Virtualization-aware Access Control for Multitenant Filesystems

Giorgos Kappes, Andromachi Hatzieleftheriou and Stergios V. Anastasiadis
Department of Computer Science and Engineering, University of Ioannina, Greece
Email: {gkappes,ahatziel,stergios}@cs.uoi.gr

***Abstract*—In a virtualization environment that serves multiple tenants, 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 present several security requirements in multitenant filesystems. 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 use a public cloud to experimentally evaluate a prototype implementation of Dike that we developed. At several thousand tenants, our prototype incurs limited performance overhead up to 16%, unlike an existing solution whose multitenancy overhead approaches 84% in some cases.**

I. Introduction

Cloud infrastructures are increasingly used for a broad range of computational needs in private and public organizations. We call tenant an independent organization that is customer of the networked services offered by a cloud provider [1]. Access control over the resources of a multitenant environment is a challenging problem because of the enormous number of end users involved and the isolation of security administration required across independent organizations. Authentication and authorization have already been extensively studied in the context of distributed systems [2]–[6]. However, a cloud environment introduces multitenancy characteristics that warrant reconsideration of the assumptions and solution properties.

Service co-location in the datacenter offers opportunities for improved storage concentration of application or system files (e.g., shared documents, root images). Although virtual disks are attractive for their versioning, isolation and migration properties, a file-based interface can additionally support fine-grained controlled sharing, easy resource administration, and file-level performance optimizations. Existing file-based solutions face scalability limitations because they either lack support for multiple tenants, rely on global-to-local identity mappings for multitenancy, or have the guests and a centralized filesystem (or proxy) sharing the same physical host [7]–[9].

For data and metadata scalability we rely on an object-based, distributed filesystem to handle the storage requirements of guests belonging to different tenants. In the Dike design that we introduce, each client mounts directly the filesystem (instead of indirectly through a proxy server). The filesystem natively manages the access control metadata of each tenant and ensures that each tenant can only access its own namespace. Controlled file sharing is relatively straightforward as a result of the file-based access to a common filesystem with file-granularity access control. We provide prototype implementation of the above approach in the Ceph production-grade, distributed filesystem. We experimentally quantify the limited performance overhead of our design.

Although file-based storage access has been advocated to improve data sharing, manageability and performance in virtualization environments, the access-control management across the principals of different tenants by a cloud provider remains a challenging problem [9]–[11]. We can summarize our contributions as follows: (i) Description of access-control requirements in consolidated storage. (ii) Architectural design of multitenant access control for an object-based storage backend. (iii) Prototype implementation of multitenancy in a production-grade, object-based filesystem. (iv) Experimental performance evaluation of multitenancy scalability and overhead across different systems.

In the remaining document we motivate our work on file-level storage consolidation (§II) and identify several system requirements (§III). Then we describe our system design (§IV) and prototype implementation (§V). We present our experimentation environment and explain some representative experimental results (§VI). We point out previous related research in comparison to our work (§VII). Finally we summarize our conclusions and plans for future research (§VIII).

II. Motivation

Remote storage access through a file-based interface often improves the performance of virtual machines in comparison to block-based access [12]. Nevertheless, a file-based interface is also criticized to compromise the storage robustness to client bugs, and restrict the general benefits of I/O virtualization [13].

The block-based interface isolates a virtual disk into a protection domain under full control by the guest owner [7]. On the other hand, a file-based interface provides the "killer" advantage of configurable guest isolation and sharing support at fine granularity [7]–[9], [14], [15]. The file-based interface also facilitates semantics awareness, which strengthens the consistency and fault isolation of filesystems, and enables several performance optimizations. For instance, the treatment of metadata as data by the block-based interface makes a guest susceptible to corruption from system crashes, unless all writes are treated as synchronous below the guest (e.g., with write-through caching) [12], [13], [16]. Also, filesystem nesting through a block-based interface may lead to performance degradation, e.g., in write-dominated or latency-sensitive workloads [12], [13].

