Dike: Virtualization-aware Access Control for Multitenant Filesystems

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

Technical Report DCS2013-1
February 18th, 2013

ABSTRACT

In a virtualization environment that serves multiple customers (or tenants), storage consolidation at the filesystem level is desirable because it enables data sharing, administration efficiency, and performance optimization. Today the scalable deployment of filesystems in such environments is challenging due to intermediate translation layers required for purposes of networked file access or identity management. First we analyze the security requirements in multitenant filesystems. Then we introduce the Dike authorization architecture, which combines native access control with tenant namespace isolation that is backwards compatible to object-based filesystems. We experimentally evaluate a prototype implementation that we developed, and show that our solution incurs limited added performance overhead.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*; D.4.8 [**Operating Systems**]: Performance—*measurements*

General Terms

security, design, experimentation, performance, measurements

Keywords

access-control lists, object-based storage, scalability

1. 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 required isolation of the security administration across different organizations. Distributed authorization has already been extensively studied in the context of networked services, e.g., distributed filesystems [10]. However, a cloud environment introduces unique characteristics that warrant reconsideration of the assumptions and solution properties.

We are particularly interested to take advantage of service co-location in the datacenter to better consolidate the storage infrastructure used by common data files at the application (e.g., collaboration documents) or system level (e.g., root images). Secure storage consolidation at the filesystem level is increasingly advocated as the preferred multitenancy paradigm for cloud environments [13, 8, 4, 2, 17]. 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 guest tenants, rely on global-to-local identity mapping to manage the users of different tenants, or have the guests and a centralized filesystem (or proxy) running at the same host [13, 4, 2]. In the proposed solution we rely on an object-based, distributed filesystem to handle the storage requirements of clients (e.g., virtual machines) belonging to different tenants.

In our design we require that each client directly mounts the filesystem instead of having the filesystem mounted by an intermediate proxy. 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. With microbenchmarks and application-level experiments we quantitatively demonstrate the limited performance overhead of our design.

We can summarize our contributions as follows:

- Analysis of access-control requirements in consolidated storage for virtualization.
- Architectural design of native access control in a multitenant filesystem with backwards compatibility to object-based storage.
- Prototype implementation over a production-grade distributed filesystem.
- Experimental performance evaluation of multitenancy overheads.

In the remaining document we provide motivating scenarios of file-level storage consolidation (§ 2), introduce basic related concepts (§ 3), and present our trust and threat model (§ 4). We describe our system design and prototype in § 5 and § 6, respectively. Then we explain our experimental measurements (§ 7), and compare our work with previous research (§ 8). Finally we summarize our conclusions and plans for future work (§ 9).

2. MOTIVATION

Next we examine scenarios 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. As an optimization, there is a shared, read-only directory that is branched into the private directory of each client. A similar approach can also be applied to manage the home directories of users.

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 possibly shared system files in read-only mode (e.g., configuration scripts).

Software Repository: A public cloud provides a shared software repository that different groups of developers can fork into separate branches. The members of a group obtain writable access to their own branch, and read-only access to the branches of other groups. A simpler scheme without branches could also be used for sharing scientific datasets.

3. BACKGROUND

In a distributed filesystem, client is typically a process that provides local filesystem access to a node, while the servers implement filesystem actions across the network [10]. Principal is the user or process that accesses the filesystem through the client. In a traditional distributed filesystem, all principals are registered to a central directory service. If a principal is securely identified by the directory, it receives a ticket (i.e., capability) to contact the filesystem. For a filesystem request that does not violate the access policy, the principal is granted the ticket that provides the authorized service from a server. Object-based, distributed filesystems also rely on a similar authorization approach, but emphasize the scalability of secure data and metadata management [19].

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 or indirectly as files of network-attached storage mounted by the host. File-based access of consolidated storage has been advocated to improve data sharing, manageability and performance. Nevertheless, traditional file-based access presumes that users are registered into a central authentication service. Due to identity management challenges from the enormous number of users involved, this is most likely unrealistic for the tenants of a cloud provider.

