client unmounts the filesystem. At first communication with an MDS, the client uses the ticket to initiate a new session. The MDS receives a message of type MClientSession from the client, initializes the session state, and sends back a capability (ticket) for the root directory. From the capability the client derives an object identifier and the placement group of OSDs that contain the object replicas.

5.2 Support for Multitenant Access Control

We expanded Ceph to experiment with the scalability of native multitenant access control according to the Dike design (Fig. 9). Apart from the added multitenancy support, our current prototype implementation relies on the existing authentication and authorization functionality of Ceph. As a result, the tasks of both the TAS and PAS are currently carried out in part by the MON.

Session In a filesystem mount request to an MDS, a client has to uniquely identify the accountable tenant. In our current prototype, we derive a unique tenant identifier (TID) by applying a cryptographic hash function to the public key of the tenant (we use RIPEMD-160). The client embeds the TID into an expanded MClientSession request, and sends it to the MDS over the secure channel established with the session key. The MDS extracts the TID from the received message, and stores it in the session state. The secure session between the filesystem and an authenticated client can only serve the actions permitted to the users of the identified tenant.

Permissions Our current implementation only supports Unix-like permissions for users and groups, but it makes straightforward the addition of access-control lists in a future version. The Ceph version that our prototype relied on did not directly support access-control lists, but newer Ceph releases have been gradually adding this feature. Based on the supplied TID, a client obtains tenant view of the filesystem for access by a tenant user. The permissions of the tenant view are stored in the extended attributes of the filesystem. For global configuration settings, we also support the provider view, which enables full access permissions to the administrator of the entire filesystem. The respective permissions are stored in the regular inode fields.

Separately for each tenant, a folder stores two types of permissions: the *tree folder permissions*, which control the access of the folder; and the *tree file permissions*, which

TABLE 2
New methods that we added to class CInode of Ceph for managing the tenant permissions of an inode in the Dike prototype.

Method	Description
`bool check_tenant_perm()`	Check tenant permission
`void grant_tenant_perm()`	Grant tenant permission
`void revoke_tenant_perm()`	Revoke tenant permission
`void set_perm_uid()`	Set user ID
`void set_perm_gid()`	Set group ID
`void_t set_perm_mode()`	Set file permissions
`uid_t get_perm_uid()`	Return user ID
`gid_t get_perm_gid()`	Return group ID
`mode_t get_perm_mode()`	Return file permissions

control the access of the files contained in the folder. We allow a collection of files to share the permissions specified in the tree file permissions of their parent folder.

Alternatively, a user can explicitly set the access permissions of a particular file. Accordingly, we create new *private file permissions* for the respective tenant. In order to authorize the file request of a user, the MDS initially searches the file's extended attributes for the permissions attribute. If a private permissions attribute exists, it is used for the authorization. Otherwise, the MDS applies the tree file permissions stored in the extended attributes of the parent folder.

Development We developed the Dike prototype by modifying the client and the metadata server (MDS) of the CephFS distributed filesystem. The implementation required the addition of 2023 commented C++ lines into the codebase of Ceph v0.61.4 (Cuttlefish). We extended the client interface with two new operations to grant or remove tenant access for a file or a folder (*ceph_grant_tenant_access* and *ceph_revoke_tenant_access*). These operations specify the ownership and permission on a file or folder in a way similar to `setattr`, but they additionally accept a tenant ID as an argument.

For each of these two operations we also added a corresponding handler at the MDS that invokes the respective method at a CInode object to assign or remove tenant permissions. In the CInode class of Ceph, we added a total of nine new operations to set and retrieve the permissions of tenants and individual users (Table 2). The Dike implementation required to modify all the filesystem functions of the original Ceph related to permissions handling, including the inode constructor.

We manage the extended attributes in memory as key-value pairs stored in a C++ map structure (red-black tree). As key, we use the string `tid:model:perm`, in which the field `tid` holds the tenant identifier, the `perm` specifies the type of permissions, and the `model` is set to `"UNIX"` for Unix permissions or `"ACL"` for access control list. In the Unix model, we set the value of the key-value pair to `"uid:gid:mode"`, where the `uid` and `gid` fields refer to the user and group identifiers, and the `mode` contains the file permissions.

If the client uses the provider view, then we directly update the regular inode of the filesystem. Otherwise we save the user/group IDs and the file permissions into extended attributes keyed by TID; furthermore, we update the regular inode of the filesystem according to the user/group IDs and the file mode of the parent inode.

A file capability is only sent to a client whose authenticating tenant is permitted to access the file. In order to enforce the access policy, we expanded the returned capability of Ceph to include the tenant identifier and the respective file ownership metadata. A client cannot directly read or write the access control information stored in the extended attributes of a file. Instead, only the filesystem is allowed to access the extended attributes on behalf of authorized client requests.

Finally, we implemented an administrator tool that combines the functionalities of the Unix `chmod` and `chown` utilities, but accepts a tenant ID as an additional argument. The tool invokes the new calls that we added to the client interface and can be used by a file system administrator to assign or revoke tenant permissions on files and folders.

6 Performance Evaluation

We experimentally examine the scalability of Dike along several parameters, and compare the respective overhead to that of Ceph and other systems.

6.1 Experimentation Environment

We conducted several experiments on a local cluster and the Amazon public cloud using three different benchmarks.

Local cluster Our first testbed relies on an isolated local cluster consisting of 64bit x86 servers running Debian 6.0 Linux. We used up to a total of 11 machines: 5 machines for the filesystem nodes and 6 machines for the client hosts. Each filesystem server is equipped with 1 quad-core x86-64 CPU at 2.33GHz, 3-6GB RAM, 2 SATA 7.2KRPM 250GB hard disks, 1Gbps link and Linux v3.9.3. A server with 6GB RAM is used as MDS. From the remaining 4 servers with 3GB RAM, 3 are OSDs and 1 is MON. Each OSD uses the first disk to store the root filesystem and a journal file of 1GB size, and it has the second disk formatted with the XFS filesystem to store objects. Each client host is equipped with 2 quad-core x86-64 CPUs at 2.33GHz, 4GB RAM, 2 SATA 7.2KRPM 500GB hard disks, 1Gbps link, and runs Linux v3.5.5 with Xen v4.2.1. We set up each client guest with 1 dedicated core, 512MB RAM, 2 blktap devices for root and swap partitions, and Linux v3.9.3.