Wide compatibility across different backend storage systems and frontend guest operating systems makes the block-based interface a preferable choice [13]. Instead, several manageability benefits are currently provided by both interfaces. Versioning is supported at coarse granularity in virtual disks, or fine granularity with individual files [7]. Both virtual disks and distributed filesystems can support migration [17], thin provisioning [12], [17], consistent snapshotting [18], and disaster recovery through multi-site replication (e.g., GPFS, GlusterFS, SnapMirror [9], [17]). Finally, a file-based interface makes straightforward the support of content searchability across different guests, e.g., for malware detection [7], [14].

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

Virtual Desktops The private cloud of an enterprise 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.

Shared Workspace The home directories of collaborating users are maintained in a shared filesystem. 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 [1]. The filesystem treats each business customer as a tenant with separate application files in writable mode (e.g., databases), but shared system files in read-only mode (e.g., libraries).

Software Repository A public cloud provides a shared software repository that different groups of developers can fork into separate branches. A group obtains writable access to its own branch, and read-only access to branches of other groups.

III. System Requirements

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

A. Goals and Assumptions

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

1) **Isolation** Each tenant is free to choose identities for its users. Therefore we isolate the identity space and access control of different tenants to prevent 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) **Backwards compatibility** Leverage the architectural characteristics of successful filesystems to ensure backwards 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.

A client provides local service access to a principal (e.g., user or machine) over the network, and a server implements service actions. We assume that networked data storage is accessible at file granularity to a large number of principals from different tenants. For scalability and compatibility reasons, we adopt the architecture of an object-based, distributed filesystem. A collection of object servers (OSDs) securely store the data and metadata in object form. Over the object servers, the metadata servers (MDSs) partition the file namespace, indexes and permissions to achieve locality and load balancing.

B. System Trust and Threat Model

A distributed filesystem protects the confidentiality and integrity of stored data and metadata by restricting remote accesses to authorized principals. The clients and servers of the filesystem run on the nodes of a datacenter that is physically operated and protected by an independent provider. Multiple virtual nodes generally share a physical host. Secure hardware is used to certify the integrity of the system software stack of each node e.g., through a hash chain generated by a Trusted Platform Module [19]. A central monitor builds up the infrastructure trust with remote attestation.

Public keys or their hashes uniquely identify the tenants, principals and services. Certificate is a cryptographically signed statement of authenticity [2]. The private keys of the entities are persistently stored in encrypted form, and only appear in plaintext form at the volatile memory of authorized nodes. The nodes securely communicate over symmetric keys, which are dynamically agreed upon via public-key cryptography or securely exchanged in encrypted form.

The provider has no malicious intent to compromise the system security. However, there may be other reasons (e.g., poor practices) for which the provider is not trusted by some applications. A tenant may externally apply techniques of encryption, hashing and auditing to strengthen end-to-end confidentiality, integrity and freshness [20]. We target filesystem access control without any explicit attempt to provide solutions for public-key distribution, denial of service, traffic analysis, and general multitenant sharing of resources other than storage (e.g., computation).

IV. System Design

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

A. Identity Management

Identity management refers to the representation and recognition of entities as digital identities in a specific domain [21]. Existing systems often handle identities through a centralized or peer-to-peer structure, or by using global-to-local mappings [5], [9], [22]. However, a multitenant environment complicates the secure operation of a shared filesystem due to the multiple independent organizations involved. Thus, we follow a hierarchical management scheme due to the isolation and scalability properties that it offers. Each tenant maintains a private authentication service to locally manage the identities of its principals. The authentication services of legitimate tenants are registered with the filesystem. Requests from an approved tenant can be processed by the filesystem according to the access permissions of each file.