File-based storage virtualization can be managed through versioning and access-control lists (ACLs) (Ventana, [13]). Server file ACLs have system-wide effect to all clients; the guests apply their own format and rules to specify the guest file ACLs of their users. A shared NFS server allows guests at a host to access networked object servers through a custom protocol. Alternatively, a network-filesystem protocol can connect a host-based fileserver to multiple co-located guests (VirtFS, [4]). The mapped security model stores the guest credentials as extended attributes at the fileserver, and the passthrough security model stores the guest credentials directly on the fileserver. Hierarchical delegation enables a tenant to assign identities to local principals; the subdirectories from different volumes are combined to a unified tenant view (HekaFS, [2]). Different tenants can coexist by translating the pair of tenant and principal identifier to a unique per-server principal identifier.

The scalability of Ventana and VirtFS is limited by the centralized NFS-like functionality at the host. VirtFS mainly targets a fileserver co-located on the host of the guests without isolating the principals of different guests. HekaFS lacks support for sharing and applies global-to-local identity mapping that was previously criticized as cause for limited scalability in grid computing [7]. In the present work, we aim to natively support multitenancy by directly storing access-control metadata at the fileserver without the need for identity translations from one tenant to another.

4. TRUST AND THREAT MODEL

The clients and servers of the filesystem all run in one datacenter that is physically protected and operated by an independent provider. A secure co-processor certifies the software stack on each physical host (e.g., hash chain generated by a Trusted Platform Module [12]). A central monitor establishes the trust of the infrastructure from the integrity of the participating nodes. Public keys (or hashes thereof) uniquely identify tenants, principals and services. The nodes securely communicate over temporary symmetric keys dynamically agreed upon via public-key cryptography. The private keys of principals and services are permanently stored in encrypted form and only appear in cleartext form at the volatile memory of authorized nodes. Before the reallocation of host memory across different nodes, the memory contents are scrubbed to prevent information leakage.

The filesystem protects the confidentiality and integrity of stored data and metadata by restricting accesses to authorized principals. We assume that 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 for some applications. In that case the tenant may externally apply techniques of encryption, hashing and auditing to achieve end-to-end confidentiality, integrity and freshness [3]. We target filesystem access control without any explicit attempt to provide solutions for public-key distribution, entity authentication, denial of service and traffic analysis. Finally, we do not address general distributed processing, which involves multitenant sharing of resources other than storage (e.g., computation).

5. SYSTEM DESIGN

In the present section we introduce the Dike system of access control for multitenant shared storage at the file level.

5.1 Assumptions and Goals

The storage system follows the architecture of an object-based, distributed filesystem (Fig. 1). A collection of data servers are responsible to redundantly store the data and metadata in object form. Multiple metadata servers achieve locality and load balancing by partitioning over the object servers the name, data index and ACL of different files. The system can flexibly manage the secure access to stored objects with help from the operating system at each object

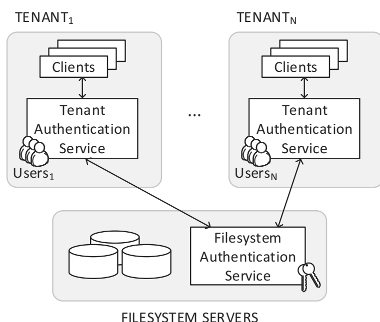


Figure 1: The Dike multitenant access-control system. A tenant authenticates its own principals, and the filesystem authenticates the authentication service of each tenant.

server. In the proposed scheme of filesystem access control, we set the following goals:

1. **Isolation:** Securely isolate the identity space and access control of principals from different tenants.
2. **Sharing:** Enable flexible file sharing among the principals of the same and different tenants.
3. **Efficiency:** Provide native support of multitenant access control for filesystem performance and scalability.
4. **Compatibility:** Ensure architectural compatibility with existing scalable and reliable filesystems.

5.2 Authentication

A trusted monitor at the datacenter certifies the integrity of the software stack running at each client. The client receives the secret key to decrypt its private key and uses public-key cryptography to securely connect with a tenant. Every tenant certifies the identity of local clients and principals with its own authentication service that is securely registered to the filesystem authentication service (Fig. 1). A principal connects to a particular client and provides a secret password for authentication by a tenant that also stores the password in encrypted form. After the successful authentication, the principal receives a secret key to decrypt its private key that is made accessible at the client. From the tenant the principal retrieves a secure ticket to access the metadata server of the filesystem, and from the metadata server retrieves another secure ticket to access a particular data server. Message freshness is ensured with a requester-provided nonce that the replying party returns modified according to a known function (e.g., increment by one).