Cloud platform Our second testbed consists of EC2 instances from the US East region of the Amazon Web Services (AWS). We use a total of 36 instances: 3 instances of type ”m1.large” (4x64bit cores, 15GB RAM) as fileservers, 32 instances of type ”t1.micro” (1x64bit core, 615MB RAM) as microbenchmark clients, and 1 instance of type ”c.medium” (2x64bit cores, 1.7GB RAM) as application client. All instances run Red Hat Enterprise Linux Server v6.4 with Linux v3.9.3. On the three fileservers we alternatively run three servers of Ceph, Dike, GlusterFS, and HekaFS with replication factor 3 (for GlusterFS and HekaFS see §6.2.2, §7). GlusterFS and HekaFS manage both data and metadata on all three fileservers. Instead, Ceph and Dike run an OSD instance on each fileserver, but also use two of the fileservers to additionally run the MDS and the MON, respectively. (Ceph uses a single MDS, because the version that we used provides unstable support of multiple active MDSs.)

Benchmarks We measure the metadata performance using the mdtest v1.9.1 from LLNL [41]. This is an MPI-based

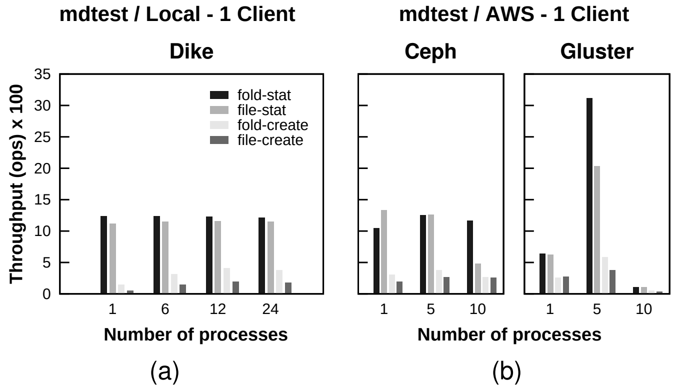

Fig. 10. The mdtest throughput of 1 client is maximized at 12 processes in the local cluster (a), and 5 processes in AWS (b). We show representative measurements for Dike (a), along with Ceph and GlusterFS (b).

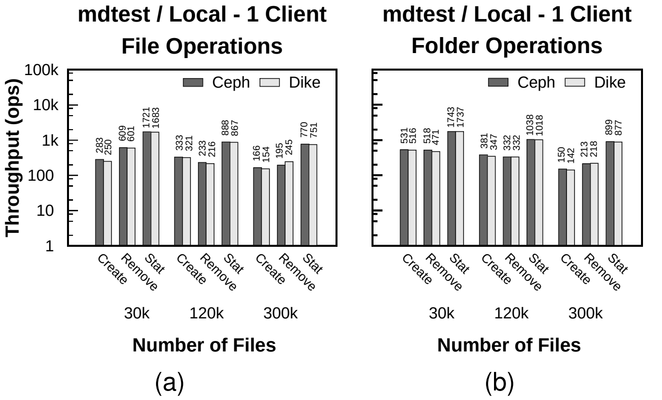

Fig. 11. Performance of mdtest at increasing number of files across Ceph and Dike. We use 36 tenants in Dike and 1 client in both systems.

microbenchmark running over a parallel filesystem. Each spawned MPI task iteratively creates, stats and removes a number of files and folders. At the end of the execution, the benchmark reports the throughput of different filesystem operations.

In a preliminary experiment, we created a total of 31,104 files and folders equally divided across the benchmark tasks. With one tenant over our two testbeds and different filesystems we measured the throughput for different numbers of processes in a single client. We found that the mdtest throughput is roughly maximized at 12 processes for the local client and 5 processes for the AWS client. In Fig. 10 we show representative measurements for (a) Dike over the local cluster along with (b) Ceph and GlusterFS over AWS. Accordingly, we configure the number of mdtest processes per client equal to 12 in the local cluster and 5 in AWS for the rest of the document.

In the MapReduce application workload, we used Stanford's Phoenix v2 shared-memory implementation of Google's MapReduce. We study the reverse index, which receives as input a collection of HTML files, and generates as output the full-text index with links to the files. Our dataset contains 78,355 files in 14,025 folders and occupies 1.01GB. We measure the latency of index build broken down across several metadata operations.

In the Linux Build application workload, we store the source of the Linux kernel (v3.5.5) in a shared folder of the filesystem. Then we use soft links to make the code accessible in private folders of the tenants. We measure the total time to create the soft links and build the system image.

In our experiments, we treat as main measured metric the throughput of mdtest, the index build time of MapReduce, and the compilation time of Linux Build. We repeated the experiments at least 3 times and as many times as needed (up to 20) to constrain the 95% confidence-interval half-length of the main metric within 5% of the average value.

6.2 Experimentation Results

Across the three different workloads, we examine the performance overhead of Dike over Ceph at different numbers of clients after a preliminary experimentation with the filesystem size. Then we study the performance and overhead in systems with up to one thousand of configured tenants or more. Finally, we evaluate the performance improvement arising from the activation of common permissions in Dike.

6.2.1 *Scalability*

In our first set of experiments, we examine the comparative performance of Dike over Ceph at an increasing number of clients across the different benchmarks.

mdtest We start with the mdtest performance (measured in operations per second, or ops) in the local cluster. Before examining different numbers of clients, we do a basic experiment about the effect of the filesystem size to the system throughput in Dike configured with 36 tenants. We consider alternative collections of 30k, 120k and 300k files, and equally distribute each collection across 10 folders. With 1 client accessing the files of a single tenant in Fig 11a, we find Dike to reduce the throughput of Ceph by a moderate percentage of 0-11.5% across the three operations and actually increase it by 25.8% in one case (remove/300k). In comparison to stat, the reduction is higher in the create and remove operations, most likely due to the locking contention of the update activity that they involve. At 30k files, the create throughput of Dike (250ops) is 11.5% below Ceph (283ops), unlike the stat throughput of Dike (1,683ops) that is only 2.2% below Ceph (1,721ops). Increasing the files from 30k to 300k reduces noticeably the measured throughput across both systems but the relative results remain comparable. We also examine the performance of folder operations in Fig 11b to find out that increasing the number of files also reduces substantially the performance across all the operation types. Thus, we establish some basic understanding of the system sensitivity to the workload size.

