SECURE MANAGEMENT OF NETWORKED STORAGE SERVICES: MODELS AND TECHNIQUES

A Thesis
Presented to
The Academic Faculty

by

Aameek Singh

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August, 2007

SECURE MANAGEMENT OF NETWORKED STORAGE SERVICES: MODELS AND TECHNIQUES

Approved by:

Dr. Ling Liu, Advisor, Committee Chair
College of Computing
Georgia Institute of Technology

Dr. Douglas M. Blough
Electrical and Computer Engineering,
and College of Computing
Georgia Institute of Technology

Dr. Karl Aberer
Computer and Communication Science
EPFL Lausanne, Switzerland

Dr. Calton Pu
College of Computing
Georgia Institute of Technology

Dr. Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Dr. Kaladhar Voruganti
Advanced Development Group
Network Appliance, Inc.

Date Approved: April 12, 2007

To

Amrik Aman

ACKNOWLEDGEMENTS

This dissertation is a culmination of nearly five years of research. During this time, I have been helped, motivated and inspired by many individuals both professionally and personally. I would like to take this opportunity to express my sincere gratitude to all of them.

First and foremost, I would like to thank my advisor Prof. Ling Liu for her constant support and encouragement. Her astute judgment and understanding has been pivotal in shaping this dissertation. She has always given me the freedom to pursue my (diverse) research interests and nudged me along with critical feedback when it was most needed. Her knack for understanding situations from my perspective has not only helped in my research but also assisted me immensely in making the right career choices. For her consistent sound advice, I thank her from the bottom of my heart. Also my wife joins me in thanking Prof. Liu for her generous accommodation of my schedule and allowing me to often work from home during the last two years.

I would also like to thank my PhD committee members Prof. Calton Pu, Prof. Mustaque Ahamad, Prof. Douglas Blough, Prof. Karl Aberer and Dr. Kaladhar Voruganti for their insightful comments regarding the dissertation research. Their suggestions have improved the quality of the thesis and I thank them for their help. I would specially like to thank Prof. Pu and Prof. Ahamad for being sounding boards for my ideas throughout my PhD and helping me identify important problems.

My acknowledgements would be incomplete without recognizing the contributions of the Storage Systems group at IBM Almaden Research Center. My three internships there have been amongst the most memorable times of my PhD. For providing a great research environment and friendly atmosphere, I thank my manager, Mr. Sandeep Gopisetty and my mentor Dr. Kaladhar Voruganti. They have been great friends, mentors and motivators. I would also like to thank Dr. Sandeep Uttamchandani, Dr. Madhukar Koropolu and Dr. Arup Acharya for being wonderful colleagues. I also thank IBM for supporting me during

the last two years with the IBM PhD Fellowship in 2005-06 and 2006-07.

The DISL research group at Georgia Tech has been a great venue for discussing half-baked ideas and I thank every current and former member for being part of it. I also thank the excellent support staff, especially Ms. Susie McClain and Ms. Barbara Binder for their always prompt administrative help.

The rigor and hard work of my PhD has been balanced by the fun and joy of being in company of good friends from the erstwhile "*Tumlin gang*" – Tejas Iyer, Ashok Ponnuswami, Apurva Mudgal, Barath Petit, and many other friends – Aranyak Mehta, Parikshit Gopalan, Nayantara Bhatnagar and Bugra Gedik, to name a select few.

Of course, all this would have neither been possible nor worthwhile, but for my family. It is impossible to put into words my feelings of love and gratitude for my mother Mrs. Surinder Kaur, my father Late S. Amrik Singh, my brothers Ramneek and Parteek and my *bhabis* Preeti and Silvi. They have strived for me when I wouldn't, they have dreamt for me when I couldn't and their blessings are the reason who I am and where I am. Also, a bear hug for my most ardent fan club – Harpragaas, Elahi and Ruhani, the cutest nephew and nieces ever. I am deeply grateful to the (real) doctors in Chandigarh, Dr. Deepak Bhasin, Dr. Vidhu Bhasin, Bhavna and Dinkar for their always uplifting encouragement and loving support through this journey, especially to my final year tea and maggi partner Bhavna. My heartfelt gratitude also goes out to the entire Freedom family for their unconditional support of everything I pursued.

Finally, to the most pleasant distraction in my life, Deepshikha Bhasin, if it weren't for you, I might have finished a year earlier but if it weren't for you, I certainly would not have finished at all. I thank you for your love, affection, inspiration and the countless hours of cooking.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In the increasingly digital world, enterprises are generating more data than ever before. Additionally, data is being stored for longer periods of time for both business intelligence and regulatory compliance. The enterprise storage infrastructures are tasked with not only storing this large amount of data, but also having to deal with increasing reliance on anytime and anywhere access to data. As a result, storage management has become increasingly complex and expensive.

For cost-effective data storage and management, enterprises are beginning to look at the new paradigm of *Networked Storage Services*, also referred to as *Storage-as-a-Service*. In this model, enterprises store their data at a remote site managed by an external storage service provider (SSP) and access it over a high speed network. The SSP not only manages the storage systems, but also provides superior backup and disaster recovery solutions. It also enhances data availability and content dissemination abilities of the enterprise.

The networked storage services model, however, faces unique challenges before its widespread acceptance. The foremost challenge in this model is that of preserving data confidentiality and enforcing access control over SSP-stored data. By *outsourcing* storage to an external service provider, enterprises have to adopt new security mechanisms for their multiuser environments, without placing an inordinate amount of trust in the SSP. Such access control and security mechanisms have to be efficient in performance and able to be easily integrated into existing enterprise infrastructures.

The second important challenge in this model is that of efficient storage management of SSP data centers. The SSPs not only have to manage large amounts of data in a cost-effective manner, but they also have to provide an on-demand infrastructure in which client enterprises can easily grow or shrink their storage requirements. Maintaining such large dynamic storage environments is a complex challenge and new technologies are required for efficient change management and resource allocation.

This dissertation addresses many of the challenges described above. First, we have developed security mechanisms that efficiently provide access control for the outsourced storage services model. Our primary contribution in this domain is the development of novel techniques for *embedding* access control into data. Instead of relying on a trusted reference monitor for access control, we use efficient cryptographic techniques to *embed* access control into the stored data itself and thus, no enforcement is trusted with the storage service provider. Using this technology, we have made the following contributions:

- *Filesystem Access Control*: We introduce a new access control system, called xACCESS, that *embeds* expressive UNIX-like access control semantics into SSP-stored data. Using novel cryptographic access control primitives and completely in-band key management, xACCESS can be easily integrated into existing enterprise environments. We also analyze the privacy characteristics of its data sharing mechanisms and propose enhancements that allow users to share data more securely and conveniently.

- *Secure Filesystem Search*: We also use the access control embedding concept to provide secure keyword search over SSP-stored data. Multiuser search services need to enforce access control even during search and many current enterprise search approaches are prone to *inference* attacks, that extract unauthorized information about the underlying filesystem. Our access control technique is resilient towards such attacks and provides greater indexing efficiency due to its distributed architecture.

This dissertation also addresses the challenge of managing large dynamic SSP data centers. The theme of our contributions in this domain is to develop autonomic management techniques that assist SSP administrators to quickly and efficiently integrate client-initiated changes. We have developed techniques that analyze the impact of a change and can plan to efficiently accommodate the change, with minimal administrator involvement. The former, Zodiac is a "*what-if*" analysis framework that proactively analyzes the impact of a proposed change on the storage area network, before actually applying the change. This prevents expensive change related operational errors and reduces downtimes. The latter, SPARK, is a novel resource allocation framework, that can quickly re-assign resources to applications

in response to a client workload surge, device failure or planned growth and downtime. SPARK is unique in its ability to provide coupled storage and CPU resource allocation and effectively exploits the heterogeneity of storage area networks for greater efficiency.

In summary, this thesis addresses two important technical challenges facing the networked storage services model. By using our access control *embedding* schemes, enterprises can secure their data without placing trust in the storage service provider and the service providers can use our change management and resource allocation techniques for adequately dealing with their infrastructure dynamics.

CHAPTER I

INTRODUCTION

With continued advances in communications and computing, the amount of digital data continues to grow at an astounding rate, doubling almost every eighteen months [65]. Not just in size, the role of data in modern enterprises has increased in significance as well. Enterprises are now storing more data and also keeping it for a longer period of time for both business intelligence and regulatory compliance purposes. This trend has put tremendous strain on enterprise storage infrastructures.

While the cost of storage hardware has dropped, storage management has become increasingly complex. A storage administrator has to perform many complex tasks like provisioning, performance bottleneck analysis, change analysis, disaster recovery planning and security analysis. As a result, managing enterprise storage has become very expensive and is actually estimated to be 75% of the total cost of ownership [25]. In contrast, enterprise IT budgets have grown only slightly over the years [99]. This has forced enterprises to look for new ways to effectively manage their data.

One paradigm finding success in alleviating storage management costs is that of *Networked Storage Services* or as more commonly known *Storage-as-a-Service*. Much like an email or a web hosting service, this paradigm delivers *raw storage* as a service over a network. In this model, as shown in Figure-1, enterprises use an external storage service provider (SSP) to store their data at a remote SSP-managed site and access it over a high-speed network (public or private)[1]. The use of an external party as a storage service provider is often highlighted through the term *outsourced storage*.

This Storage-as-a-Service paradigm offers various advantages to an enterprise:

1. **Storage Management**: Obtaining storage as a service frees the enterprise from expensive and expertise-intensive storage management task. The storage devices are

[1] When the network is the public internet, this model is also referred to as *internet storage*.

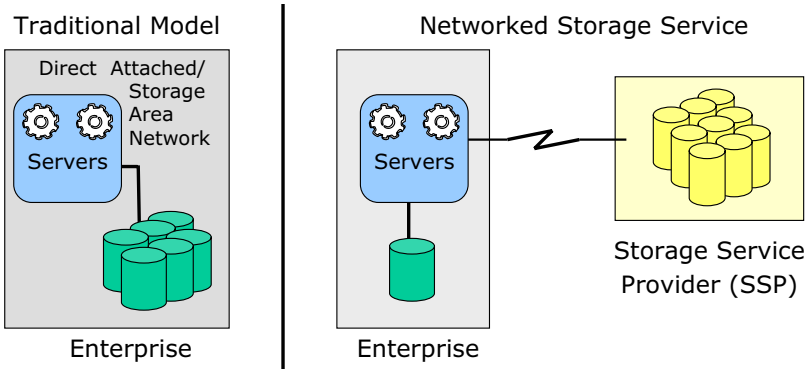


Figure 1: Networked Storage Services Paradigm

owned and managed by the SSP which uses economies of scale to amortize costs and efficiently manage large amounts of data. In fact, this model is highly suitable for enterprises that are considering expanding their storage infrastructures to a SCSI based Storage Area Networks (SANs) which cause a significant rise in management complexity. Hasan et al [77] cite a study by Goldman-Sachs that estimates over 350% cost savings by outsourcing storage to SSPs.

2. **On-demand Storage Infrastructure**: Through Storage-as-a-Service, enterprises obtain an adaptive storage infrastructure. Their storage infrastructure can now grow or shrink according to their needs using on-demand storage provisioning and decommissioning. Enterprises pay only for the amount of storage that they use and only for the duration that they use it. This avoids expensive over-provisioning.

3. **Disaster Recovery**: Most SSPs package their raw storage services with superior disaster recover (DR) and backup solutions. SSPs use multiple geographically-separated sites to store backups and can even provide a *hot standby*. This offers great value to enterprises especially those that are mandated by regulations to maintain such backups, for example, health care and medical facilities. Many SSPs like [12, 93] nowadays also provide versioning support, in which every write to data is versioned and users can go back to any previous version in history.

4. **Content Dissemination**: Enterprises can also use superior bandwidth and availability characteristics of SSPs for more efficient content dissemination to its users. This

is especially useful when handling temporary surges in workloads without buying new hardware/bandwidth or acquiring more management capabilities.

Interestingly, this paradigm gained substantial popularity (and subsequent notoriety) during the dot-com era. Many internet companies, that did not have adequate IT staff and budgets used external storage service providers for their data storage needs. However, as the dot-com companies failed, so did most storage service providers. We refer the reader to [77] for an interesting historical analysis of the storage-as-a-service paradigm.

With improved economic climate, continued data growth and increasing reliance on any-time access to data, this model is re-emerging and many big and small companies offer storage-as-a-service products like Amazon S3 [6], SUN Grid [171], Arsenal Digital [12] and Iron Mountain Digital [93]. However, a widespread acceptance of this model within enterprises has been impeded due a number of technical challenges. We discuss some of these challenges in the following section.