5.3 Authorization

The filesystem grants to a principal a permitted file access according to the tenant-issued ticket. The authorization policy is specified in ACLs maintained by the filesystem. The rules of principals that belong to different tenants and the filesystem are respectively maintained across separate ACLs. The ACL of a tenant for a particular file is a list of entries; each entry consists of a principal's identity and a representation of the permitted actions. There is a separate ACL, where the filesystem maintains the permissions of its native principals. A file can be configured as private or shared across the principals of a single or multiple tenants.

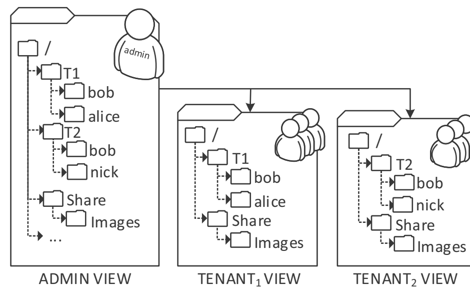


Figure 2: Admin and tenant view of the filesystem metadata in the Dike system.

For administration purposes the system provides selective access to metadata in the form of views (Fig. 2). The filesystem administrator has access to the *admin view*, which allows specification of permissions at the granularity of entire tenants or individual principals. Instead the *tenant view* allows a tenant administrator to configure the metadata made accessible to the tenant by the filesystem admin. Depending on whether it belongs to the filesystem or a tenant, respectively, a principal can only access a subset of the admin or tenant view filtered according to the applicable permissions.

5.4 Security Analysis

The specification of the authorized client and principal in an encrypted ticket along with the secure ticket exchange among nodes prevent a principal from getting unapproved access to the data and metadata of other principals from the same or different tenant. An attacker may manage to penetrate a client and guess the password of a regular principal. Then the attacker is still unable to modify the system-wide access policy, which affects the native principals of the filesystem or the principals of other tenants. Special protection measures make harder to forge the identity of the filesystem administrator, e.g., by disabling access to the respective account from outside the datacenter.

6. 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 prototyping platform with scalable management of metadata and extended attributes.

6.1 Outline of Ceph

Ceph consists of four components: 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. Both data and metadata are stored on OSDs, but they are separately managed for greater scalability. The metadata is dynamically partitioned across the MDSs to preserve locality and achieve load balancing. A registered client shares a secret key with the monitor. When a user requests from the client to mount a filesystem, the client is authenticated by the monitor and receives a session key encrypted with the secret key. The session key is used by the client to securely request from the monitor a ticket that authenticates the client to the MDSs and OSDs.

The ticket is encrypted with a secret key that the monitor

Method	*Description*
`bool check_tenant_perm()`	Check tenant permission
`void grant_tenant_perm()`	Grant tenant permission
`void set_unix_uid()`	Set user ID
`void set_unix_gid()`	Set group ID
`uid_t set_unix_mode()`	Set file permissions
`uid_t get_unix_gid()`	Return user ID
`gid_t get_unix_gid()`	Return group ID
`mode_t get_unix_mode()`	Return file permissions

Table 1: The methods that we added into class `CInode` to manage the tenant permissions of an inode.

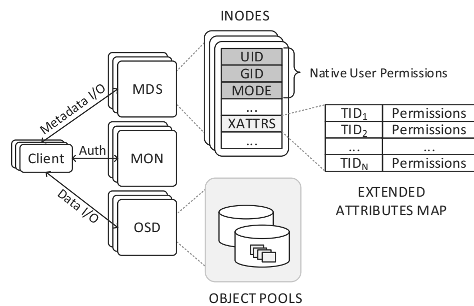


Figure 3: Prototype implementation of the Dike multitenant access-control system.

shares with the MDSs and OSDs. The client uses the ticket to initiate a new session with the MDS. The MDS receives from the client a message of type `MClientSession` and sends back the capability (i.e., ticket) that enables access of the root directory at the OSDs. In general the returned capability contains the inode number, the permitted operations, the replication factor and the striping method of the file. From the capability the client derives an object identifier, which is hashed to the placement group of OSDs that contain the object replicas. A Ceph directory is stored as a single object, or as a collection of fragments with each fragment on a different object. A directory entry includes the name, the inode and the extended attributes of a file. Every MDS maintains a journal of recently-updated metadata. Metadata updates are labeled as *projected* while written to the journal but not yet to the in-memory cache, *committing* while queued to the disk, and *committed* when written in stable storage.