In Fig 12 we examine different numbers of clients on an equal number of distinct tenants over the local cluster. The clients equally divide the creation of 31,104 files, which are either located in a shared folder (Shared), or equally distributed across 36 private folders, one per client (Private). We measure the throughput of file create, remove and stat across the two folder types. In Fig 12a we consider the operation throughput of Dike at increasing number of clients up

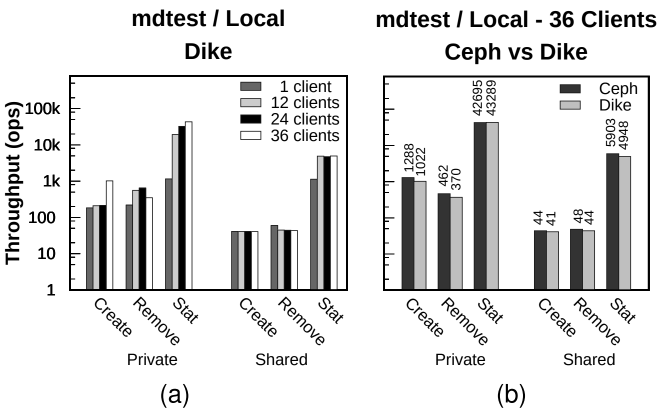

Fig. 12. Performance of mdtest file operations across different numbers of clients. We consider files distributed over private or shared folders of Dike (a) in comparison with the original Ceph (b). Dike uses 36 tenants to assign a separate tenant per client.

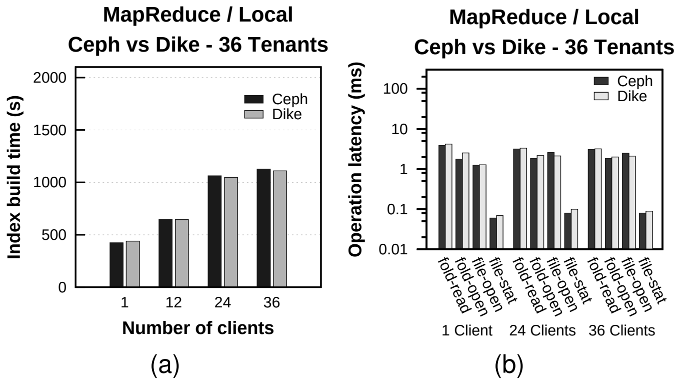

Fig. 13. Performance comparison of MapReduce over Ceph and Dike across different number of clients. We configure Dike with 36 tenants and assign a separate tenant per client.

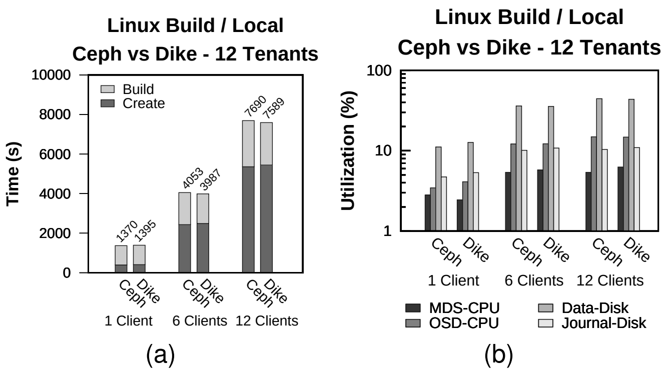

Fig. 14. (a) Performance and (b) resource utilization of the Linux build workload between Ceph and Dike across different numbers of clients. We use Dike with 12 configured tenants. For the OSD case, we depict the average utilization across the three OSD nodes of the system.

to 36 (432 processes total). The throughput of stat increases almost linearly (37.4x) in private folders, but an order of magnitude less (4.4x) in shared folders. The performance of create and remove increases much less (e.g., create/private by 5.6x) or even drops at 36 clients (e.g., remove/shared by 27.3%).

In Fig 12b we compare the performance of Ceph and Dike at 36 clients under mdtest. In private folders, Dike improves marginally the stat performance of Ceph by 1.4%, but reduces the create performance by 20.6% (from 1,288ops down to 1,022ops). In the shared folder, Dike reduces the stat performance by 16.2%. Overall, Dike reduces the throughput of Ceph roughly by 0-21%, because it updates both the inode and extended attributes to provide multitenancy. Furthermore, folder sharing among the clients reduces consistently the performance of the private folders by a factor of up to 29.1x in Ceph and 25.1x in Dike.

MapReduce We subsequently compare the performance of MapReduce over Ceph and Dike across different numbers of clients. In Fig 13a we measure the build time of the reverse index operation over the local cluster. The collection of indexed files is stored in a shared folder and the created indices are held in memory. We configure Dike with 36 tenants, and assign a distinct tenant to each client. We vary the number of clients between 1 and 36 and notice that the performance of Dike remains similar to that of Ceph. The highest overhead of Dike appears with 1 client at 3.8%, in which case Ceph takes 423.6s to build the index and Dike takes 439.8s.

In Fig 13b we examine the latency of filesystem metadata operations at the client under MapReduce. In most cases, we find the latency measured in Dike to be comparable to that of Ceph across the different operations and number of clients. Notable exception is the opendir (fold-open) that takes 10.5-42.7% more in Dike than Ceph when the number of clients drops from 36 down to 1. In the case of readdir (fold-read) the respective overhead of Dike is much lower in the range 4.9%-10.5%. As already explained in the implementation, we attribute the Dike overhead to the extra cost of reading the tenant permissions from the extended attributes.

Linux Build In Fig. 14 we compare the performance of the Linux compilation over Ceph and Dike with the common permissions disabled. In Fig. 14a, we measure separately the average time to create the soft links and build the system image. As the number of clients increases from 1 to 12, the overhead of Dike remains relatively low. In particular, it drops from 4.6% to 1.6% in link creation and from 0.7% to -8.1% in image build. Indeed, Dike with 12 clients takes 2,145s which is lower than 2,334s in Ceph. We notice the highest increase in the experiment duration at 1 client, as Dike takes 1,395s (1.8% higher) instead of Ceph that takes 1,370s. By looking at the resource utilization of the MDS and the OSDs in Fig. 14b, we observe Ceph with 1 client to utilize 3.5% the CPU and 11.1% the data disk of the OSDs. Instead, Dike utilizes these two resources 4.1% (19.1% higher) and 12.7% (14.0% higher), respectively, leading to slighly longer experiment duration, as pointed out above for the case of 1 client.