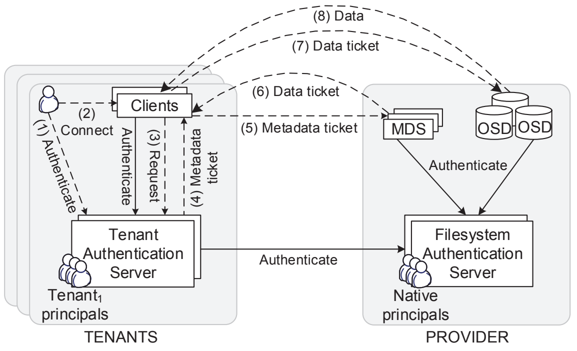


Fig. 1: The Dike multitenant access-control system. In the parentheses over dashed lines we enumerate the steps of file access by a principal. The solid lines refer to the authentication initialization of the system components.

B. Authentication

We partition the task of authentication among the provider and the tenants. Each tenant uses a separate *tenant authentication service (TAS)* to authenticate the local clients and principals. Additionally, the provider operates a *filesystem authentication service (FAS)* to authenticate the metadata servers, data servers and tenant authentication servers of the system (Fig. 1). The principals are distinguished into native filesystem principals, who are trusted to manage the entire filesystem, and tenant principals, who manage or use the tenant resources.

A client is initially authenticated by the TAS (Fig. 1). Each tenant principal is authenticated (step 1) by the TAS, and requests (2) filesystem access through the client. On behalf of the principal, the client contacts (3) the TAS and receives back (4) a *metadata ticket* to request (5) filesystem access from the metadata server (MDS). The MDS uses a FAS-issued certificate of TAS to validate the received metadata ticket. Then, the MDS issues (6) to the client a *data ticket* to access (7-8) a data server (OSD). The data ticket securely specifies the principal and permissions of the authorized operation. The above procedure is similar for accesses by filesystem principals with the main difference that both the clients and principals are directly authenticated by the FAS.

C. Authorization

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

For administration purposes, the filesystem selectively makes the metadata accessible to different entities in the form of views. The filesystem administrator uses the *filesystem view* to specify permissions for entire tenants or individual principals. Instead a tenant administrator uses a *tenant view* to configure the metadata that has been made accessible to the respective tenant by the provider. A principal obtains filtered access to a subset of the filesystem or tenant view according to the applicable permissions.

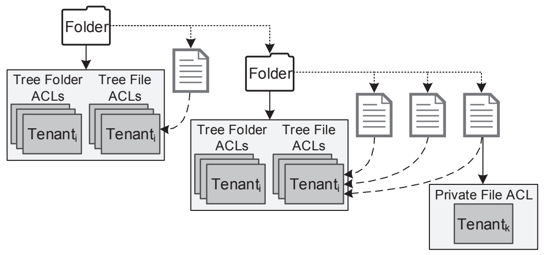


Fig. 2: Inheritance and common permissions in Dike with tree file and folder permissions. We also show private file permissions, which are limited to individual files.

Dike allows an access policy to specify a file as private or shared across the principals of a single or multiple tenants. Subsequently, cross-tenant accesses in Dike are natively supported by the filesystem. In general-purpose public-key cryptography, the certification hierarchy can have an arbitrary number of levels. Instead, Dike only uses a two-level organization to let the TAS of each tenant be certified by the FAS. Dike also differs from the multi-realm Kerberos protocol, in which remote accesses require tickets by the local and remote realm to be granted either directly, or hierarchically through a common ancestor [3].

D. Inheritance and Common Permissions

Cloud storage systems generally handle an enormous number of files. Managing several permissions for each file involves considerable space and time requirements. In Dike, we support 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. From our isolation goal it follows that we enforce the inheritance and common permissions separately within each tenant.

Let 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 principal (similar to the Unix `umask` semantics). Inheritance trivially applies in this case because all files and folders are created with the same default settings, respectively.