Note that an enterprise can outsource its storage in many forms – (a) block storage, (b) filesystems, or even (c) structured databases (*Database-as-a-Service* [74]). In this dissertation, we primarily address outsourcing of enterprise filesystems which, according to industry estimates, contain nearly 85% of all enterprise data [27].

1.1 Challenges

In this section, we describe some of the important challenges facing the storage-as-a-service paradigm. In Section-1.1.1, we describe security and access control issues associated with storing confidential data at an external service provider. Section-1.1.2 describes the service provider's challenge of managing large data centers, while providing an on-demand storage infrastructure to enterprises. We describe the enterprise-to-SSP network connectivity and performance complexities in Section-1.1.3 and finally, few non-technical challenges faced by this model are briefly described in Section-1.1.4.

1.1.1 Security and Trust

By outsourcing storage to the SSP, enterprises trust it to reliably store data and provide high availability access to it. However, this trust can not be translated to critical security issues

like data confidentiality and access control[2] and pressing questions need to be addressed:

- **Data Confidentiality** – *Can the SSP be trusted to respect data confidentiality?*

  If yes, data can be stored in plaintext at the SSP (and thus can be read by SSP and its employees). Most enterprises today do *not* like to associate this trust in the SSP. This could be due to the requirement to protect intellectual property or to comply with government regulations, for example, regulations protecting privacy of patients in handling medical data. To preserve data confidentiality from the SSP, enterprises would have to employ cryptographic techniques and integrate them with their existing storage infrastructures. This can raise complex key management challenges for multiuser environments where different users have different privileges to data.

- **Access Control Enforcement** – *Can the SSP be trusted to enforce access control policies on enterprise users?*

  Another form of security issue raised in this model is whether the SSP should be trusted to enforce access control. As an example, an enterprise can choose to encrypt its data (to preserve confidentiality) using a single symmetric key. It can then give this encrypted data to the SSP along with access control policies on how different files (encrypted) can be accessed by different users. Many enterprises find this form of trust to be unacceptable as well. Other than the core issue of trusting a third party for enforcing *your* access control, this also raises an issue of verifiability – it is tough to verify and monitor the SSP on its ability to enforce access control. Secondly, if an insider user (who possesses the encryption key) colludes with the SSP, all enterprise data can be compromised. Such insider attacks are increasingly becoming common [29] and enterprises aim to prevent such drastic confidentiality breaches.[3] These no-trust environments in the storage-as-a-service model are a significant departure from traditional access control architectures that rely on a trusted reference

[2] A useful guideline for identifying trusted or non-trusted attributes is *verifiability*. If an enterprise can detect and verify an attribute then that can be entrusted to the SSP by inclusion in the Service Level Agreement (SLA) with penalties if inadequately delivered.

[3] Even the SSP could prefer a no-trust security model as it rids them of all liability that a potential data confidentiality breach could cause.

monitor and new access control techniques need to be developed.

- **Data Integrity** – *How to ensure integrity of stored data?*

  A related security issue is detection of unintentional or malicious attempts at modifying stored data. As the SSP is not fully trusted, new cryptographic integrity and signing protocols are required that can detect unauthorized modifications to data and resist replay or rollback attacks, in which a malicious user or SSP attempts to replay (appropriately-signed) old data contents. Recently, there have been a number of research efforts that have proposed techniques [106, 94, 67] and can provide high levels of integrity and consistency.

1.1.2 Storage Management

The security challenge described above addressed complexities associated with the client-enterprise side of the storage-as-a-service model. On the service provider's side, primary complexity is involved in efficiently managing large amounts of data. SSPs could be required to manage petabytes of storage, while providing an on-demand infrastructure to enterprises, for example, immediately able to provision a terabyte of storage. While significant work has been done recently to improve overall storage management including techniques like policy-based management [3] and autonomic computing [5, 9, 10, 64], one characteristic distinguishes the SSP infrastructure from typical enterprise environments – the *highly dynamic* nature of its environment.

Changes to storage infrastructures are typically dreaded as they are highly error-prone. In fact, estimates by Gartner indicate that 70% - 80% of changes resulting in downtimes are initiated within the organization [114]. As a result, introducing changes requires significant lead-time for adequate planning by storage administrators. This planning involves tasks like "what-if" analysis, resource allocation/de-allocation planning and network and security analysis. Enterprises managing their own data can control introduction of changes into their environment and also usually overprovision their systems in order to quickly integrate changes.

In contrast, for a SSP providing an on-demand storage infrastructure, long lead-times

for integrating changes are unacceptable as clients would immediately require access to a modified environment. Secondly, SSPs are typically under greater pressure to manage their resources as efficiently as possible since that is their core specialty and revenue earner. As a results SSPs can not afford to grossly overprovision their systems. To support these highly dynamic environments in an efficient manner, development of new change-management techniques are required that eliminate or at the least minimize manual planning and provide efficient resource allocation plans.

1.1.3 Network Connectivity and I/O Performance

The next challenge relates to the interface between the client-enterprise and the SSP. Client enterprise access to data stored at the SSP site would typically go over a Wide Area Network (WAN) or atleast a Metropolitan Area Network (MAN). Traditional Fibre-channel based Storage Area Neworks (SANs) can not go over long distances, and it requires use of channel extenders or other technologies like Wavelength Division Multiplexing (WDM) [175] and Synchronous Optical Networking (SONET) [160] for low-latency data transfer. Most of these technologies are in use today though primarily for disaster recovery and long-distance replication. Integrating them into primary storage access paths and to provide filesystem like access over these could require new mechanisms. Additionally, OS and filesystem caching work like [131, 139] needs to be adopted to provide efficient wide area file services (WAFS).

1.1.4 Business Challenges

For greater acceptance of the Storage-as-a-Service model, certain non-technical challenges also need to be resolved. Enterprises may find it unacceptable to lock-in their data with a single storage service provider and their long term storage needs linked to the service provider's business success. Another tricky issue is negotiation of the Service Level Agreements (SLAs) with appropriate monitoring and penalties built into the agreement. Additionally, as many enterprises are mandated by government regulations to store their data for a long time, issues relating to legal liabilities in case of a disaster and data loss, need to be resolved as well.

These challenges manifest themselves in different forms and severity depending upon

the architecture and model used for providing storage as a service. We discuss some of the important storage-as-a-service models in the next section.

1.2 Storage-as-a-Service Models

Storage can be delivered as a service in a multitude of ways, with each model offering distinct set of advantages and challenges. The design of models varies depending upon the I/O path chosen for data and metadata, centralized or decentralized access and complexity of key management. In this section, we discuss a number of service models and give our recommendation for a model, that we believe offers superior advantages. However, first we begin by differentiating storage-as-a-service from a related application services model.

1.2.1 Comparison with Application Service Provider Model

It is important to distinguish storage-as-a-service from the known application service provider (ASP) model. Figure-2 contrasts the two paradigms. In the ASP model, enterprises not only outsource their data, but also certain applications to the service provider. Users of the enterprise can then access these applications through internet based frontends. Various companies like IBM [92], HP [79], Salesforce.com [154] provide such application hosting services.

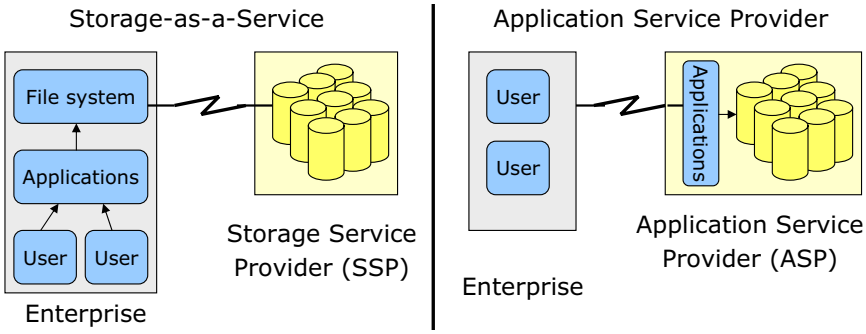


Figure 2: SSP vs. ASP model

The ASP model helps enterprise offload application deployment and management complexities along with their storage. However, it suffers from two main disadvantages:

1. **Security**: Because of application deployment at the service provider, data needs to be stored in plaintext as well. This is unacceptable for many enterprises that aim

to protect their intellectual property or to comply with government regulations. In contrast, the SSP models can store encrypted data at the SSP and thus preserve data confidentiality.

2. **Flexibility**: By outsourcing applications to the service provider, any custom application deployment requires service provider's approval and co-operation which can limit an enterprise's flexibility to modify its environment. In contrast, in the SSP model applications continue to run at the enterprise's site and the environment closely resembles a local storage model from an application perspective.

In this dissertation, we consider the requirements of security-conscious enterprises and use the secure Storage-as-a-Service model. In the next three sections, we discuss different models for providing storage as a service.

1.2.2 Storage Services using Enterprise Proxies

The first service model uses a single point of access to store and retrieve data stored at the SSP. As shown in Figure-3, the enterprise location uses an encryption filesystem to encrypt/decrypt data stored at the SSP and uses a proxy gateway file server that serves data requests from users that are accessing from other locations[4] .

To transition from a local storage to this model, enterprises only need to set up an encryption filesystem [19, 197] that encrypts all data before writing it to SSP and correspondingly decrypts data when reading. Users are oblivious to the SSP and only communicate with the gateway file server through commonly used protocols like *nix based Network File System (NFS) or Windows based Common Internet File System (CIFS) protocol. In this model, it is easy to preserve data confidentiality, with easy key management (in fact, all data can be encrypted with a single key only accessible by the enterprise encryption filesystem). Also, access control is enforced by the trusted enterprise. However, this model has the following undesirable characteristics:

[4]In fact, as a general practice, filesystems that are exported through file servers are exclusively used in that mode and local direct access is less frequent. Thus, in that case, even users within that enterprise location would access data through the gateway server.

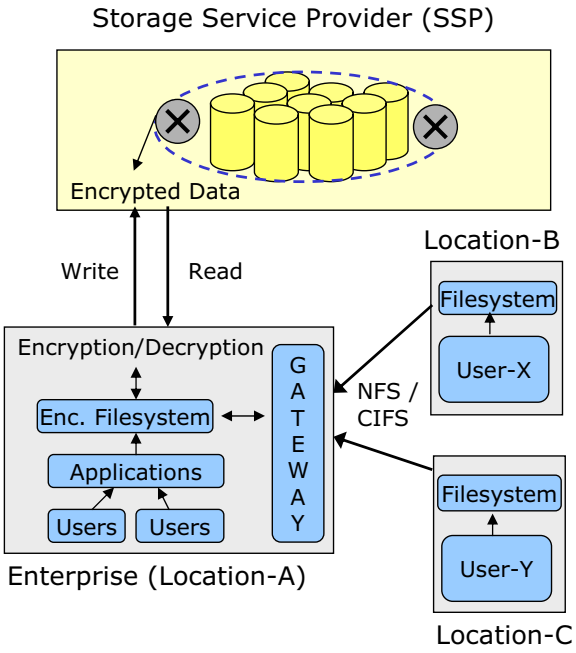


Figure 3: Proxy Storage Services model

- *Centralization*: The proxy services model uses a central gateway for all data and metadata I/O. This would cause I/O bottlenecks and not scale well for large workloads. It also becomes a single point of failure and eliminates the potential usage of SSP as a content dissemination service.

- *Additional Network Hop*: Since only the enterprise file system accesses data from the SSP directly, file accesses by users from other locations would require an additional network hop. This would adversely impact I/O performance, though data caching can mitigate some of these costs.

- *Two Cryptography Steps*: For every data access, this model requires additional cryptographic operations. For example for a `read` by a user, the enterprise filesystem would first decrypt the data when accessing from the SSP (since only the enterprise filesystem knows the master encryption key) and then re-encrypt it for sending it over-the-network to outside users, who would have to decrypt it again. If users were allowed to access the SSP directly, additional cryptographic costs can be avoided.

- *Gateway Management*: Finally, in this model the enterprise is responsible for managing the gateway file server, which would also have administration costs.

In summary, this model is more appropriate if the access to enterprise data occurs only at a single enterprise location. The centralization and cryptographic costs make it less suitable for distributed access to data.

1.2.3 Metadata Services using Enterprise Proxies

The second service model aims to reduce the data centralization impact of the first model by splitting data and metadata I/O paths. In this model, shown in Figure-4, the enterprise gateway serves as a metadata server and only provide information about data like address for data blocks and encryption key for the requested file. Data is then obtained directly from the SSP. Similar (non-encryption) *out-of-band* filesystems [117, 140] are in use today and a recent standardization effort for parallel NFS (pNFS) [141] is developing protocol standards for clients.