6.2.2 *Comparative Multitenancy Overhead*

mdtest Next we run up to 32 mdtest clients in the AWS testbed (1 t1.micro EC2 instance/client). The clients equally divide the creation of 48,000 files, and each client equally divides the respective number of files among its processes. In Dike we alternatively create 1,000 or 5,000 private folders

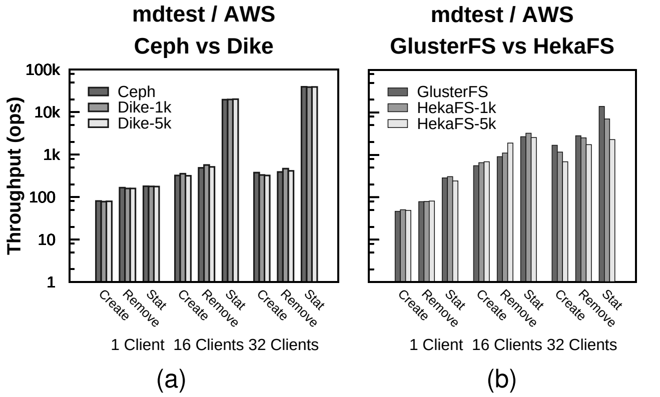

Fig. 15. Throughput comparison of filesystem operations in mdtest between Ceph and Dike along with GlusterFS and HekaFS, with the number of tenants alternatively set equal to 1,000 or 5,000.

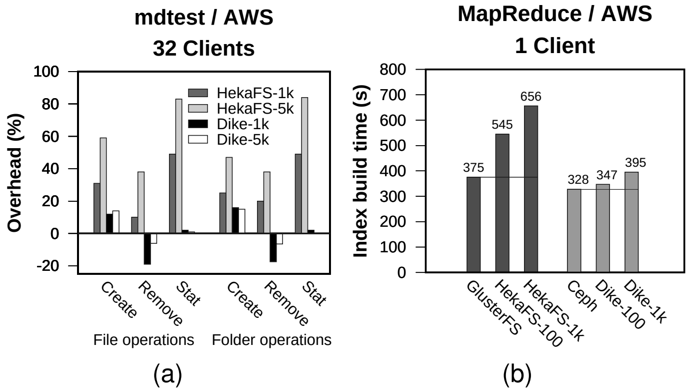

Fig. 16. Comparison of relative overheads by Dike over Ceph and HekaFS over GlusterFS in the filesystem throughput of (a) mdtest with up to 5,000 tenants and (b) MapReduce with up to 1,000 tenants.

and activate access permissions for a single tenant per folder to emulate 1,000 (Dike-1k) or 5,000 distinct tenants (Dike-5k), respectively.

In Fig. 15a we show the total throughput of mdtest file operations in Ceph and Dike over AWS. Quite reasonably, we notice a higher number of clients to increase the system performance, but Dike only adds a limited overhead to Ceph. At 1 client, Ceph serves the file create at 81ops, Dike-1k at 78ops and Dike-5K at 80ops. At 32 clients, Dike-1k only reduces the Ceph throughput by 0-12%, and the 5k configuration adds an extra 2% overhead in Dike with respect to the original Ceph. In some cases, Dike even improves the performance of the file operations (up to 19.5% at 32 clients with remove), most probably due to the reduced update contention across the private filesystem folders.

GlusterFS is an open-source, distributed filesystem from RedHat. It supports translator layers for the addition of extra features. HekaFS is a cloud filesystem implemented as a set of translators over GlusterFS. In order to isolate the identity space of different tenants, HekaFS uses a mapping layer to translate the local user identities of the tenants to globally-unique identities [4]. HekaFS appears to strictly store the files of different tenants on distinct private folders, which makes it unclear whether it currently supports secure file sharing between tenants [4].

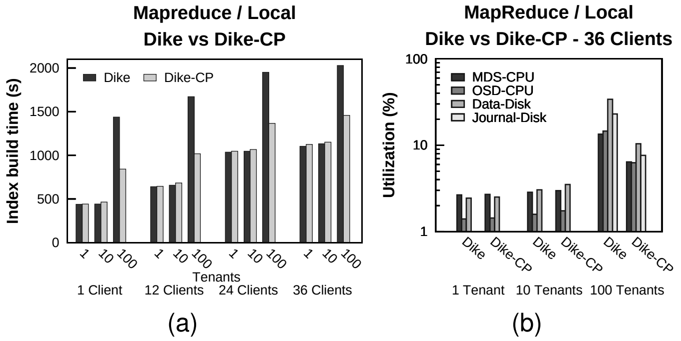

Fig. 17. (a) Comparative advantage of activating common permissions in Dike (Dike-CP) at different numbers of tenants and clients with local MapReduce. (b) Comparative resource utilization across Dike and Dike-CP in local MapReduce. For the OSD case, we depict the average utilization across the three OSD nodes.

In Fig. 15b we compare the throughput of mdtest file operations across GlusterFS and multitenant HekaFS on AWS. At increasing number of tenants from 1 up to 1k or 5k, performance remains about the same with 1 client and slightly improves with 16 clients, but substantially drops with 32 clients. For instance, the stat throughput of 32 clients drops from 13,818ops in GlusterFS to 7,017ops (49.2%) in HekaFS-1k, and 2,290ops (83.4%) in HekaFS-5k.

Fig.16a summarizes our comparative results of Dike and HekaFS at 32 clients. Overall, the performance overhead remains similar across file and folder operations. However, the overhead of file and folder create reaches 12-16% in Dike-1k and 14-15% in Dike-5k. Instead, the overhead in both file and folder stat of HekaFS over GlusterFS approaches 49% with 1k tenants and 84% with 5k tenants, i.e., nearly up to two orders of magnitude higher than that of Dike (0-2%).

MapReduce In Fig. 16b we further explore the multitenancy overhead using MapReduce over AWS. We use 1 client over a filesystem configured with 100 and 1k tenants, respectively. Based on the measured time of index build, the overhead of Dike over Ceph lies in the range 6-20%, and that of HekaFS over GlusterFS in 45-75%. We conclude that HekaFS under either mdtest or MapReduce ends up at much higher performance overhead in comparison to Dike with a moderate (a hundred) or large number (a thousand) of tenants.