6.2 Multitenant Access Control

In Ceph we implemented native support for multitenant access control according to the Dike design (Fig. 3). We deliberately avoid global-to-local identity translations because they introduce performance bottlenecks, replica inconsistencies and impersonation vulnerabilities. Instead we enforce the access policy by restricting the session between a client and the filesystem to only serve the permitted actions of the principal who initiated the session. In a filesystem mount request to an MDS, a client has to securely identify the tenant of the principal. In order to derive the tenant identifier (TID) we apply the RIPEMD-160 cryptographic hash function on the public key of the tenant. Then we include the TID into an expanded MClientSession request and send it to the MDS over a secure session. For authentication purposes the request should additionally carry a tenant-issued certification (not supported yet in our prototype).

The MDS extracts the TID from the MClientSession message and stores it as a field of the session class. Our current implementation only supports Unix-like permissions of individual users and groups, but it is straightforward to add access-control lists in a future version of our code. We facilitate the system administration through the support of multiple filesystem views. Based on the supplied TID, a client obtains tenant view of the filesystem for access by a principal of the tenant. For configuration purposes, we also provide the admin view that enables full access permissions to the filesystem. We extended the *CInode* class of Ceph with eight new operations to set and retrieve the permissions of tenants and individual principals (Table 1).

When the tenant view is used, the permission attributes are stored in the extended attributes of the filesystem; otherwise the regular fields of the inode are accessed. The extended attributes are managed as key-value pairs stored in a C++ map structure (red-black tree). We use as key the string `"TID`‖`perm_type"`, where TID is the tenant identifier and `perm_type` is set to `"UNIX"` for Unix permissions or `"ACL"` for the ACL model. In the Unix model the value of the pair can be set to `"UID:GID:mode"`, where UID and GID are the user and group ID, respectively, while mode represents the Unix file permissions.

We modified all the filesystem functions related to permissions handling, including the constructor of a new inode. If the client uses the admin 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 under TID; we also update the regular inode of the filesystem according to the user/group IDs and file mode of the parent inode. A capability is only sent to a client whose tenant has access to the file. In order to allow or deny a file access to a client, we modified the returned capability to include the tenant identifier and the respective file ownership metadata. In general a client cannot directly access the extended attributes that contain access control information because it is responsibility of the filesystem to read and update extended attributes on behalf of authorized client requests.

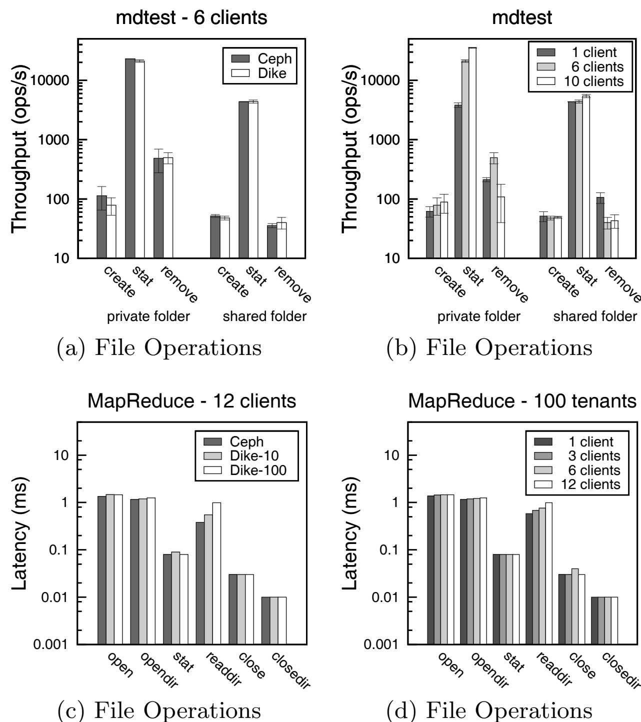
7. PERFORMANCE EVALUATION

7.1 Experimentation Environment

We mainly use a cluster of x86 servers running Linux kernel v3.5.5 (Debian v6.0 amd64 squeeze). Each server is equipped with one quad-core 64-bit Intel Xeon processor at 2.33GHz, 2-4GB RAM, two SATA 250GB 7.2KRPM HDs, and one activated gigabit link. We used up to four servers as client hosts, each running the Xen v4.2.0 hypervisor (1 core/1GB) and three virtual machines (1 core/1GB each). We developed the Dike prototype over Ceph v0.48.2 (Argonaut). We have Ceph (or Dike) installed on hosts with 2GB RAM. One host is shared by the MDS and monitor, while two other hosts are used as OSDs running the Btrfs filesystem with 1GB journal.