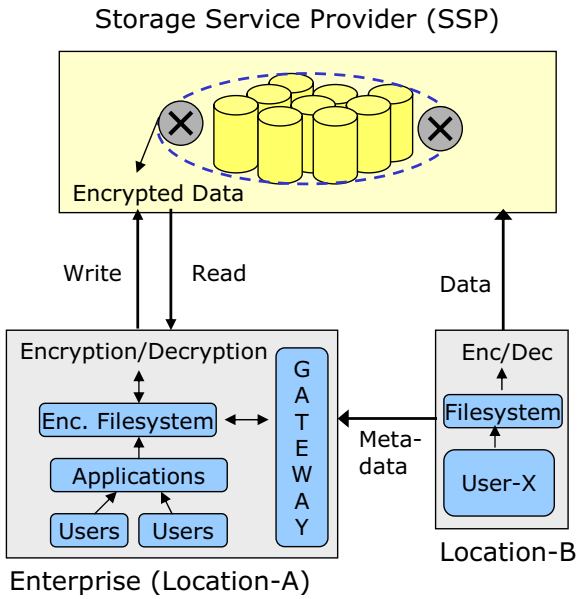


Figure 4: Metadata-Proxy Storage Services model

While this model removes some data bottlenecks, metadata I/O continues to be centralized and the costs of opening two network connections (for data and metadata) can be

expensive, especially while accessing small files. Secondly, the encryption strategy is different from the previous model in the following manner. Since clients access data directly from the SSP and we do not trust the SSP for access control enforcement, we have to ensure that no user obtains decryption keys for data that he/she can not access. As a result, we have to encrypt each file with a different key and only users that have permissions to access that file can obtain that key from the metadata server. Thus, there are greater number of keys to manage now and clients have to perform encryption/decryption as well. Additionally, the management of gateway metadata servers continues to be the enterprise's responsibility.

1.2.4 Decentralized Storage Services Model

The decentralized services model considers an environment where client-enterprise and its users access data directly from the storage service provider (Figure-5). This eliminates any data or metadata I/O bottlenecks and is most suitable for using the SSP as a content dissemination service. Additionally, there are no management requirements for a gateway server.

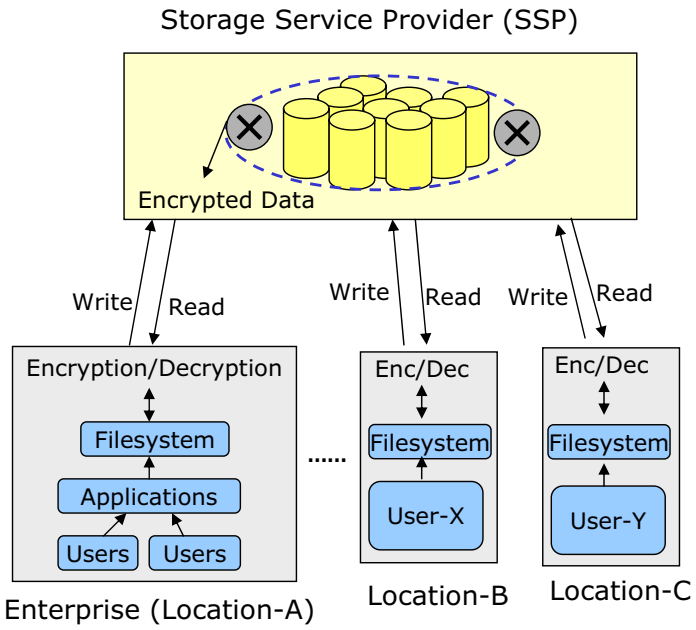


Figure 5: Decentralized Storage Services model

Similar to the metadata proxy model, each file would need to be encrypted with a different key, but distribution of keys to users has to be managed in a decentralized manner.

Table 1: Comparing Storage-as-a-Service Models

Feature	**Enterprise Proxy Model**	**Metadata Proxy Model**	**Decentralized Model**
Centralization	Data and metadata	Metadata	none
I/O Performance	Two N/W hops	Two N/W connections	Single hop and connection
Cryptography Costs	Additional crypto	Regular crypto	Regular crypto
Management	Gateway management	Gateway management	No gateways
Key Management	Easy (can use one key)	Many keys	Many keys

Some related work in this area [67, 2] has addressed the key management problem by using public key cryptography schemes [115], which are known to have poorer performance. However, faster symmetric key cryptography based designs are possible and we describe our system, xACCESS, in Chapter-2. Another complexity involved in this model would be the usage of decentralized locking techniques to ensure data integrity and consistency. Recent work in this area [106, 67] have addressed many of these concerns.

1.2.5 Model Recommendations

In this section, we summarize various storage-as-a-service models and provide our recommendations for choosing a particular model. Table-1 summarizes various aspects of the three models.

We believe that by centralizing data and metadata I/O through an enterprise proxy, many of the original benefits of the Storage-as-a-Service model are lost – both in terms of performance and management costs. While the metadata proxy model alleviates data centralization, it still centralizes metadata requests, which will have an impact on performance. It also requires continued management of the proxy gateway by the client enterprise and can be a single point of failure. We prefer the **decentralized service model** as it provides the best environment to exploit all benefits of the Storage-as-a-Service model. Additionally, we have developed efficient in-band key distribution techniques that mitigate the key management complexity of the decentralized model.

Next, we describe the research hypothesis for our security and storage management work

for the decentralized model in this dissertation.

1.3 Research Hypothesis

This dissertation research is based on the following three hypothesis:

- *Access Control Embedding*: In the decentralized storage-as-a-service model, there is no trusted access control engine in data access path that can enforce access control on enterprise users. To enforce access control in this model, enterprises can *embed* access control into the stored data itself using cryptographic mechanisms, which ensure that users can only decrypt data that they had been authorized to access. Secondly, using efficient symmetric key cryptographic operations and in-band key management techniques, it is possible to build an efficient access control system that provides expressive semantics over the decentralized storage-as-as-service model.

- *Access Control Embedding for Applications*: The access control embedding concept can be extended to support other access control aware applications that require a trusted enforcement engine at runtime, for example, multiuser filesystem search. For such a service, access control needs to be enforced during search ensuring that no user can search through data that he/she is not authorized to access. Most common enterprise search approaches rely on a trusted access control monitor at runtime for filtering of query results. Additionally, such techniques are prone to various inference attacks that leak private information to unauthorized users. In contrast, by embedding access control into indices, enterprise search can be secured against common inference attacks and the search service can be securely hosted at an untrusted service provider.

- *Autonomic Management*: As the storage service provider aims to provide an on-demand storage infrastructure to clients, complexity is introduced in its management due to manual administrator involvement in change analysis and resource allocation. For efficient management of such environments, autonomic techniques can be used to (a) proactively analyze the impact of a change on the infrastructure, and (b) obtain high quality resource allocation plans for improved utilization and performance. Such

techniques will reduce costs and provide quicker integration of client initiated changes into the SSP infrastructure.

In the next section, we describe the specific contributions of this thesis.

1.4 Thesis Contributions

To the best of our knowledge, this dissertation is a first comprehensive attempt at addressing core challenges encompassing the enterprise storage-as-a-service paradigm. In this dissertation, we have proposed novel techniques along two dimensions – (a) security and access control, and (b) SSP storage management. Specifically, we have made the following contributions:

1. **Access Control with xACCESS**: We have developed an access control system, called xACCESS, that uses novel cryptographic access control primitives (CAPs) to *embed* access control into stored data and not rely on the SSP for enforcement of security policies. xACCESS is able to support an expressive UNIX-like access control model and its in-band key management technology ensures a seamless transition from local storage to the storage-as-a-service model with negligible user involvement. We have also analyzed the privacy characteristics of the access control model and developed enhancements that provide better privacy support to users while sharing their data. To the best of our knowledge, xACCESS is the first such system that provides an expressive access control model in the decentralized storage services model and it also outperforms related proposals by over 40% on a number of benchmarks.

2. **Secure Multiuser Filesystem Search**: We use a similar concept of access control embedding to develop secure multiuser keyword-search for SSP-stored data without trusting the service provider. Our approach uses a novel access control barrel (ACB) primitive to support access control aware search ensuring that users can only search through files that are accessible to them. Unlike existing enterprise search techniques our approach is resilient to common attacks that can leak private information. Our access control embedding architecture is a first such technique in which a search service can be hosted at an untrusted service provide for multiuser environments.

3. **Efficient Management of SSP Data Centers**: To efficiently handle client-initiated changes into SSP storage area networks (SAN), we have developed techniques for (a) analyzing the impact of a change (Zodiac) and (b) to plan for efficiently accommodating that change (SPARK). Zodiac is a proactive impact analysis engine, that can autonomically analyze the impact of a change on the system before actually applying that change. This provides a useful tool for enterprises to prevent change related operational errors. In case the proposed change requires re-allocation of resources, we have also developed a resource allocation framework, called SPARK. It uses modern virtualization technologies to quickly re-assign resources to applications in response to client workload surges, planned growth, failures or scheduled downtimes. SPARK is a first resource allocation engine that provides integrated allocation of application storage and CPU, accounting for affinities between CPU and storage nodes.

1.5 Thesis Organization

This thesis is organized as follows:

Chapter-2 describes the design of our access control system, xACCESS for enterprise storage-as-a-service environments, including the architecture and implementation of the xACCESS prototype. We also analyze its data sharing mechanisms and propose enhancements for more secure and convenient data sharing. In Chapter-3, we describe the design and architecture of our secure multiuser search scheme that provides keyword search over SSP-hosted data without trusting the SSP.

Next, we describe efficient change management and resource allocation techniques for managing dynamic storage area networks (SANs) of the storage service provider. In Chapter-4, we describe Zodiac, the "what-if" analysis engine for policy enabled SANs. Chapter-5 describes our new resource allocation mechanism for integrated storage-CPU placement in SSP SANs.

We conclude the thesis with directions for future work and open problems in Chapter-6.

CHAPTER II

ACCESS CONTROL WITH XACCESS

In the decentralized storage-as-a-service model, all users access data directly from the storage service provider. Also, as mentioned earlier, in our security model the service provider is not trusted for access control enforcement. This lack of a trusted server in the I/O path makes security and access control, a particularly challenging problem. In this chapter, we will describe our novel access control system called xACCESS that can provide an expressive UNIX-like access control model over outsourced data. It has additional desirable properties of in-band key management and superior performance. We will also analyze the *privacy* support in its data sharing mechanisms and propose enhancements using a novel *view* primitive.

It is important to note that the xACCESS system is a solution for data confidentiality and access control for the decentralized storage-as-a-service model. These are but only two key pieces of the overall security strategy for any enterprise. For example, issues like intrusion detection, firewalls and protection against denial of service (DoS) attacks still need to be addressed. We continue to investigate unique challenges posed by such security technologies in the decentralized storage-as-a-service models as part of our future work.

We start with some background information relating to UNIX access control, filesystems design and cryptographic operations.

2.1 Background Information

In this section, we describe key background concepts, the knowledge of which is important for understanding xACCESS.

2.1.1 UNIX Access Control Model

The UNIX access control model [148], also supported by Linux and other flavors (together called *nix) is a discretionary access model in which each file system object (file, directory,

links) has an associated owner who controls the access to that object. This access can be granted to three kinds of users: (1) *owner*, (2) *group*, and (3) *others*. The *owner* is the object owner, the *group* is a set of users to which the owner belongs (for example, a user group for students, faculty) and *others* are all other users. In notation, the permissions bits are listed in the {*owner, group, others*} order.

Further, the granted access is of three types:

1. **Read:** For a file, this means that a user can read a file. For a directory, this means that a user can list its contents using `ls` [107]. For links, the permissions are for the object that are pointed-to by the link and the link's permissions are not used. The read permission is represented by a 'r'.

2. **Write:** For a file, the write permission allows a user to write to a file. For a directory, it allows a user to add/delete/rename directory contents. For links, the permissions are again for the pointed-to object. The write permission is represented by 'w'.

3. **eXecute:** For a file, the execute permissions allows running the file as a program (for example, a shell script). For directories it allows users to traverse that directory and if the directory contents have appropriate permissions, the user can then access those contents. For example, to access directory `grand-child` with path `dir/child/grand-child`, both `dir` and `child` need to have execute permissions. The execute permission is represented by a 'x'.