6.2.3 *File Sharing and Common Permissions*

MapReduce In Fig 17a, we measure the performance advantage of activating common permissions in Dike (Dike-CP) on the local cluster configured with 1, 10 or 100 tenants. Each file in Dike maintains separate permissions for each tenant. On the contrary, the files in Dike-CP adopt the tree file permissions of their parent folder. If we set the number of tenants to 1 or 10, then the index build time is comparable across Dike and Dike-CP. For instance, with 36 clients and 1 tenant, Dike-CP slightly increases the index build time of Dike by 1.9% (from 1,104s to 1,125s). In fact, the results become more interesting if we raise the number of tenants to 100. Dike-CP achieves a substantial reduction of the build time by 41.4% (from 1,438s to 843s) at 1 client, and by 28.1% (from 2,029s to 1,458s) at 32 clients.

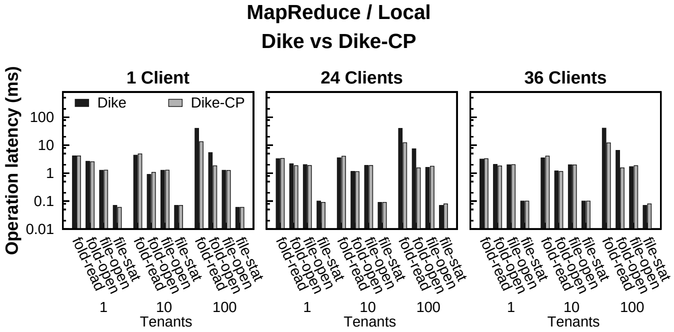

Fig. 18. Throughput comparison of filesystem operations across Dike and Dike-CP in local MapReduce.

Each file access in Dike involves the retrieval of a file-permissions object, which increases the required processing time in both the MDS and the OSDs along with the disk I/O at the OSDs. On the contrary, accessing the files of a folder in Dike-CP can be authorized by retrieving only once the tree file permissions of the folder. In Fig. 17b we show details of resource utilization in the MDS and the (average of) OSDs for Dike and Dike-CP. With 1 tenant, the most demanded resource in the filesystem is the MDS CPU with measured utilization at 2.7% in both Dike and Dike-CP. However, as we increase the tenants to 10 or 100, the data disk of the OSDs becomes the most demanded resource. The respective utilization is 3.1% and 34.0% in the case of Dike, and 3.5% and 10.4% in the case of Dike-CP. Therefore, at 100 tenants, Dike-CP reduces the utilization of the bottleneck resource by more than a factor of 3, which translates into the substantially reduced build time observed in the previous paragraph.

In Fig 18 we explore the latency of individual file operations across different numbers of tenants in Dike and Dike-CP. We notice that readdir (fold-read) is highly sensitive to the number of tenants. As the tenants increase from 1 to 100 at 1 client, the latency of readdir grows 9.7x (from 4.1ms to 40.3ms) in Dike, and 3.2x (from 4.2ms to 13.4ms) in Dike-CP. Essentially, Dike-CP reduces the latency of readdir by a factor of 3 in comparison to Dike. However, the latency across the different operations remains relatively insensitive to the different numbers of clients that we tried.

Linux build Additionally, in Fig.19 we examine the Linux Build benchmark with 1 up to 12 clients and dedicated tenant per client with common permissions in Dike activated or not. In the case of 1 tenant and 1 client, Dike-CP slightly increases by 7.4% the total benchmark time of Dike (to 1,491s from 1,389s). At the other extreme with 100 tenants and 12 clients, Dike-CP only achieves a limited benefit of 2.9% with respect to Dike, as Dike-CP reduces the experiment time from 7,896s to 7,664s. These results can be justified by the fact that the multitenancy support provided by Dike in Linux Build already incurs low metadata-management overhead (2% over Ceph).

Overall, we conclude that Dike adds multitenancy support to Ceph at limited performance overhead below 21%. Instead, the overhead of HekaFS over GlusterFS is higher, up to 84% in some cases.

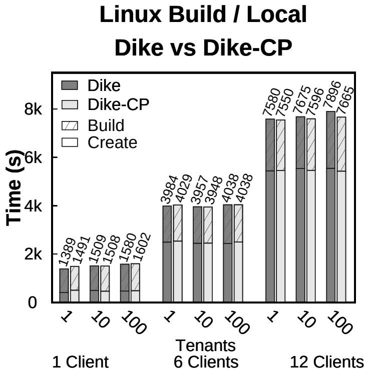

Fig. 19. Limited comparative advantage of Dike-CP over Dike in Linux build, running over the local cluster with the number of configured tenants and clients as specified at the x axis.

7 Related Work

We summarize representative related research in the recent and older published literature about security in distributed filesystems, virtualization, and cloud storage with respect to our present work.

Distributed Filesystems All the principals of a distributed system are often registered into a central directory (e.g., Kerberos [36]). Secure data transfer between clients and storage through an object-based interface is addressed in the Network-Attached Secure Disk model [42]. The Self-certifying File System prefetches and caches remote user and group definitions to flexibly support file sharing across administrative domains [35]. Plutus applies cryptographic storage to support secure file sharing over an untrusted server [43]. It aggregates into filegroups the files with identical sharing attributes, and lazily revokes readers by deferring the re-encryption of files until their update. In trust management, a credential binds the keys of principals to the authorization to perform certain tasks [6]. The users from different institutions are mapped to an authoritative list of identifiers in order to maintain uniform ownership and permission metadata over Lustre [44]. Hypergroup is an authorization construct that we previously introduced for the scalable federation of filesystems in grid environments [45].

In object-based filesystems, an authenticated principal obtains a capability from the metadata service to access an object server [8]. Secure capabilities based on public-key cryptography were previously shown to achieve scalable filesystem authorization [40]. The extended capability of the Maat protocol securely authorizes I/O for any number of principals or files at a fixed size through Merkle hash trees [27]. IDEAS applies inheritance for scalability in role-based access control [46]. Secure protocols have been proposed to enforce authentication, integrity and confidentiality at the data traffic of the Google File System [47]. However, the above research does not directly address the multitenancy of consolidated cloud storage.

Virtualization Multitenancy isolation without sharing support was previously achieved in the filesystem by either running separate virtual machines per tenant, or shielding the filesystem processes of different tenants [48]. File-based storage virtualization is enabled by Ventana through versioning and access-control lists (ACLs) [2]. Server file ACLs have system-wide effect, while guest file ACLs are controlled by the guests. A shared proxy server at the host provides file access to local guests from networked object servers. Ventana serves multiple virtual machines, but without tenant isolation.