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

Finally, we can explicitly set the permissions of an individual file to *private tree permissions*. These are distinct from the tree file permissions inherited from the parent folder. In Fig. 2 we present an example of two folders and multiple files adopting the tree file or private file permissions.

E. Security Analysis

A server securely receives from a client a fresh tamper-proof ticket that specifies the authorized client and principal of an access. According to the file permissions and the ticket-specified identities, a principal is denied unapproved access to the data and metadata of other principals at the same or a different tenant.

It is possible that an attacker penetrates the client of a tenant and impersonates oneself as a legitimate principal. Given that the permissions of different tenants are stored separately, cross-tenant policy violation is prohibited by design. Indeed, the harm of such an attack is limited to the private or shared files that are accessible by the compromised tenant. The attacker cannot modify the system-wide access policy and affect the principals of other tenants or the native principals of the filesystem (except for files shared across them).

In the case that the account of a filesystem administrator is compromised, an attacker may be able to gain complete access to the permissions and data of the system. The implications of such an attack can be limited, if the tenants apply external protection techniques, such as encryption or hashing of the stored data. Special protection measures can reasonably harden external attacks. For instance, the provider may apply secure virtualization at the network level. As another measure, the provider can disable direct access of the filesystem or tenant administrator from outside the datacenter.

V. System Prototype

Next we describe our implementation of the Dike multitenant access control over a distributed filesystem. The prototype implementation is based on Ceph, a flexible platform with scalable management of metadata and extended attributes [18].

A. 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) reliably store objects, and the monitor (MON) manages the server cluster map [18]. A registered client shares a secret key with the MON. When a principal requests a filesystem mount, the respective client receives a session key after being authenticated by the MON. With the session key, the client securely requests the desired Ceph services from MON and receives back an authenticating ticket for the MDSs and OSDs.

The MDS maintains the state of communication with a client in an entry of a session map. Unless a session fails, it 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 for the root directory. From the capability the client derives an object identifier and the placement group of OSDs that contain the object replicas.

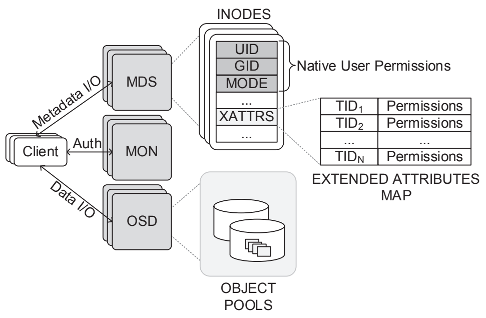


Fig. 3: Prototype implementation of the Dike multitenant access-control system. Our solution is based on the extended attributes of the Ceph filesystem.

B. Support for Multitenant Access Control

We expanded Ceph to natively provide multitenant access control according to the Dike design (Fig. 3). Apart from the added multitenancy support, our current prototype implementation relies on the authentication and authorization functionality of 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 (e.g., RIPEMD-160) to the public key of the tenant. The client embeds the TID into an expanded MClientSession request, and sends it to the MDS over the secure channel based on the session key. The MDS extracts the TID from the received message, and stores it in the established session. The secure session between the filesystem and an authenticated client can only serve the actions permitted to the principals of the identified tenant.

Permissions Our current implementation only supports Unix-like permissions for users and groups, but it makes straightforward to add access-control lists in a future version. Based on the supplied TID, a client obtains tenant view of the filesystem for access by a principal of the tenant. The permissions of the tenant view are stored in the extended attributes of the filesystem. For global configuration settings, we also support the filesystem 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 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 principal can explicitly set the access permissions of a particular file by creating a new entry of permissions for the respective tenant.

A file capability is only sent to a client whose tenant is permitted access to 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.

Development We developed the Dike prototype by adding 2023 commented C++ lines into Ceph v0.61.4 (Cuttlefish). The new functionality was implemented in the client and metadata server of the Ceph filesystem. We extended the CInode class of Ceph with eight new operations to set and retrieve the permissions of tenants and individual principals. We also modified all the Ceph filesystem functions related to permissions handling, including the inode constructor.