To improve the granularity of user permissions, lately POSIX Access Control Lists (ACLs) [72] have been introduced into many *nix variants. These allow setting permissions at individual user levels as opposed to *group* or *others*. However, the {*owner, group, others*} is still the most dominant paradigm and many *nix installations do not even have ACLs enabled.

2.1.2 Filesystem Data Structures

In this section, we describe important data structures used in filesystems. While many concepts are similar in most filesystems, our discussion specifically is based on the Linux

Ext2 file system and we refer the reader to [30] for a more detailed description.

A filesystem is composed of various objects like files, directories, symbolic links and hard links. Each of these filesystem objects is stored using two data structures – *metadata* and *data*. The metadata contains information about various attributes like a unique inode number, owner ID, access permissions, and size. It also contains pointers to data blocks for that object. The metadata objects are stored indexed by the inode number.

The data blocks for a file contain its content. For a directory, the data blocks contain a *directory table* with two columns containing the names and inode numbers of all subfiles and subdirectories contained within that directory. Using these inode numbers, the metadata for those subfiles and subdirectories can be looked up (*traversing* the directory). For a symbolic link, the data blocks contain the path of a target filename to which the link points. A hard link to a file or directory is only a different entry into the directory table but uses the same inode number, thus pointing to the same metadata object.

For xACCESS, we need to design similar data structures in the decentralized storage-as-a-service model.

2.1.3 Symmetric and Public-key Cryptography

Cryptographic operations on data can vary significantly in performance based on the form of cryptography. In **symmetric key cryptography**, a single key is used to both encrypt and decrypt data. Examples of such schemes include AES [115, 133] and DES [115, 134]. Symmetric key cryptography is known to be very efficient and can provide throughputs greater than 70 MB/s on a regular desktop machine [2].

In **public key cryptography**, there are a pair of keys associated with a user – (a) *public key* that is openly published and (b) *private key* that is held private by a user. These keys are asymmetric in nature such that any data encrypted with the public key can only be decrypted with the private key and vice versa. The most common public key scheme is RSA [115, 150]. Public key schemes are known to have worse performance. Additionally, encryption/decryption with the private key is much more expensive than with the public key as the private key is longer in size. Thus, it is preferable to use symmetric key cryptography

as much as possible in all filesystem operations.

2.2 Access Control Challenges

Next we describe various challenges involved in enforcing access control in the decentralized storage services model. We illustrate these challenges using an example shown in Figure-6.

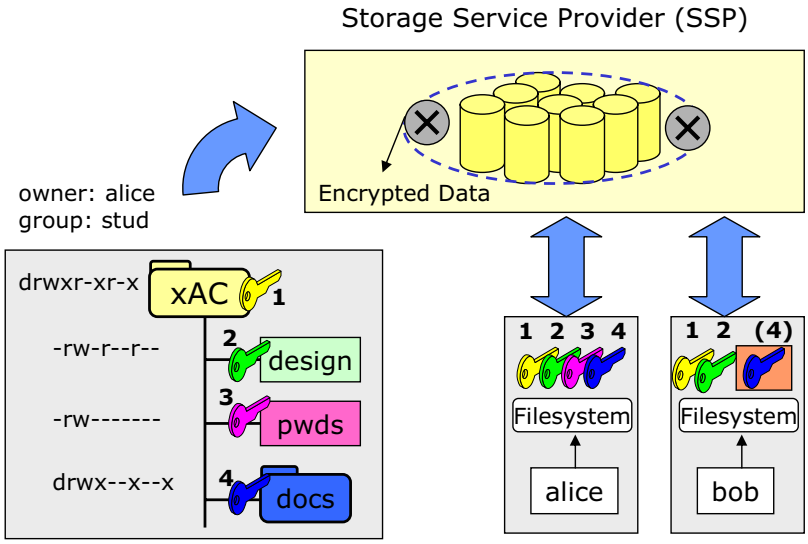


Figure 6: Challenges in Access Control Enforcement

Consider an enterprise with a filesystem consisting of the directory structure as shown in bottom left corner of Figure-6. It includes multiple files and directories with different sets of permissions for different users (figure shows UNIX-based permissions). Assume that administrators decide to outsource this filesystem by sending data to the SSP[1]. Without a trusted access control engine at the time of data access, access control can only be enforced by "*embedding*" control information into the data. This is accomplished by encrypting each file with a unique key and controlling the distribution of these keys, ensuring that users can only obtain keys for files that they are authorized to access. For example, data owner `alice` possesses all four keys that allow her to access all data, whereas `bob` does not have the key for the file "`pwds`" since it did not have `read` permissions for users other than `alice`.

[1]This decision could be based on various reasons. For example, it could be that the utility of this data has dropped and it is better to use expensive local storage for sometime more important. Or on the opposite end of the spectrum, it could be that too many users are accessing this data and the enterprise wishes to use better dissemination abilities of the SSP.

This access control embedding is complex due to the following reasons:

- **Key Distribution**: In the decentralized storage service model, each user requires keys for all files that he/she can access. Distributing this large number of keys to users in a decentralized manner is a challenging problem. Some related work in this area [94, 76] proposed using out-of-band channels for distribution. For example, if `alice` wishes to share a file with `cathy`, she would have to email the key for that file to `cathy`. We believe that a good solution to this problem should ensure that user involvement is minimal in managing keys. This will ensure that the transition from local storage to the storage-as-a-service model is the least disruptive. Our xACCESS system accomplishes this by completely hiding key management details from users. Instead the xACCESS filesystem handles key management in-band and users continue to access the system using regular mechanisms.

- **Access Control Expressiveness**: Another important challenge in the decentralized service model is to provide an expressive access control model like the ones used in typical local systems (for example, UNIX permissions [148]). Most of the related work [94, 67, 2, 126] provides only a restricted access control model with few permission settings. We believe this to be a critical limitation for two reasons – (a) users are accustomed to using typical access control models and changing the model requires re-education, and (b) in the process of transitioning from local storage to the storage-as-a-service model, if users want to provide equivalent security semantics on their data, they would have to be manually involved in mapping permissions to the new model. This would again cause greater disruption and slow the transition process. Our xACCESS system is able to provide an expressive access control model by carefully manipulating filesystem metadata and key distribution. For example, in Figure-6, user `bob` can only obtain keys for contents of directory "`docs`" if he knows the names of the subfile or subdirectory (equivalent to the execute-only semantics for UNIX directories [148]).

- **Performance**: In the storage-as-a-service model, each data and metadata access requires cryptographic operations – decryption for `read` and encryption for `write`. To minimize its impact on performance, it is crucial that the system uses the efficient symmetric key cryptography as much as possible. Most related work [67, 2] does use symmetric key cryptography for file *data* encryption but relies on the expensive public key cryptography [115] for *metadata* as it makes key distribution easier. For example, by encrypting metadata objects (containing data decryption keys) with public keys of users that are authorized to access those objects, it is easily ensured that only those users obtain access. However, this comes with a performance penalty for every metadata operation. In contrast, xACCESS predominantly uses symmetric key cryptography for metadata operations and is able to outperform these comparable approaches by over 40% on a number of filesystem benchmarks.

Next, we describe the basic concepts and data structures used in xACCESS.

2.3 Basic Concepts and Data Structures

Before we get into the detail of xACCESS design, we describe some of the basic concepts and infrastructure requirements for xACCESS.

2.3.1 User and Group Keys

In xACCESS, each user has a public-private key pair denoted by $< B_u, P_u >$, where B_u is the public key and P_u is the private key known only to user u. This key pair effectively serves as the identity of the user. User groups also have a similar public-private key pair $< B_g, P_g >$. We also assume that each user knows the public keys for all other users. This would imply existence of a public key infrastructure or usage of Identify-Based Encryption [22] schemes in which the email address of the user is a valid public key. Also, the group keys are distributed to users by storing them encrypted with the public keys of group members (individually). Figure-7 shows the table structure for distributing keys for a user group `student` ($< B_{stud}, P_{stud} >$), which has two members `joe` and `jane`. E_K denotes encryption using key K.

Hash(uid)	Encrypted Key Block
Hash(joe)	$E_{Bjoe}\{<B_{stud}, P_{stud}>\}$
Hash(jane)	$E_{Bjane}\{<B_{stud}, P_{stud}>\}$

Figure 7: Secure Group Keys Storage at SSP

These encrypted group keys are stored at the SSP. When a user, say `jane`, logs into the system (that is, mounts the SSP file system), she obtains her encrypted group key blocks and uses her private key to decrypt and thus obtain her group keys. The first column containing hash of the user ID is used to index the key blocks, so that a user directly obtains only his/her blocks. By hashing the user ID, identity of the users is held confidential from the SSP. For hashing, any secure hash function such as SHA1 [53] or MD5 [149] can be used.

2.3.2 Encrypting Data and Metadata

As described earlier, all file and directory **data** is encrypted using distinct symmetric keys. In order to handle large files efficiently, files are typically divided into multiple blocks and each block is encrypted separately. This helps accommodate updates more efficiently by avoiding the need to re-encrypt an entire file after a `write`. The number of blocks per file does not impact xACCESS design in any manner and for ease of exposition, we will assume that all file data fits into a single data block. Also recall that a directory's data block contains the directory-table structure (as described earlier in Section-2.1.2).

Each data block has two sets of keys associated with it:

1. **Data Encryption Key (DEK)**: The DEK is a unique symmetric encryption key used to encrypt the data block. The symmetric nature implies that the data block is encrypted and decrypted using the same key. As an example, DEK could be a 128-bit AES [133] key.

2. **Data Signing (DSK) and Data Verification Key (DVK)**: The DSK and DVK are a pair of asymmetric keys such that any content signed (that is, encrypting the hash of content) with the DSK, can only be verified (decrypting signature and comparing with actual hash of content) with the DVK and conversely, verification with DVK only

succeeds if content was signed with the DSK. Signing and verification is required to differentiate readers from writers. Note that we use symmetric keys for *data* encryption for performance reasons. So any user who has `read` permissions on a file, thus possesses the DEK, can attempt to write to that file as well (by encrypting new content with DEK). Since in our security model, we do not trust the SSP for enforcing access control, we have to develop mechanisms to detect any such malicious attempts at writing (by users or even the SSP). Signing and verification is one such technique. We ensure that only writers can obtain the DSK and whenever a file is modified by a writer, they sign the hash of the file content with the DSK. Now, all readers, who possess DVK, can verify whether the file was written by an authorized user. This provides us a mechanism of distinguishing writers from readers without trusting the SSP. Note that while public key schemes like RSA [150] can be used for signing and verification, there are other techniques like ESIGN [135, 136] that are over an order of magnitude faster [106].

2.3.2.1 Metadata

In xACCESS, we use symmetric key cryptography for **metadata** objects as well. Thus, we have a similar set of keys:

- **Metadata Encryption Key (MEK)**: MEK is a unique symmetric key used for encrypting metadata objects, for example, 128-bit AES.
- **Metadata Signing Key (MSK)**: Similar to the data signing key, the MSK is distributed only to users that can write to the metadata object. In our current implementation, we limit this to the object owner[2].
- **Metadata Verification Key (MVK)**: MVK is used by readers of metadata to verify whether it was written by an appropriate user.

Table-2 summarizes the basic infrastructure and the notation used in xACCESS.

[2]There are certain metadata attributes that can be modified by readers or data-writers as well, for example, *last-access-time* and *size*. We describe how we handle those attributes in the next subsection.

Table 2: xACCESS Basic Concepts

For	Concept	Notation	Usage
User	Public Key	B_u	Serves as identity and aids in
	Private Key	P_u	exclusive information exchange
User Group	Public Key	B_g	Serves as identity and aids in
	Private Key	P_g	exclusive information exchange
Data Block	Data Encryption Key	DEK	Encrypts/decrypts data
	Data Signing Key	DSK	Writers sign data with DSK
	Data Verification Key	DVK	Readers verify data with DVK
Metadata	Metadata Encryption Key	MEK	Encrypts/decrypts metadata
	Metadata Signing Key	MSK	Owners sign metadata with MSK
	Metadata Verification Key	MVK	Readers verify metadata with DVK

Next, we describe the internal data structures for xACCESS filesystem. This design plays a key role in providing an expressive access control model, while primarily using symmetric key cryptography for metadata objects.

2.3.3 Key Data Structures

There are two key data structures in the xACCESS filesystem – (a) metadata, and (b) directory table.

2.3.3.1 Metadata

As described in Section-2.1.2, traditionally a metadata object consists of various attributes for a file (or directory) like inode number, owner, group, permissions, size and it also contains pointer to the data block for that object. To read a file (or directory), the user first looks up the metadata object for that file's inode number and then follows the pointer to the data block. In xACCESS, data blocks are encrypted and appropriate keys to read/sign/verify need to be distributed to appropriate users.