VirtFS uses a network protocol to connect a host-based fileserver to multiple local guests without isolating their respective principals [3]. The system stores the guest credentials either directly on the file ACLs or indirectly as file extended attributes. The scalability of Ventana and VirtFS is limited by the centralized NFS-like server running at the host. Instead we advocate the networked access of a scalable distributed filesystem (e.g., Ceph) directly by the guests.

The Manila File Shares Service is an OpenStack project under development for coordinated access to shared or distributed filesystems in cloud infrastructures [12]. The architecture securely connects guests to a pluggable storage backend through a logical private network, a hypervisor-based paravirtual filesystem, or a storage gateway at the host. Dike is complementary by adding multitenancy support to Ceph for natively isolating the different tenants at the storage backend.

Bethencourt et al. investigated the realization of complex access control on encrypted data with the encrypting party responsible to determine the policy through access attributes (ciphertext-policy attribute-based encryption) [49]. Instead, we focus on the native multitenancy support by the filesystem.

Pustchi and Sandhu introduced the multi-tenant attribute-based access control model (MT-ABAC) and considered different types of attribute assignment between the trustor and trustee tenant [29]. Ngo et al. introduce delegations and constraints for MT-ABAC over a single and multiple providers [50]. Our work is complementary to the above studies because we introduce entity definitions and protocol specifications for a cloud filesystem, and develop a system prototype for the comprehensive experimental evaluation of the multitenant access-control scalability in data storage.

Cloud Storage Different tenants securely coexist in HekaFS [4]. A tenant assigns identities to local principals, and translates the pair of tenant and principal identifier to a unique system-wide identifier through hierarchical delegation. However, HekaFS complicates sharing and applies global-to-local identity mapping that was previously criticized as cause for limited scalability [34]. We experimentally measure the overhead resulting from the multitenancy support added by HekaFS over GlusterFS.

Existing cloud environments primarily apply storage consolidation at the block level. Guests access virtual disk images either directly as volumes of a storage-area network (SAN) or indirectly as files of network-attached storage (NAS) mounted by the host [51]. The S4 framework extends Amazon's S3 cloud storage to support access delegation over objects of different principals via hierarchical policy views [52]. CloudViews targets flexible, protected, and performance-isolated data sharing. It relies on signed view as a self-certifying, database-style abstraction [53].

Our support for fine granularity of access control by the provider distinctly differentiates our work from existing systems. In particular, the AWS Elastic File System [54] only allows cross-tenant access at the granularity of entire filesystems (e.g., mount to a client) rather than files and individual users.

The abstraction of Secure Storage Regions enables the limited storage resources of a Trusted Platform Module to be multiplexed into persistent storage [55]. HAIL combines error-correction redundancy with integrity protection and applies MAC aggregation over server responses to achieve high availability and integrity over distributed cloud storage [56]. CloudProof allows customers of untrusted cloud storage to securely detect violations of integrity through public-key signatures and data encryption [57].

Secure Logical Isolation for Multi-tenancy (SLIM) has been proposed to address end-to-end tenant isolation in cloud storage. It relies on intermediate software layers (e.g., gateway, gatekeeper, guard) to separate privileges in information access and processing by different cloud tenants [58].

The data-protection-as-a-service architecture introduces the secure data capsule as an encrypted data unit packaged with security policy; the execution of applications is confined within mutually-isolated secure execution environments [11]. Excalibur introduces the policy-sealed data as a trusted computing abstraction for data security [59]. Shroud leverages oblivious RAM algorithms to hide access patterns in the datacenter [60]. We also target secure storage in the datacenter, but with native multitenancy support at the file level through the access-control metadata of object-based fileservers.

8 Discussion

In current cloud infrastructures, a shared filesystem typically stores disk images or file shares with access restricted to a single tenant without native sharing support. For instance, network-level isolation is commonly applied in the form of a Virtual Private Cloud based on encrypted communication channels or VLAN settings [30]. The file access control is fully enforced by the tenant, and security violations from vulnerable configurations compromise the resources of the tenant.

The provider neither enforces the access control of individual users, nor directly supports the storage sharing among the users of different tenants. The infrastructure primarily manages administrative entities, but leaves the virtual machines of the tenant responsible to control the access rights of individual users over local or networked resources. Consequently, the tenant administrators are burdened with the task of deploying traditional directory services to both specify and enforce user access rights, while storage sharing remains inflexible and potentially error-prone.

By moving the enforcement of access control from the virtual machine to the shared filesystem, we essentially strengthen the protection of confidentiality and integrity in the stored data over a shared consolidated environment. An individual access is permitted if it identifies securely the identities of the requesting tenant and end user according to the policy specification that has been previously configured securely into the underlying filesystem. Our prototype implementation and experimental results demonstrate the feasibility of the Dike authorization architecture with low overhead at up to thousands of tenants.

As the cloud ecosystem is increasingly populated with lightweight virtual machines, such as containers or library operating systems, we anticipate that the infrastructure will need to more actively participate in the enforcement of secure access control [17], [18]. Possible benefits from this transition include: (i) reduced amount and complexity of guest software (application or system), (ii) less sensitivity to tenant software development or configuration bugs, and (iii) lower tenant administrative overhead for service deployment or management. Accordingly, the services operated by the tenant will need to focus more on application-specific functionality and be able to outsource to the provider the basic task of secure data access or sharing by one or multiple users in the same or a different tenant.

9 Conclusions and Future Work

We consider the security requirements of scalable filesystems used by virtualization environments. Then we introduce the Dike system design including secure protocols to natively support multitenant access control. With a prototype implementation of Dike over a production-grade filesystem (Ceph) we experimentally demonstrate a limited multitenancy overhead below 21% in configurations with several thousand tenants. Our plans for future work include integration of Dike into a trusted virtualization platform in the datacenter and further experimentation with I/O-intensive applications at large scale over different object-based filesystems. We also plan to consider weaker trust assumptions as the cloud use cases expand and the resulting multitenancy environment becomes more complex. Finally, we intend to study filesystem multitenancy over multiple (possibly federated) clouds.

Acknowledgments

The authors are thankful to the anonymous reviewers, whose constructive comments helped improve the manuscript. Accessing the Amazon Web Services through credit from an "AWS in Education Research Grant" award is gratefully acknowledged.

References

[1] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," in *ACM SIGCOMM Conf.*, London, United Kingdom, Aug. 2015, pp. 183–197.