VI. Performance Evaluation

We experimentally examine the scalability of Dike along several parameters with respect to Ceph and other systems.

A. Experimentation Environment

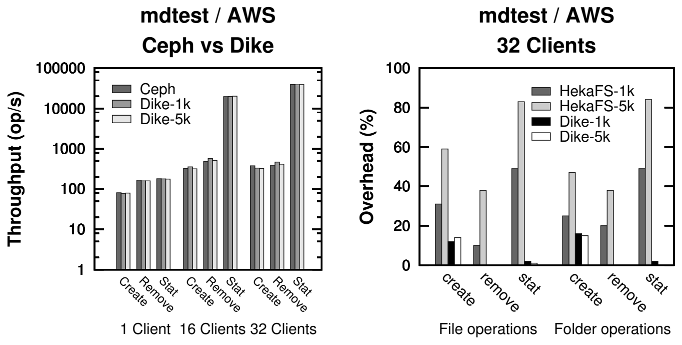
Our testbed consists of EC2 instances from the US East region of the Amazon Web Services (AWS). We use up to 3 instances of type "m1.large" (4x64bit cores, 15GB RAM) as fileservers and 32 instances of type "t1.micro" (1x64bit core, 615MB RAM) as microbenchmark clients. All instances run Red Hat Enterprise Linux Server r6.4 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 §VI-B). 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 run the MDS and MON on two of the fileservers.

We measure the metadata performance using the mdtest v1.9.1 from LLNL. This is an MPI-based microbenchmark running over a parallel filesystem. Each spawned MPI task iteratively creates, stats and removes a number of files/folders. We found 12 processes per client to maximize the throughput in the local cluster, and 5 processes to maximize the throughput of an AWS client node. We repeated the experiments as needed to constrain the 95% confidence-interval half-length of throughput within 5% of the average value.

B. Experimentation Results

In Fig 4a we run up to 32 mdtest clients in the AWS testbed. 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 1k or 5k private folders. We enable the access permissions of each folder for a single tenant, which respectively leads to 1,000 (Dike-1k) and 5,000 distinct tenants (Dike-5k). An increasing number of clients tends to improve the total throughput of both systems (except for a slight drop in remove from 16 to 32 client). At 1 client, Ceph serves the file create at 81op/s, Dike-1k at 78op/s and Dike-5k at 80op/s. The overhead of Dike is similarly limited in remove and stat. Even at 32 clients, Dike-1k only reduces the throughput of Ceph by 0-12%, while increase of the tenants to 5k adds 2% extra overhead.

GlusterFS is an open-source, distributed filesystem from RedHat. It supports translator layers for the addition or removal of features. HekaFS is a cloud filesystem implemented



(a) Tenant Scalability (b) Multitenancy Overhead

Fig. 4: (a) We use mdtest to compare the scalability of Dike and Ceph across different numbers of tenants. (b) We compare the overhead of multitenancy support in Dike over Ceph to that of HekaFS over GlusterFS using private folders.

as a set of translators over GlusterFS. In order to isolate the identity space of different tenants, HekaFS uses a map file to translate the local user identities of the tenants to globally-unique identities [9]. However, identity mapping was previously criticized as cause for limited scalability [22]. Also, 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 [9].

In Fig. 4b we measure the throughput decrease (denoted as overhead) in file and folder operations incurred by Dike over Ceph and HekaFS over GlusterFS. In AWS, we use 32 clients to run mdtest in a filesystem configured with 1,000 or 5,000 tenants and private folders per tenant. The overhead of file and folder create operations approaches 12-16% in Dike-1k, and 14-15% in Dike-5k. On the contrary, the overhead of HekaFS over GlusterFS in file and folder stat operations approaches 49% with 1k tenants, and 84% with 5k tenants. In stat, the overhead of Dike (0-2%) remains nearly up to two orders of magnitude below that of HekaFS (49-84%).