To do this key distribution in-band, conceptually, we rely on the same semantics of *metadata leads to data* and add three new fields to the metadata structure for DEK, DSK and DVK for the data block of that object. The idea is that now metadata not only points to the data block but also provides knowledge (keys) to appropriately read/write to that data block. Of course, not all users will have the DSK field populated (only writers do) and this selective availability of keys is what gives us an expressive access control model. We

discuss this issue further in Section-2.4.

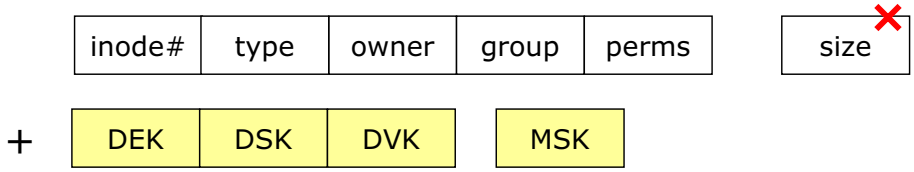


Figure 8: xACCESS Metadata

Figure-8 shows the modified metadata structure. There are additional modifications to the metadata structure as well. First, the MSK is also included within the metadata. For owners of the file or directory, the MSK will allow them to sign the metadata object when they update it, for example, while changing permissions of the file. The second modification is to remove the `size` field from the structure. Since we do not want any user who has permissions to write to a file (thus modify its size), to have an equivalent permission to write to the metadata structure, we take out the `size` field and store it within the first data block[3].

2.3.3.2 Directory Table

As mentioned in Section-2.1.2, the data block for a directory contains a table structure that consists of two columns containing the inode numbers and names of the subfiles and subdirectories contained within that directory. For a user accessing the contents of the directory, the inode number corresponding to the name of desired subfile or subdirectory is looked up and the metadata object for that inode number is obtained. From that metadata, the data block of subfile or subdirectory can then be accessed as described above.

Consistent with these semantics of a *directory-table leading to metadata of subfiles/directories*, we add two new columns to the directory table structure, containing the MEK and MVK for the subfiles/directories. Thus, now the directory table not only provides information about how to obtain the metadata object for subfiles/directories, but also provides the keys to decrypt/verify that metadata object. Figure-9 shows the modified directory table structure.

[3]Similar treatment is required for *last-modify-time*. The `nlinks` attribute is also removed since any user with the metadata `read` permission can create a link to that object, so we do not want to allow that user to modify the metadata structure.

		+	+
inode#	name	MEK	MVK
1001	file-a	[file-a-MEK]	[file-a-MVK]
...	...	...	...

Figure 9: xACCESS Directory Table Structure

Note that fields containing keys in metadata and directory-table structures (new fields denoted by '+' in Figure-8 and Figure-9) are not always accessible to all users. In fact, their selective accessibility is what accomplishes access control over this data. This selective accessibility is enforced according to the access control model being supported using a novel concept called Cryptographic Access control Primitive (CAP). In the next section, we describe CAPs for the UNIX access control model.

2.4 Cryptographic Access Control Primitives

A Cryptographic Access control Primitive (CAP) for a filesystem object tries to replicate a particular access control setting (for example, a read-only permission) in the storage-as-a-service model by manipulating accessibility of keys in metadata and directory-table structures and using cryptographic schemes when required. As an example, to support a read-only permission for a file, its CAP ensures that the DEK and DVK fields are accessible in the metadata structure, but not the DSK (possession of data signing key allows `writes`). In some cases, these fields may be further encrypted to achieve access control objectives. Below, we describe the design of CAPs for UNIX directories and files. We will also provide a complete example later in Section-2.4.3.

2.4.1 UNIX Directory CAPs

In the UNIX access control model [148], a directory can have three kinds of permissions – (a) `read`: allows listing the contents of the directory (that is, the command "`ls`" [107]), (b) `write`: allows adding/deleting contents of the directory (equivalent to modifying the directory-table structure) and (c) `eXecute`: allows traversal of the directory and accessing

its contents.

Figure-10 shows the CAPs for directories. The left column shows the permission being designed, the middle column shows the keys fields of metadata and right column shows the data blocks (that is, directory table). To support `zero` permissions, the metadata object has all fields inaccessible (denoted by dark shade)[4]. Consequently data block for the directory (that is, the directory-table structure) is inaccessible (since DEK contained within the metadata is inaccessible). `Read-only` permissions on a directory allow listing its content, but neither modification nor traversal. For this permission, the CAP design is to make the DEK and DVK accessible in the metadata. Now, using DEK the data block for this directory (containing the directory-table) can be decrypted. Further, only the "*name*" column is accessible in the directory-table structure. This implies that a user can only obtain the names of the contents of this directory and not the inode numbers or keys - the equivalent semantics of the `read` permission.

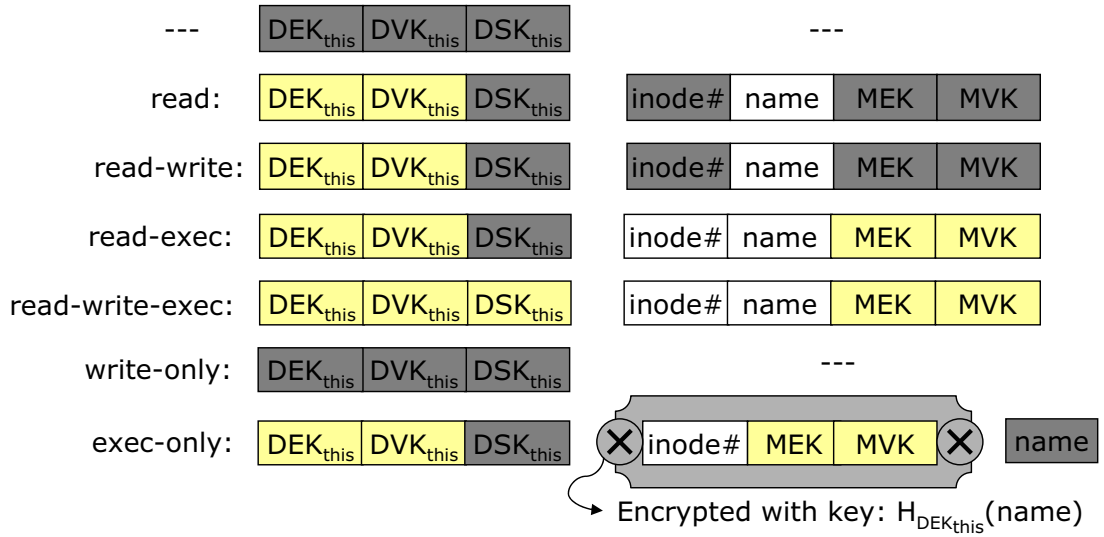


Figure 10: UNIX Directory CAPs

For UNIX, the `read-write` directory permission has the same semantics as `read` since `write` does not work without an `execute` permission. As a result, its CAP design is the same as `read`. With a `read-exec` permission, a user is allowed to traverse the directory and

[4]The subscript "*this*" in the figure indicates that the keys are for the current directory, different from the keys contained in its directory-table structure, which are for the subfiles/directories of this directory.

access its contents, but modification is not allowed. The CAP design for this permission is to make both DEK and DVK accessible in the metadata structure (thus allowing decryption of the directory-table). Within the directory-table, all four columns are accessible since with the `exec` permission, users are allowed traversal and access to the metadata of all subfiles/subdirectories. Using the inode number, users can then obtain the metadata objects of a subfile and using the MEK decrypt that metadata object (and verify with MVK). With a `read-write-exec` permission, users can also modify the directory-table (add/delete contents) and to allow that, the DSK field in the metadata is also made accessible. Next, the `write-only` permission has the same semantics as `zero` permissions since `write` for directories does not work without `exec`; therefore, its CAP is the same as having no permissions.

The most interesting CAP design is for the `exec-only` permissions. The semantics of the UNIX `exec-only` permissions are that users can *not* list the contents of the directory, but can traverse it and access subfiles/directories if they *know* their names. In other words, a user can not do an "`ls`" on the directory, but can "`cd`" into it and access contents by using their exact name. This is a widely used permission in UNIX systems and our study at two large organizations showed that greater than 70% of users use `exec-only` permissions on directories [164]. To support this permission in xACCESS, the directory-table structure requires further manipulation using cryptographic primitives. We accomplish this in the following manner.

First note that since users are allowed to traverse the directory, we have to provide access to the directory-table. In order to do so, the DEK and DVK field in the metadata are made accessible. Next, since a user is not allowed to list the contents, the *name* column in the directory-table is made inaccessible. Finally, a user is allowed to lookup the metadata of a subfile/directory if he/she knows the name. This is accomplished by encrypting the inode number, MEK and MVK fields *row-wise* with new keys derived from the name of the subfile/directory. This new key is derived by using a keyed hash function like MD5 [149] or SHA1 [53] with DEK_{this} as the key and taking the hash of the name. These hash functions are secure hash functions ensuring that it is highly unlikely that two different names will

hash to a same value. Now, any user who knows the name of the subfile/directory can derive this new key and then use this key to decrypt the appropriate row in the directory-table, thus getting access to the metadata of the subfile/directory. This provides equivalent `exec-only` semantics in the storage-as-a-service model.

One UNIX permission not supported in xACCESS is the `write-exec` permission due to the use of symmetric keys for encrypting data blocks. Any user who has `write` permissions, and so has the encryption key, can decrypt the data blocks using the same key and thus can read its contents. However, this permission setting is extremely rare in real systems; in fact, our study of two real enterprise UNIX systems actually found no directory with this permission. As part of our future work, we are looking at using asymmetric mechanisms for supporting this permission. Next, we describe the CAPs for UNIX files.

2.4.2 UNIX File CAPs

In the UNIX access control model, files also have three permissions – (a) `read`: allows reading the content of a file, (b) `write`: allows modifying the content and (c) `execute`: allows running the file as a program. Figure-11 shows the design of the CAPs for UNIX files. In case of files, data blocks are not used in access control design and thus have been omitted from the figure.

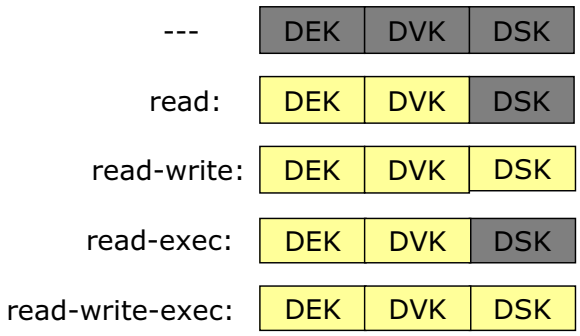


Figure 11: UNIX File CAPs

For `zero` permissions, all key fields in the metadata are inaccessible. For `read` permissions, the DEK and DVK are accessible which will allow decryption and verification of the data block. `Read-write` permission is supported by making the DSK accessible as well. The `read-exec` permission has the same semantics as `read` since once the file has been

decrypted the client filesystem can execute it as a program. As a result it has the same CAP design as `read`. Similarly, the `read-write-exec` has the same CAP as `read-write`.

Similar to directories, we cannot support `write-only` permissions because of the fact that we use symmetric keys to encrypt data. Also, no storage-as-a-service model can enforce `exec-only` permissions for a file since it would imply that the file can be executed as a program without decrypting (equivalent to reading) it.

Using these file and directory CAPs, we can support different permissions individually on SSP-stored files and directories. Next, we describe a complete example that illustrates the design for a directory structure with both files and directories.

2.4.3 CAPs Example

Using our earlier filesystem example of Figure-6, we illustrate the complete design of meta-data and other data structures for supporting UNIX permissions in the storage-as-a-service model. Figure-12 shows the complete design with the directory structure and permissions being supported in the top left corner.