[2] B. Pfaff, T. Garfinkel, and M. Rosenblum, "Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks," in *USENIX Symp. on Networked Systems Design and Implementation*, San Jose, CA, 2006, pp. 353–366.

[3] V. Jujjuri, E. V. Hensbergen, and A. Liguori, "VirtFS: Virtualization aware File System pass-through," in *Ottawa Linux Symp.*, 2010.

[4] J. Darcy, "Building a cloud file system," *USENIX; login:*, vol. 36, no. 3, pp. 14–21, Jun. 2011.

[5] E. Wobber, M. Abadi, M. Burrows, and B. Lampson, "Authentication in the Taos operating system," *ACM Trans. Comput. Syst.*, vol. 12, no. 1, pp. 3–32, Feb. 1994.

[6] S. Miltchev, V. Prevelakis, S. Ioannidis, J. Ioannidis, A. D. Keromytis, and J. M. Smith, "Secure and flexible global file sharing," in *USENIX Annual Technical Conf., Freenix Track*, San Antonio, TX, Jun. 2003, pp. 168–178.

[7] J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *USENIX Winter Conf.*, Dallas, Texas, Jan. 1988, pp. 191–202.

[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *USENIX Symp. on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006, pp. 307–320.

[9] G. Kappes, A. Hatzieleftheriou, and S. V. Anastasiadis, "Virtualization-aware access control for multitenant filesystems," in *IEEE Intl Conf. on Massive Storage Systems and Technology*, Santa Clara, CA, Jun. 2014.

[10] D. T. Meyer, J. Wires, N. C. Hutchinson, and A. Warfield, "Namespace Management in Virtual Desktops," *USENIX; login:*, vol. 36, no. 1, pp. 6–11, Feb. 2011.

[11] D. Song, E. Shi, I. Fischer, and U. Shankar, "Cloud data protection for the masses," *Computer*, vol. 45, no. 1, pp. 39–45, Jan. 2012.

[12] https://wiki.openstack.org/wiki/Manila.

[13] http://docs.openstack.org/developer/keystone/.

[14] https://aws.amazon.com/documentation/iam/.

[15] D. Muthukumaran, D. O'Keeffe, C. Priebe, D. Eyers, B. Shand, and P. Pietzuch, "FlowWatcher: defending against data disclosure vulnerabilities in web applications," in *ACM Conf. on Computer and Communications Security*, Denver, CO, Oct. 1995, pp. 603–615.

[16] M. Mihailescu, G. Soundararajan, and C. Amza, "Mixapart: Decoupled analytics for shared storage systems," in *USENIX Conf. on File and Storage Technologies*, San Jose, CA, Feb. 2013, pp. 133–146.

[17] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, "Security of OS-level virtualization technologies," in *Nordic Conf. on Secure IT Systems*, Tromsø, Norway, Oct. 2014, pp. 77–93, Springer LNCS 8788.

[18] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie, "Jitsu: just-in-time summoning of unikernels," in *USENIX Symp. on Networked Systems Design and Implementation*, Oakland, CA, May 2015, pp. 559–573.

[19] C. Chen, H. Raj, S. Saroiu, and A. Wolman, "cTPM: a cloud TPM for cross-device trusted applications," in *USENIX Symp. on Networked Systems Design and Implementation*, Seattle, WA, Apr. 2014, pp. 187–201.

[20] J. Darcy, "HekaFS," http://hekafs.org.

[21] https://github.com/kozyraki/phoenix.

[22] M. L. Badger, T. Grance, R. Patt-Corner, and J. M. Voas, "Cloud computing synopsis and recommendations," National Institute of Standards and Technology, Tech. Rep. NIST SP - 800-146, May 2012.

[23] W. Arthur and D. Challener, *A Practical Guide to TPM 2.0 Using the Trusted Platform Module in the New Age of Security*. Apress, Jan. 2015.

[24] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: virtualizing the trusted platform module," in *USENIX Security Symp.*, Vancouver, Canada, Jul. 2006, pp. 305–320.

[25] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping Trust in Commodity Computers," in *IEEE Symp. on Security and Privacy*, May 2010, pp. 414–429.

[26] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," in *USENIX Workshop on Hot Topics in Cloud Computing*, San Diego, California, Jun. 2009.

[27] A. W. Leung, E. L. Miller, and S. Jones, "Scalable Security for Petascale Parallel File Systems," in *ACM/IEEE Conf. Supercomputing*, Nov. 2007, pp. 16:1–16:12.

[28] Y. Li, N. S. Dhotre, Y. Ohara, T. M. Kroeger, E. L. Miller, and D. D. E. Long, "Horus: Fine-grained encryption-based security for large-scale storage," in *USENIX Conf. on File and Storage Technologies*, San Jose, CA, Feb. 2013, pp. 147–160.

[29] N. Pustchi and R. Sandhu, "MT-ABAC: a multi-tenant attribute-based access control model with tenant trust," in *Intl Conf. on Network and System Security*, New York, NY, Nov. 2015, Springer LNCS 9408.

[30] R. Yeluri and E. Castro-Leon, *Building the Infrastructure for Cloud Security A Solutions View*. Apress, Mar. 2014.

[31] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, “Retro: targeted resource management in multi-tenant distributed systems,” in *USENIX Symp. on Networked Systems Design and Implementation*, Oakland, CA, May 2015, pp. 589–603.

[32] E. Pattuk, M. Kantarcioglu, Z. Lin, and H. Ulusoy, “Preventing cryptographic key leakage in cloud virtual machines,” in *USENIX Security Symp.*, San Diego, CA, Aug. 2014, pp. 703–718.

[33] A. Jøsan, M. A. Zomai, and S. Suriadi, “Usability and privacy in identity management architectures,” in *Australasian Information Security Workshop: Privacy Enhancing Technologies*, Ballarat, Australia, Jan. 2007, pp. 143–152.

[34] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell’ Agnello, A. Frohner, K. Lorentey, and F. Spataro, “From gridmap-file to VOMS: managing authorization in a grid environment,” *Future Generation Computer Systems (Elsevier)*, vol. 21, pp. 549–558, 2005.

[35] M. Kaminsky, G. Savvides, D. Mazières, and M. F. Kaashoek, “Decentralized user authenication in a global file system,” in *ACM Symp. Operating Systems Principles*, Bolton Landing, NY, Oct. 2003, pp. 60–73.