7.2 Experimentation Results

Microbenchmark. First we measure the system performance with the mdtest v1.8.3 from LLNL. This is a microbenchmark running in the MPI environment over a par-



(a) File Operations (b) File Operations

(c) File Operations (d) File Operations

Figure 4: Performance comparison of Ceph and Dike with mdtest and MapReduce across different numbers of clients (Ceph and Dike) and tenants (Dike only).

allel filesystem. Each spawned MPI task iteratively creates, stats and removes a specified number of files/directories. We configure every client to run 5 tasks in 3 iterations. A total number of 12000 created files are equally divided among the tasks of the experiment. Dike is configured to support 10 tenants and has each client accessing the filesystem through a dedicated tenant. We examine the cases that either every client creates files in a private folder of the filesystem, or all clients use a shared folder.

We show the average execution time of the experiment along with the standard deviation as an error bar. In Fig. 4a we compare the throughput of mdtest running on 6 clients. The measured performance is comparable between Ceph and Dike. A notable case is file create over a private folder, where Dike with 78.6ops/s lies 31% lower than Ceph with 113.7ops/s. In Fig. 4b we examine the measured throughput as a function of the number of clients. We notice that increasing the number of clients from 1 to 10 leads to higher throughput for create and stat by about a factor of 10, but lower throughput for remove beyond 6 clients. This behavior is reasonable given the different intensity of contention caused by shared (e.g., stat) or exclusive (e.g., remove) locking involved in the operations, respectively.

MapReduce. We gathered application-level measurements with Stanford's Phoenix v2 shared-memory implementation of Google's MapReduce. Our MapReduce application is called reverse index: it receives a collection of HTML files and generates the text index with links to the files. Our dataset contains 40,376 files and occupies 530.8MB. We measure the latency of several metadata operations during the index building. We repeated the experiment to constrain the 95% confidence-interval half-length within 5% of the average stat latency. We consider three different configurations: the original Ceph, the Dike with 10 tenants (Dike-10), and the Dike with 100 tenants (Dike-100). Due to multitenancy, there is a corresponding higher volume of stored ACL metadata in Dike. Nevertheless most operations of Dike are completed in latency comparable to that of the original Ceph (Fig. 4c). One exception is readdir whose latency increases by a factor of 2 between the original Ceph and Dike-100. Similarly, as we increase the number of clients from 1 to 12 in Dike-100 (Fig. 4d), the latency of readdir grows from 0.58ms to 0.99ms. This latency increase is relatively limited because the folder metadata returned by readdir is cached by the client.

Linux Kernel Build Time		
#Clients	Ceph (s)	Dike (s)
1	4,677.74	4,716.51
4	4,701.09	4,750.88

Table 2: Comparison of the kernel build time from multiple clients across the Ceph and Dike system.

Linux Build. Over a different cluster (2.66GHz quad-core processor, 2x300GB 15KPRM SAS HDs/host) we store the source of the Linux kernel (v3.5.5) in a shared folder of the filesystem with 3 OSDs. Then we use soft links to make the code accessible to private directories of the tenants. We measure the average time to build the system image by up to four clients with dedicated tenant per client (for Dike). The extra latency of Dike is 5% with one client and 7% with four clients (Table 2). Overall Dike incurs a limited performance overhead across the cases that we experimented with.

8. RELATED WORK

Secure multitenancy in cloud storage supports multiple customers at low cost [5]. The virtualization-based multitenancy architecture (VMT) runs separate virtual machines for each customer over a distributed filesystem. Instead the OS-based multitenancy architecture (OSMT) relies on the fileserver kernel to isolate the resources of different customers leading to lower execution overhead. SilverLine introduces data isolation over a cloud hosting infrastructure [11]. It uses labels to control information flow between files and processes within a single machine or across the network. The S4 framework extends Amazon's S3 cloud storage to provide data sharing across different web services [17]. It supports access delegation over the objects of different users via hierarchical, filtered views of the applicable policy.