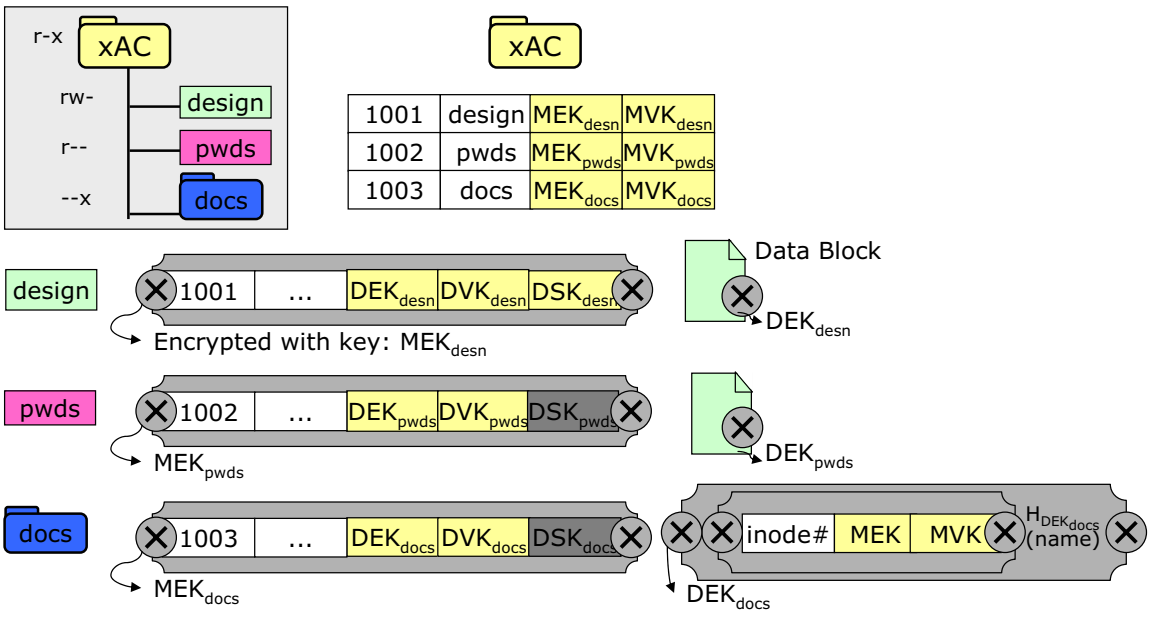


Figure 12: CAPs Example

First, to support `read-exec` permission on the xAC directory, according to the CAP

design for this permission (Section-2.4.1), all four columns of its directory-table are accessible. These columns contain the inode numbers, names, MEK and MVK for the contents of xAC, namely files *design* and *pwds* and directory *docs*.

To access the file *design* with `read-write` permissions, the user would obtain the encrypted metadata object from the SSP using its inode number, decrypt it using MEK_{desn} and verify it with MVK_{desn} (inode number and keys are obtained from the xAC directory-table). From the metadata of the file, the user can obtain the DEK_{desn} to read the file, DVK_{desn} to verify it and DSK_{desn} to write to it.

File *pwds* has a similar structure except that the DSK_{pwds} is inaccessible – the CAP for `read-only` permission (Section-2.4.2).

For directory *docs* with `exec-only` permission, first its metadata object can be decrypted using MEK_{docs} available from the directory-table of xAC. From the decrypted metadata object, DEK_{docs} is obtained which is used to decrypt the directory-table structure. Internally, the directory-table structure is built using the `exec-only` CAP design described earlier, that is, the name column is inaccessible and the inode number, MEK, MVK fields for subfiles/directories are encrypted using keys derived from their names. A user with knowledge of the name can derive the key and decrypt the structure.

This completes the example and demonstrates how the CAP based design is able to support an expressive UNIX-like access control model. Later in Section-2.5.1, we will describe specifically how different filesystem operations (like `stat`, `mkdir`) are implemented in our prototype.

Note: Observe that our key distribution mechanism exploits the hierarchy of the file system. For example, to access a file or directory, its metadata keys are obtained from its parent directory's directory-table. The parent directory would have been accessed by obtaining keys from the *grand-parent* directory and so on. This will lead all the way up to the namespace root (for example, "/"). Now the question remains – how do users obtain access to this root element?

In traditional filesystems, the metadata for the root is contained with a data structure called the *superblock* [30]. We describe the superblock structure for xACCESS next.

2.4.4 Filesystem Superblock

A superblock contains description of the basic structure and attributes of the filesystem like number of free blocks, number of free inodes and importantly the inode number of the first inode in the filesystem, that is, the namespace root ("/" for ext2). In case of xACCESS, along with the inode number, we also store the MEK_{root} and MVK_{root} that allow decrypting the metadata for the filesystem namespace root directory.

We could potentially distribute this superblock (with root encryption keys) out-of-band to authorized users. However, we can accomplish its distribution using completely in-band mechanisms by using the following technique. For each authorized user u, we store the superblock encrypted with the public key of u (B_u) and store it at the SSP. That is, we store $E_{B_u}\{\text{Superblock}\}$ for all authorized users of the filesystem. Now, when a user mounts the filesystem, he/she decrypts the superblock using private key P_u and obtains access to the metadata structure of the namespace root. This way, no out-of-band distribution is required and only a one-time public key cryptographic operation is required (at mount time).

2.4.5 Multiple CAPs per Object

So far, we have described how a user can mount a filesystem by decrypting the superblock and then access the filesystem using hierarchical CAPs based design. However, different users will have different access rights to filesystem objects, for example, users `alice` and `bob` have different access rights to the directory `docs` in Figure-6. We need to devise mechanisms such that `alice` can access her CAPs (with `read-write-exec` permissions), whereas `bob` only gets access to a CAP with *exec-only* permissions.

We have developed two schemes to allow different users access to different CAPs for the same filesystem object. These schemes differ in the amount of storage overheads, update costs and metadata access costs.

2.4.5.1 Scheme-1: Different Metadata Structures

The first scheme separates metadata structures for different users, that is, the filesystem tree structure (metadata and directory-table components) is replicated for each user with CAP design based on the access permissions for that particular user. For example, `alice` will have her own metadata objects starting from the namespace root to all files that she can access, with intermediate directory-table objects containing keys that access appropriate CAPs for `alice`. User `bob` will have a similar separate filesystem tree.

Clearly, this scheme has additional storage overheads. Based on our prototype implementation, for a filesystem with one million files, it will cost nearly \$0.60 per user per month according to storage prices of the Amazon S3 [6] storage service. Additionally, this scheme has update overheads, since whenever a new object is created or existing metadata object modified, updates need to be made to the filesystem tree of each user that can access that object. As a result this scheme is more suitable for scenarios when writes to metadata objects are infrequent.

It is worthwhile to note that most related work uses public key cryptography for metadata [67, 106, 2] which is equivalent to this scheme since every metadata object is separately encrypted with the public keys of all users that can access that object, thus replicating it for all such users.

Next, we describe another scheme in which users can share CAPs if they have similar access rights to objects.

2.4.5.2 Scheme-2: Sharing of CAPs

The second scheme avoids replicating metadata structures by observing that number of CAPs per object is typically a much smaller number than the number of users that can access that object. For example, for our access control model described above, there are only five unique CAPs per directory and four per file. Using this observation, this scheme replicates metadata structures only for the number of CAPs that are required to be supported and includes indirection from users' metadata and directory-table structures to point to the correct CAP for their individual permissions. Figure-13 shows this scheme for our running

example of Figure-6 with users `bob` and `john` able to share CAPs since they have same access permissions on those objects. This results in low storage and update overheads.

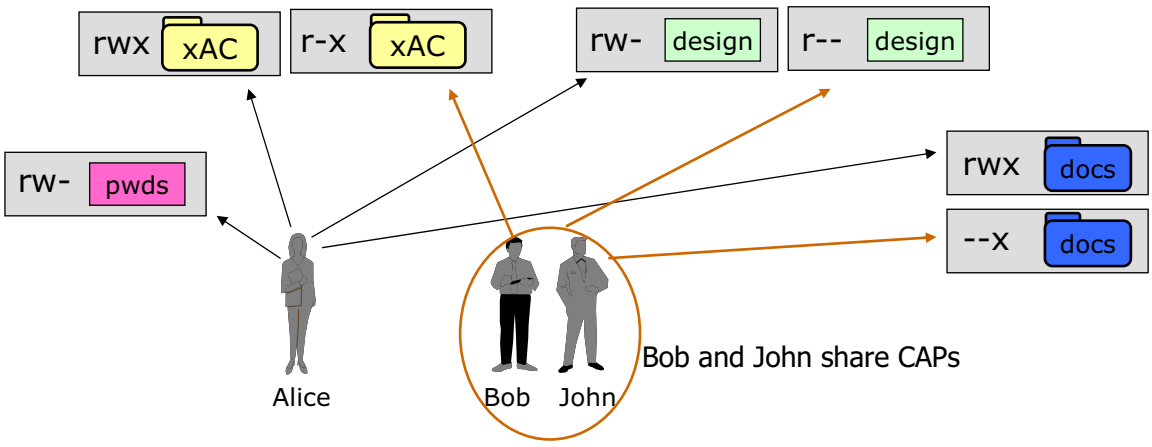


Figure 13: Handling multiple users by sharing CAPs

It is possible that users that share CAPs for a certain part of the directory structure may *split* at a certain point, that is their permissions diverge at a certain file/directory. One typical cause of this divergence is POSIX ACLs [72] when permissions for specific users or groups are added to the traditional UNIX {*owner, group, others*} model [148]. It is important to note that the total number of such splits is a small number, as they typically occur at a higher level directory (for example, "*/home/*") and children directories later inherit permissions from the parent directory. Thus, once a split occurs, users continue to use their separate CAPs. In order to accommodate these few split points, we use public key cryptography technique, similar to the one used for the superblock. More specifically, the metadata structures for split-point objects are encrypted with the public keys of users that can access it and internally these metadata structures point to the specific CAPs as dictated by the new access control permissions. Thus, Scheme-2 offers a tradeoff of reduced storage and updated costs at slightly higher access costs.

2.4.6 Design Summary and Overheads

In this section, we summarize some of the key points in the xACCESS design and concretely describe overheads of providing such expressive access control over the decentralized storage-as-a-service model.

- xACCESS provides in-band key management (users only manage their public-private key pair). We accomplish this by exploiting the hierarchy of filesystems and storing data encryption keys in metadata objects and metadata encryption keys in parent directory's directory-table structures.

- xACCESS predominantly uses symmetric key cryptography for metadata objects. Only the namespace root and few *split-points* are accessed using public key cryptography. As we show in Section-2.9, use of symmetric key cryptography helps outperform comparable approaches by over 40% on a number of benchmarks.

- xACCESS is also able to support an expressive UNIX-like access control model through the novel use of cryptographic access control primitives.

Next, we concretely discuss various overheads in xACCESS and other similar storage-as-a-service access control systems.

- *Encryption Overheads*: A common overhead for any encryption based access control system like xACCESS is the cost of encrypting data *and* metadata. All systems [94, 67, 119, 76, 2] store encrypted data at the SSP, but differ in the choice of cryptographic operations used. While symmetric key cryptography is generally used for *data* encryption, only xACCESS, Plutus [94] and CNFS [76] use symmetric key cryptography for metadata as well, with most other systems [67, 119, 2] using expensive public key cryptography.

- *Key Management Overheads*: Another overhead with encryption based storage-as-a-service models is the cost of managing keys. For example, Plutus [94] and CNFS [76] requires owners to email keys to users that are authorized to access their files. Such out-of-band key management has additional overheads with significant user involvement required for every metadata or permissions update. In contrast, xACCESS's in-band key management functionality drastically reduces these management costs.

- *Key Revocation and Re-encryption Overheads*: Another important overhead occurs due to revocation of a user's permissions (for example, removing a user's `read` permission). Such revocations require re-encryption of files as the revoked user could have cached the data encryption key (DEK) for that file. This overhead is also common to all encryption based access control systems and exists due to the access control embedding methodology. In traditional local storage systems, such revocation is handled through the trusted access control engine like an OS or Kerberos based key distribution center's revocation lists. We discuss this issue further in Section-2.5.

- *Access Control Expressiveness Overheads*: Since all other proposed systems provide only minimal `read` and `write` permission settings, one overhead unique to xACCESS design is its cryptographic access control primitives based operations. Such CAPs based design incurs costs in order to provide expressive access control semantics over the decentralized storage-as-a-service model. For example, the design of the `exec-only` CAP for UNIX directories requires an additional row-wise encryption of the directory-table structure in addition to the core data and metadata encryption. Secondly, creating multiple CAPs for various permission settings that need to be supported (Scheme-2 above) requires multiple metadata objects to be created.

Next, we describe the architecture and implementation details for xACCESS and later experimentally evaluate its overheads in comparison to other approaches in Section-2.9.1.

2.5 Architecture and Implementation

The xACCESS system is composed of three main components, as shown in Figure-14.

- **Migration Tool**: This component is responsible for the initial setup and migration of data from local storage to the storage-as-a-service model. It can perform more efficient bulk data transfers (by using compression and other optimization techniques) and create the cryptographic infrastructure, if required (that is, generating user and group keys).