In summary, Dike adds multitenancy support over Ceph at limited performance overhead up to 16%, unlike HekaFS that incurs overhead over GlusterFS up to 84% in our experiments.

VII. Related Work

All the principals of a distributed system are often registered into a central directory (e.g., Kerberos [3]). Secure data transfer between clients and storage through an object-based interface is addressed in the Network-Attached Secure Disk model [4]. Plutus applies cryptographic storage to support secure file sharing over an untrusted server [23]. 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 [6]. However, the above research does not directly address the multitenancy of consolidated cloud storage.

Multitenancy isolation without sharing support was offered by running separate filesystem virtual machines per tenant, or shielding the filesystem processes of different tenants [24].

File-based storage virtualization is enabled by Ventana through versioning and access-control lists (ACLs), but without tenant isolation [7]. VirtFS uses a network protocol to connect a host-based fileserver to multiple local guests without isolating their respective principals [8]. 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 [11]. 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.

Secure Logical Isolation for Multi-tenancy (SLIM) has recently 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 [10]. The data-protection-as-a-service architecture introduces the secure data capsule as an encrypted data unit packaged with security policy, and it confines applications within mutually-isolated secure execution environments [15]. We also target secure storage in the datacenter, but with multitenancy support at the access-control metadata of object-based fileservers.

VIII. Conclusions

We consider the security requirements of scalable filesystems used by virtualization environments. Then we introduce the Dike system design 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 up to 16% in configurations with several thousand tenants. Our plans for future work include integration of Dike into a trusted virtualization platform in the datacenter, further experimentation with I/O-intensive applications at large scale over different object-based filesystems, and consideration of weaker trust assumptions.

Acknowledgment

This research was supported in part by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund (Project "Cloud9"). Credit provided by AWS in Education Grant award to access the Amazon Web Services is gratefully acknowledged.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] A. W. Leung, E. L. Miller, and S. Jones, "Scalable Security for Petascale Parallel File Systems," in *ACM/IEEE Intl Conf High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2007, pp. 16:1–16:12.

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

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

[9] J. Darcy, "Building a cloud file system," *USENIX; login:*, vol. 36, no. 3, pp. 14–21, Jun. 2011, (see also http://hekafs.org).

[10] 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.

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

[12] D. Le, H. Huang, and H. Wang, "Understanding performance implications of nested file systems in a virtualized environment," in *USENIX Conf. File and Storage Technologies*, San Jose, CA, Feb. 2012, pp. 87–100.

[13] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok, "Virtual machine workloads: The case for new benchmarks for NAS," in *USENIX Conf. File and Storage Technologies*, San Jose, CA, Feb. 2013, pp. 307–320.

[14] 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.

[15] 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.

[16] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *USENIX Conf. File and Storage Technologies*, San Jose, CA, Feb. 2013, pp. 45–58.

[17] J. K. Edwards, D. Ellard, C. Everhart, R. Fair, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, K. A. Smith, and E. Zayas, "FlexVol: flexible, efficient file volume virtualization in WAFL," in *USENIX Annual Technical Conf.*, Boston, MA, Jun. 2008, pp. 129–142.

[18] 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. Operating Systems Design Implementation*, Seattle, WA, Nov. 2006, pp. 307–320.

[19] 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.

[20] A. Juels and A. Oprea, "New Approaches to Security and Availability for Cloud Data," *Comm. ACM*, vol. 56, no. 2, pp. 64–73, Feb. 2013.

[21] A. Jøsan, M. A. Zomai, and S. Suriadi, "Usability and privacy in identity management architectures," in *Australasian Inf. Sec. Workshop: Privacy Enhancing Technologies*, Ballarat, Australia, Jan. 2007, pp. 143–152.

[22] 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.

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

[24] 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.