Virtual directories persistently represent complex searches of files [8]. The combination of a file-based interface with virtual directories facilitates accesses to multiple filesystems, information finding, file sharing and system administration in a virtualization environment. Support of storage access from different institutions requires consistent ownership and permission data across multiple client mounts [18]. Over Lustre the client user and group identifiers can be mapped to an authoritative list of filesystem identifiers. The Maat protocol provides scalable security for petascale distributed filesystems [6]. The extended capability authorizes I/O for any number of users and files, is cryptographically secure, and maintains fixed size through Merkle hash trees.

Excalibur uses a trusted computing abstraction (policy-sealed data) to seal and unseal data according to a specified node policy [15]. CloudProof allows customers of cloud storage to securely detect and prove violations of integrity, write-serializability and freshness [14]. Assuming that the cloud is entirely untrusted, access control over read and write requests is enforced through data encryption with secret keys, and update verification with public-key signatures.

The use of access control was investigated across different communication methods [16]. Groups have been suggested as an effective structure to emulate several complicated means of access control. Access control has been comprehensively examined across known distributed filesystems [10]. ACLs have been criticized for their inability to support non-local users, and limited scalability with user bases crossing organization boundaries. Direct authorization through trust management certificates has been suggested to better meet the requirements for autonomous delegation across organization boundaries [9]. In prior research a method was proposed for hierarchical access control in federated file services across different administrative domains [7]. In the present work, we study the problem of storage multitenancy over virtualization environments, which introduces new challenges as a result of the system consolidation involved in the same datacenter.

9. CONCLUSIONS

We analyze 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 we experimentally demonstrate a limited performance overhead for up to a hundred tenants. Our plans for future work include integration of Dike into a virtualization platform that supports trusted computing in the datacenter, and further experimentation with interesting I/O-intensive applications at large scale.

10. ACKNOWLEDGEMENTS

This research has been co-financed 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.

11. REFERENCES

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

[2] J. Darcy. Building a cloud file system. *USENIX ;login:*, 36(3):14–21, June 2011.

[3] A. Juels and A. Oprea. New Approaches to Security and Availability for Cloud Data. *C. ACM*, 56(2):64–73, Feb. 2013.

[4] V. Jujjuri, E. V. Hensbergen, and A. Liguori. VirtFS: Virtualization aware File System pass-through. In *Ottawa Linux Symposium*, 2010.

[5] 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.*, pages 460–479, Lisboa, Portugal, Dec. 2011.

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

[7] G. Margaritis, A. Hatzieleftheriou, and S. V. Anastasiadis. Nepheli: Scalable Access Control for Federated File Services. *J. Grid Computing*, May 2012.

[8] D. T. Meyer, J. Wires, N. C. Hutchinson, and A. Warfield. Namespace Management in Virtual Desktops. *usenix; login:*, 36(1):6–11, Feb. 2011.

[9] 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 Conference, Freenix Track*, pages 168–178, San Antonio, TX, June 2003.

[10] S. Miltchev, J. M. Smith, V. Prevelakis, A. Keromytis, and S. Ioannidis. Decentralized access control in distributed file systems. *ACM Computing Surveys*, 40(3):10:1–10:30, Aug. 2008.

[11] Y. Mundada, A. Ramachandran, and N. Feamster. SilverLine: data and network isolation for cloud services. In *USENIX HotCloud Workshop*, Portland, OR, June 2011.

[12] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers. In *IEEE Symposium on Security and Privacy*, pages 414–429, Oakland, CA, May 2010.

[13] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *USENIX Networked-Systems Design and Implementation*, pages 353–366, San Jose, CA, 2006.

[14] 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 Conference*, pages 355–368, Portland, OR, June 2011.

[15] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *USENIX Security Symposium*, pages 175–188, Bellevue, WA, Aug. 2012.

[16] D. K. Smetters and N. Good. How users use access control. In *Symposium on Usable Privacy and Security*, Mountain View, CA, July 2009.

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

[18] 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 Conference*, Pittsburgh, PA, Aug. 2010.

[19] 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 and Implementation*, pages 307–320, Seattle, WA, Nov. 2006.