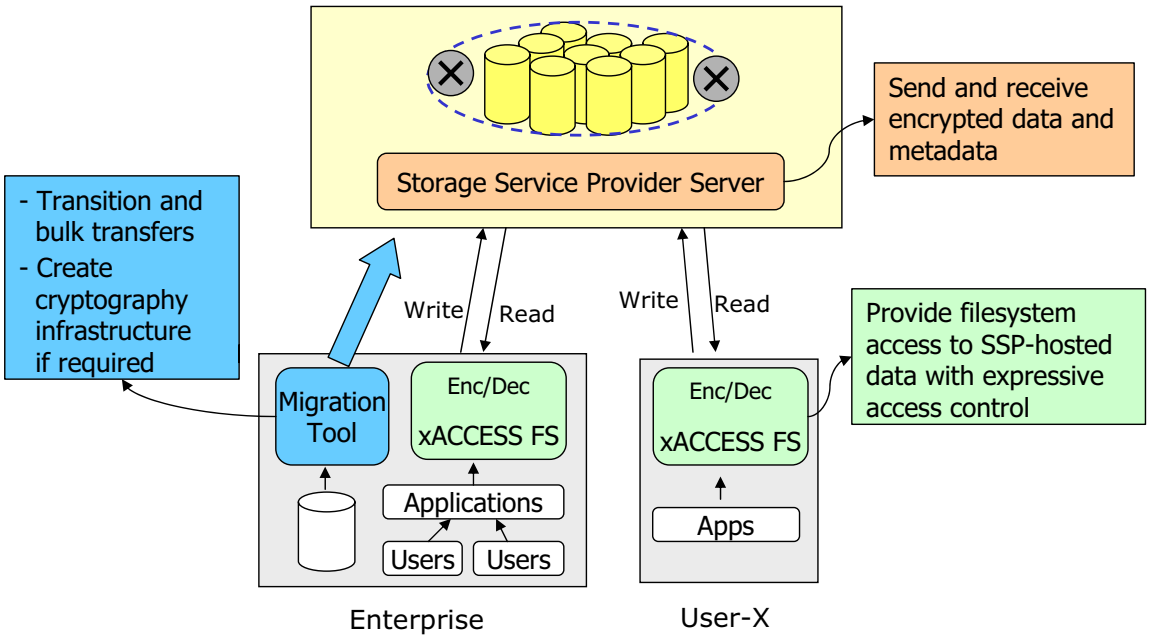


Figure 14: xACCESS Architecture

- **SSP Server**: The SSP server receives data from the migration tool (or client writes) and is responsible for serving data/metadata requests from all clients. There is no computation involved on the data at the SSP and it simply maintains a large hashtable for encrypted metadata objects and encrypted data blocks, both indexed by the inode numbers and either hash of user/group ID (for Scheme-1 above) or CAP ID (Scheme-2).

- **xACCESS Filesystem**: The most important component in the architecture is the xACCESS filesystem that will be installed at every client accessing data from the SSP. It provides filesystem access to the data stored at the SSP and performs all cryptographic operations to serve client requests. We discuss the architecture of the filesystem in greater detail next.

2.5.1 xACCESS Filesystem

The xACCESS filesystem provides a regular filesystem-like access over remotely stored SSP data to the clients. It is this component that is responsible for navigating through the cryptographic CAPs based design and encryption/decryption of metadata and data blocks.

To the user, it will appear as a regular filesystem. Figure-15 shows the architecture of our prototype filesystem.

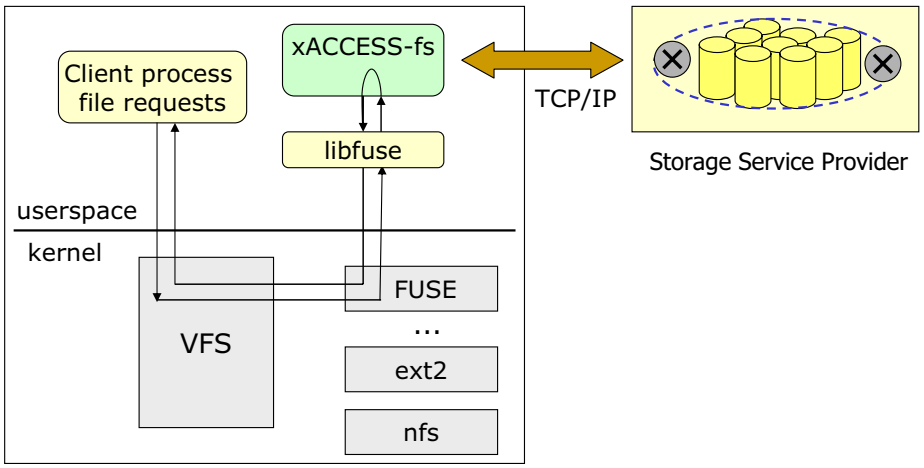


Figure 15: Architecture of xACCESS Filesystem

We developed the xACCESS filesystem in userspace using the FUSE [173] library for Linux. FUSE consists of a kernel module that *acts* as a filesystem to the Linux Virtual Filesystem Layer (VFS). On receiving filesystem requests, this kernel module passed those requests to a userspace library (*libfuse*), on which the xACCESS filesystem is based. Even though compared to kernel filesystems, there are additional kernel-userspace context switches, these costs are not significant especially since most of our data access goes over a wide area network. Also, we use TCP/IP sockets for the filesystem communication with the SSP.

To mount this filesystem, the client would access the superblock for the user mounting that filesystem and decrypt it using the user's private key (stored at a standard directory location). On decryption, the filesystem obtains the inode number, MEK and MVK for the namespace root. Next, it obtains the metadata for that namespace root element which is the same as the traditional `getattr` filesystem command. We describe the implementation of various such filesystem commands below.

2.5.1.1 *Filesystem Operations Implementation*

All POSIX compliant filesystems implement a number of filesystem operations like `getattr`, `mkdir`, `read`, `close`. For the xACCESS filesystem, these function implementations are required to handle communication with the SSP over the network and perform all encryption and decryption operations for both metadata and data blocks. Figure-16 shows how some of these functions are implemented within the xACCESS filesystem. The last column also shows different components of cost involved in implementing the function.

OP	PROCESSING	COSTS	
		Crypto	Network
getattr	obtain metadata and decrypt	1-md-dec	metadata recv
mkdir mknod [*]	create new file/dir; encrypt it; modify parent directory; encrypt it; send both to server	1-md-enc 1-parent-dir-enc	metadata send; parent-dir send
chmod [*]	modify metadata, encrypt it and send to server	1-md-enc	metadata send
read	obtain data and decrypt	1-data-decrypt	data recv
write	write into local cache		
close	encrypt file; send to server	1-data-encrypt	data send

[*] per required CAP

Figure 16: xACCESS Filesystem Operations

First, the `getattr` function is executed in response to a `stat` request [107]. This function is required to return various attributes like owner, group and permissions for the filesystem object being `stat`'ed. In xACCESS, it implies obtaining the encrypted metadata object from the SSP and decrypting it to obtain the attributes. Thus, it has the costs of one network receive of encrypted metadata and one symmetric decryption.

The `mkdir` function is executed when creating a new directory. In xACCESS, this implies creating a new metadata object, encrypting it with a unique symmetric key, modifying the parent directory's directory-table to include the new directory and re-encrypting the modified table. Finally the encrypted metadata and parent directory's directory-table are

sent to the SSP. It is important to note that multiple metadata objects and parent directory-table modifications are required (one for each supported CAP, if using Scheme-2 described in Section-2.4.5). Intuitively, these added costs are due to the fact that we are not only creating new data, but also embedding access control into it. The `mknod` function is similar in design, except that a new file is created and not a directory.

The `chmod` function is used to change permissions for a file or directory. For the xACCESS filesystem, this could simply require creating a new CAP and modifying metadata keys to point to the new CAP (for example, changing the permissions for user from `zero` to `read`). However, in scenarios when permissions are revoked, it could require re-encryption of files. For example, changing the permissions of a user from `read` to `zero`; since the user could have cached the encryption key and later try to access file data using that key, it is required that the file is re-encrypted using a new key. In related research on this topic, systems support one of the two revocation schemes – (a) immediate revocation [67], in which case a new key is created and file re-encrypted immediately during the `chmod` operation, or (b) lazy revocation [94], in which the file is re-encrypted only when its content is updated. The motivation for the latter is that the user whose permissions have been revoked, could have cached the file when he/she had access to it, thus it is only important to change the keys if and when the file is updated. While the xACCESS design can support both techniques, our prototype currently uses immediate revocation.

For data I/O functions, a file `read` obtains the encrypted data block for the file and decrypts it. In our current implementation, we cache all `write`s locally and only encrypt the file before sending it to the SSP as the result of a file `close`.

This completes the xACCESS filesystem implementation design. Next, we take a closer look at its data sharing mechanisms from a privacy angle.

2.6 Privacy Analysis for Data Sharing

The xACCESS filesystem provides a UNIX-like access control model over SSP-stored data and using this model, users can share their data with other users of the system. In fact, one of the important advantages of the storage-as-a-service model is that it encourages this

sharing of data between geographically distributed users as it eliminates the need of setting up file servers. In this section, we take a critical look at our access control model and analyze the *privacy* support in its data sharing mechanisms. For example, how does the system assist a user to share data only with desired users and prevent private information from being leaked to unauthorized users?

In order to successfully do this, we also take a look at the *convenience* of using data sharing mechanisms in typical situations. This is due to the fact that lack of convenience leads to users compromising (intentionally or mistakenly) their security requirements to conveniently fit the specifications of the underlying access control model. Please note that we use the seemingly oxymoronic phrase "*private sharing*" to indicate the desire of sharing data only with a select set of authorized users.

As part of our analysis, we looked at how users use the access control mechanisms for their data sharing needs in practice. We conducted experiments at two *nix installations (SunOS and HP-UX that follow UNIX access control model like xACCESS) of a few hundred computer-literate users each. Surprisingly, we found that large chunks of private data was accessible to unauthorized users. At one organization of 836 users, over 84 GB of data was accessible, including more than 300,000 emails and 579 passwords to websites like *bankofamerica.com* and medical insurance record websites. The reason for this surprisingly large privacy breach without exploiting technical vulnerabilities like buffer overflows or gaining elevated privileges, is the combination of lack of system support and user or even applications' privacy-indifferent behavior either mistakenly or for lack of anything better. Later in Section-2.7, we describe enhancements to the access control model that provide more secure and convenient data sharing.

The attack used in this work is a form of an **insider** attack, in which the attacker is inside the organization. The attacker could be a disgruntled employee, contractor or simply a curious employee trying to access the salaries chart in the boss's home directory. According to a study by the US Secret Service and CERT [31], such attacks are on a rise with 29% of the surveyed companies reporting[5] having experienced an insider attack in a

[5]It is believed that such attacks are usually much under-reported for lack of concrete evidence or fear of

single year [29]. Also, in complete congruence to our attack, the report finds that:

> "`Most incidents were not technically sophisticated or complex - that is they typically involve exploitation of non-technical vulnerabilities`"

Next, we discuss various issues that lead to privacy breaches and present the results of our case studies. This analysis will provide us insights into desirable privacy and convenience characteristics.

2.6.1 Privacy Breaches

In this section, we discuss various privacy breaches that occur in *nix systems. The discussion below primarily explores the popular {*owner, group, others*} *nix paradigm [148]. The advanced mechanisms like POSIX ACLs [72] enhance privacy in only a few cases and we will demonstrate a clear need of a new, more complete solution.

2.6.1.1 Selective Data Sharing

The first kind of privacy breach occurs due to the need to "*selectively*" share data. The selectivity can be of two kinds:

- **Data Selectivity**: Data selectivity is when a user wants to share only a few (say one) of the subdirectories in the home directory. So, an authorized user is allowed to access only the shared subdirectory, but not any of the sibling directories. In order to do this correctly, the owner needs to follow two steps - (a) set appropriate permissions to the shared subdirectory (at least execute permissions on the entire path to the subdirectory and the sharing permissions on the subdirectory), and (b) *remove permissions from the sibling subdirectories.* The second step is unintuitive, since the user needs to act on secondary objects that are not the focus of the transaction. Also, any new file being created needs to be protected.

- **User Selectivity**: In many situations, users need to share data with an adhoc set of users that do not belong to a single user group or are only a subset of a user group,

negative publicity [177]

and are not the entire *others* set. In this situation, the permissions for *group* or *others* are not sufficient. Also, creating a new user group requires administrative assistance which is not feasible in all cases.