[36] B. C. Neuman and T. Ts’o, “Kerberos: An authentication service for computer networks,” *IEEE Communications Magazine*, vol. 32, no. 9, pp. 33–38, Sep. 1994.

[37] D. K. Smetters and N. Good, “How users use access control,” in *Symp. on Usable Privacy and Security*, Mountain View, CA, Jul. 2009.

[38] S. Bouchenak, G. Chockler, H. Chockler, G. Gheorghe, N. Santos, and A. Shraer, “Verifying cloud services: Present and future,” *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 2, pp. 6–19, Jul. 2013.

[39] A. Juels and A. Oprea, “New Approaches to Security and Availability for Cloud Data,” *Communications of the ACM*, vol. 56, no. 2, pp. 64–73, Feb. 2013.

[40] C. Olson and E. L. Miller, “Secure capabilities for a petabyte-scale object-based distributed file system,” in *ACM Workshop on Storage Security and Survivability*, Fairfax, VA, Nov. 2005, pp. 64–73.

[41] http://sourceforge.net/projects/mdtest/.

[42] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka, “File server scaling with network-attached secure disks,” in *ACM SIGMETRICS Conf.*, Seattle, WA, 1997, pp. 272–284.

[43] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, “Plutus: Scalable secure file sharing on untrusted storage,” in *USENIX Conf. on File and Storage Technologies*, San Francisco, CA, 2003, pp. 29–42.

[44] J. Walgenbach, S. C. Simms, J. P. Miller, and K. Westneat, “Enabling Lustre WAN for Production Use on the TeraGrid: A Lightweight UID Mapping Scheme,” in *TeraGrid Conf.*, Pittsburgh, PA, Aug. 2010.

[45] G. Margaritis, A. Hatzieleftheriou, and S. V. Anastasiadis, “Nepheli: Scalable access control for federated file services,” *J. Grid Comp.*, vol. 11, no. 1, pp. 83–102, Mar. 2013.

[46] Z. Niu, K. Zhou, H. Jiang, D. Feng, and T. Yang, “IDEAS: an identity-based security architecture for large-scale and high-performance storage systems,” University of Nebraska-Lincoln, Tech. Rep., Nov. 2008, tR-UNL-CSE-2008-0013.

[47] J. Kelley, R. Tamassia, and N. Triandopoulos, “Hardening access control and data protection in GFS-like file systems,” in *ESORICS Symp.*, Pisa, Italy, Sep. 2012, pp. 19–36, Springer LNCS 7459.

[48] A. Kurmus, M. Gupta, R. Pletka, C. Cachin, and R. Haas, “A Comparison of Secure Multi-tenancy Architectures for Filesystem Storage Clouds,” in *ACM/IFIP/USENIX Intl Middleware Conf.*, Lisboa, Portugal, Dec. 2011, pp. 460–479.

[49] J. Bethencourt, A. Sahai, and B. Waters, “Ciphertext-policy attribute-based encryption,” in *IEEE Symp. on Security and Privacy*, Berkeley, CA, May 2007, pp. 321–334.

[50] C. Ngo, Y. Demchenko, and C. de Laat, “Multi-tenant attribute-based access control for cloud infrastructure services,” *Journal of Information Security and Applications*, vol. 27-28, pp. 65–84, Apr-May 2016.

[51] D. Hilderbrand, A. Povzner, R. Tewari, and V. Tarasov, “Revisiting the storage stack in virtualized nas environments,” in *USENIX Workshop on I/O Virtualization*, Portland, OR, Jun. 2011.

[52] N. H. Walfield, P. T. Stanton, J. L. Griffin, and R. Burns, “Practical protection for personal storage in the cloud,” in *EuroSec Security Workshop*, Paris, France, Apr. 2010, pp. 8–14.

[53] R. Geambasu, S. D. Gribble, and H. M. Levy, “CloudViews: communal data sharing in public clouds,” in *USENIX HotCloud*, San Diego, CA, Jun. 2009.

[54] https://aws.amazon.com/efs/.

[55] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider, “Logical Attestation: an authorization architecture for trustworthy computing,” in *ACM Symp. on Operating Systems Principles*, Cascais, Portugal, Oct. 2011, pp. 249–264.

[56] K. D. Bowers, A. Juels, and A. Oprea, “HAIL: a high-availability and integrity layer for cloud storage,” in *ACM Conf. on Computer and Communications Security*, Chicago, IL, Nov. 2009, pp. 187–198.

[57] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, “Enabling Security in Cloud Storage SLAs with CloudProof,” in *USENIX Annual Technical Conf.*, Portland, OR, Jun. 2011, pp. 355–368.

[58] M. Factor, D. Hadas, A. Hamama, N. Har’el, E. K. Kolodner, A. Kurmus, A. Shulman-Peleg, and A. Sornioti, “Secure logical isolation for multi-tenancy in cloud storage,” in *IEEE Intl. Conf. Massive Storage Systems and Technology*, Long Beach, CA, May 2013.

[59] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, “Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services,” in *USENIX Security Symp.*, Bellevue, WA, Aug. 2012, pp. 175–188.

[60] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman, “Shroud: ensuring access to large-scale data in the data center,” in *USENIX Conf. on File and Storage Technologies*, San Jose, CA, Feb. 2013, pp. 199–213.

Giorgos Kappes is currently Doctoral Candidate at the Department of Computer Science and Engineering, University of Ioannina, Greece. Previously, he received MSc (2013) and BSc (2011) degrees from the above department. His research interests include operating systems, data storage and systems security.

Andromachi Hatzieleftheriou is currently Postdoctoral Researcher at the Systems and Networking Group, Microsoft Research, Cambridge, UK. She received PhD (2015), MSc (2009) and BSc (2007) degrees from the Department of Computer Science and Engineering, University of Ioannina, Greece. Her research interests include the design and implementation of reliable local and distributed storage systems.

Stergios V. Anastasiadis is Associate Professor at the Department of Computer Science and Engineering, University of Ioannina, Greece. He has held visiting positions at the Computer Laboratory, University of Cambridge, UK (Visiting Researcher, 2015-2016), School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Switzerland (Visiting Professor, 2009-2010), Department of Computer Science, Duke University, USA (Visiting Assistant Professor, 2001-2003), and HP Labs, HP, USA (Research intern, 1998). He received MSc (1996) and PhD (2001) degrees in Computer Science from the University of Toronto, Canada. His research interests include operating systems and distributed systems with focus on several aspects of data storage including reliability and security.