In order to do selective data sharing, currently owners mostly use execute-only permissions on the home directories. The perception is that since users can not list the contents of the directory, they cannot go any further than traversing into the home directory unless they know the exact name of the subdirectory. Now, the owner can authorize desired users by giving them the names of the appropriate subdirectories. Those *authorized* users can traverse into the home directory and then use the subdirectory name to "`cd`" into it (without having to list the contents of the home directory). From data selectivity perspective, it is assumed that they cannot access the rest of the contents and from user selectivity perspective, unauthorized users cannot access any contents.

However, the underlying system cannot distinguish between such authorized or unauthorized users. Any user who can **guess** the subdirectory name can actually access the data. For an attacker inside the organization, even without a dictionary attack, this is not a herculean task. For example, for a computer science graduate school, it is highly likely that users will have directories named *research*, *classes* or *thesis*. An easy way of creating such a list of names is by collecting names from users that actually have *read* permissions on the home directories. Within the context of a single organization, or in general human psychology, it is likely that many users have similar directory names. This is essentially a form of *social engineering* [120] in which users and not systems are manipulated to reveal confidential information[6]. Of course, one simple solution is to use cryptic directory names unlikely to be guessed (security-by-obfuscation). This is inconvenient as it impacts the owner's semantics of file names.

Secondly, many times directory names do not need to be guessed at all. The names can be extracted from **history** files (like *.history* or *.bash_history*), that contain the commands last executed by the owner, like `cd`, which will include real directory names. In fact, in

[6]A well-know hacker, Kevin Mitnick said "*... social engineering was extremely effective in reaching my goals without resorting to using a technical exploit ...*" [167]

our experiments we found around 20-30% of all users had readable history files and around 40% of the total leaked data was obtained from the directory names extracted from these history files.

Thirdly, it is not always user created directories that leak information. Many **applications** use standard directory names and fail to protect critical information. For example, the Mozilla web browser [122] stores the profile directory in `~/.mozilla` and had that directory world-readable [123] in many cases, till as late as 2003. Many *nix installations with the browser installed before that have this vulnerability and we were able to obtain 579 password to financial and private websites (because users saved passwords without encrypting them). In addition, their browser caches, bookmarks, cookies and histories were also available. The browser Opera [137] also has a similar vulnerability, though to a lesser extent. While it can be argued that it is the responsibility of application developers to ensure that this does not happen, we believe that the underlying system can assist users and applications in a more proactive manner.

The POSIX ACLs [72], if used help in achieving only user selectivity. They do not address the data selectivity requirements or prevent leaking of application data.

2.6.1.2 Metadata Privacy

So far, we have only talked about the privacy breach for file *data*. However, there are many situations in which users are interested in protecting even the metadata of the files. The metadata contains information like ownership, access time, update time, creation time and file sizes. There are scenarios where a user might obtain confidential information by just looking at the metadata. For example, an employee might be interested in knowing how big is his annual review letter or did the boss update it after the argument he had with her?

The *nix access control does not provide good metadata privacy. Even if users only have execute permissions on a directory, as long as they can guess the name of the contained file, its metadata can be accessed even if the file itself does not have any read, write or execute permissions on it. Thus, if a user has to share even a single file/directory within the home directory (thus, requiring atleast execute permissions), all other files contained in the home

directory have lost their metadata privacy.

Of course, a solution is to put such files in a separate directory and protect them. However, in many cases these files might be accessed by standardized applications making it infeasible to move (for example, how to protect metadata privacy of shell initialization files (*.profile*), or history files, which are always created in the home directory). Also, from our experience, many users like to keep their active files in the home directory itself.

Again, lack of convenient support from the system leads to privacy breaches, in this case leaking file metadata.

2.6.1.3 Data Sharing Convenience

User convenience is an important feature of an access control implementation. If users find it tough to implement their security requirements, they are likely to compromise the security requirements to easily fit the underlying access control model. From our analysis of the *nix access control, along with some of the issues discussed earlier, we found the following two data sharing scenarios in which there is no convenient support for privacy.

- **Sharing a Deep-Rooted Directory**: For a user to share a directory that is multiple levels in depth from the home directory, there needs to be at least execute permissions on all directories in the path. This in itself (a) leaks the path information, (b) puts sibling directories at risk and (c) leaks metadata information for sibling directories. In order to prevent this, since most operating systems do not allow hard links to directories anymore, a user would have to create a new copy of the data, which raises consistency issues. Also, since users are more careless with permissions for deep rooted directories (they protect a higher level directory and that automatically protects children directories), a copy of such a directory could have privacy-compromising permissions.

- **Representation of Shared Data**: In many circumstances the way one user represents data might not be the most suitable way for another user. For example, while an employee might keep his resume in a directory named "*job-search*", it is clearly not the most apt name to share with his boss. The employee might want her to see the

directory simply as "*CV*". Changing the name to meet the needs of other users is not an ideal solution. This again shows the lack of adequate system support for private and convenient data sharing.

It is important to recognize that even an extremely privacy-conscious user can not protect data at all times. Exhaustive user efforts to maintain appropriate permissions on all user and **application created** data will still be insufficient to protect metadata privacy or allow private sharing of deep rooted directories with user-specific representation.

In the next subsection we present actual results from two *nix installations to see if the *nix shortcomings actually lead to privacy breaches in real situations.

2.6.2 Case Studies

As part of our study, we conducted experiments at two geographically and organizationally distinct *nix installations. Users at both installations (CS graduate schools) are highly computer literate and can be expected to be familiar with all available access control tools.

For our analysis, we consider the following data to be private:

- All user emails are considered private.
- All data under an execute-only home directory is considered private.
- Browser profile data (including saved passwords, caches, browsing history, cookies) is considered private.

The second assumption above merits further justification. It can be argued that not every subdirectory under an execute-only home directory is meant to be private (for example, a directory named *public*). However, we believe our definition to be a practical one. The semantics of the execute-only permission set dictate that any user other than the owner cannot list the contents of the directory and since the owner never *broadcasts* the names of the shared directories, an unauthorized user *should not* be able to access that data. And since we do not include in our measurements any obviously-private data from home directories of users with *read* permissions (for example, world-readable directories named *personal*,

or *private*), we believe the two effects to approximately cancel out.

2.6.2.1 Attack Methodology

Next, we describe the design of our attack[7] that scans user directories and measures the amount of private data accessible to unauthorized users. This discussion is also important since the design is eventually used to develop an auditing tool discussed later.

The attack works in multiple phases. The first step is to obtain directory name lists which can be tried against users with execute-only home directories. Three strategies are used to obtain these lists:

- *Static Lists*: These are manually entered names of directories likely to be found in the context of the organizations - CS graduate schools. For example, "*research*", "*classes*", "*papers*", "*private*" and their variants in case ("*Research*") or abbreviations ("*pvt*").

- *Global Lists*: These lists are generated by obtaining the directory names from home directories of users that have *read* permissions.

- *History Lists*: These are user specific lists generated by parsing users' history files, if readable. We used a simple mechanism, parsing only `cd` commands with directory names. It is possible to do more by parsing text editor commands (like `vim`) or copy/move commands.

In the next step the tool starts a multi-threaded scanning operation that attempts to scan each user directory. For users with **no** permissions, no scanning is possible. For users with **read-and-exec** permissions, as discussed earlier, since there is no precise way of guessing which data would be private, we only measure email and browser profile statistics. Finally, for users with **execute-only** permissions, along with email and browser profile statistics, we also attempt to extract as much data as possible using the directory name lists prepared in the first step.

[7] We took precautions to ensure that our study does not violate user privacy by collecting only aggregate statistics and randomizing order of scans.

Evaluating Email Statistics

This is done by attempting to read data from standard mailbox names - "*mail*", "*Mail*", "*mbox*" in the home directory and the mail inboxes in */var/mail/userName*. A `grep` [107] like tool is used to measure (a) number of readable emails, (b) number of times the word "*password*" or its variants appeared in the emails.

Evaluating Execute-only Data Statistics

For users with execute-only permissions on the home directory, the scanner uses the combination of static, global and the user's history lists to access possible subdirectories. Double counting is avoided by ensuring that a name appearing in more than one list is accounted for only once and by not traversing any symbolic links. While scanning the files, counts are obtained for the total number of files and the total size of the data that could be accessed.

Evaluating Browser Statistics

The mozilla browser [122] stores user profiles in the `~/.mozilla` directory. This directory used to be world-readable till as late as 2003 when the bug was corrected [123]. Within that profile directory, there are subdirectories for each profile that has been used by that user. The default profile is usually named "*default*" or "*Default User*". So even in case the .mozilla directory had execute-only permissions, it is possible to access default profile directories (unless a user specifically removed permissions). Within the profile directory, there is another directory with a randomized name ending in "*.slt*"[8]. Since the parent directories had *read* permissions, the randomization provides no security and the name is visible. Within this directory, the following files exist and (with this bug) were readable:

- **Password Database:** A file with the name of type "*12345678.s*". This contains user logins and passwords saved by mozilla when the user chooses to save them. Ideally, users should use a cryptographic master key to encrypt these passwords, but as our results will show many users do not encrypt their passwords. For such cases, mozilla

[8]Please see [124] for complete profile directory contents and their location.

stores the passwords in a base-64 encoding (indicated by the line starting with a ~ in the passwords file), which can be trivially decoded to get plaintext passwords.

- **Cookies:** The `cookies.txt` file contains all browser cookies. Many websites including popular email services like Gmail [71], Hotmail [118] allow users to automatically login by keeping their usernames and passwords (encrypted) in the cookies file. Hijacking these cookies can allow a malicious user to login into these accounts. For many other cookies related attacks, see [166].

- **Cache:** This is a subdirectory that contains the cached web pages visited by the user.

- **History Database:** Web surfing history, which many sophisticated viruses and spyware invest resources to collect, are also readable.

- **Forms Database:** Mozilla allows users to save their form data, stored in a file of type "*23456789.w*", that can be automatically filled. This could include credit card numbers, social security numbers and other potentially sensitive information. Here again, users should use a master key to encrypt this information.

2.6.2.2 Results

The complete characteristics of the two organizations are shown in Table-3, where *# ReadX* is the number of users with read and execute permissions to their home directories, *# NoPerms* are users with no permissions and *# X-only* are the users with only execute permissions. Both organizations are computer science graduate schools at two different geographical locations within the United States. At both the organizations, a significant number of users (68% and 77%) used execute-only permissions on their home directories.

Table 3: Case Study Organization Characteristics

Org.	# Users	# ReadX	# NoPerms	# X-only
Org-1	836	198	54	573 (68%)
Org-2	768	136	39	593 (77%)

Table-4 lists the amount of data extracted from execute-only home directories at Organization-1 and 2, where *# Hit Users* is the number of users that leaked private information, *# Hits* is the total number of directory name hits against all X-only users and *# Files* is the number of leaked files and *Data-Size* is the total size of those files

Table 4: Data extracted from X-only home directory permissions

Org.	# Hit Users	# Hits	# Files	Data Size
Org-1	462	2409	983086	82 GB
Org-2	380	911	364932	25 GB

As can be seen, a large fraction of users indeed leaked private information - 55% and 49% of total users respectively. Recall that we do not extract any data from users with *read* permissions on their home directories; so a more useful number is the fraction of X-only users that revealed private information. That number is 80% and 64% respectively. Also, on an average, 2127 files and 177 MB of data is leaked in the first organization for each X-only user and 960 files and 65 MB of data is leaked in the second organization. A partial reason for the lower numbers in the second organization could be the fewer number of users with *read* permissions, which would have impacted the global name lists creation. Overall, we believe this to be a very significant privacy breach.

As mentioned earlier, many times the names of the subdirectories do not need to be guessed and can be obtained from the history files in the user home directories. Table-5 lists the success rate of the attack in exploiting history files, where *# History Hits* is the number of users with readable history files, *# Files* is the number of private files leaked due to directory names obtained from history files and *Data-Size* is the size of the leaked data. As it shows, around 40% of X-only users had readable history files which led to 40-50% of total leaked data in size.

Table 5: Exploiting History Files

Org.	# History Hits	# Files	Data Size
Org-1	253	561254	35 GB
Org-2	237	155826	14 GB

Email Statistics

Table-6 presents the results of the email data extracted from users in both organizations, where *# Folders* is the number of leaked email folders, *# Emails* is the total number of leaked emails. *Size* is the size of leaked data and *# Password* is the number of times the word "*password*" or its variants appeared in the emails. Recall that this data is obtained for both X-only users and the users with *read* permissions on their home directories.

Table 6: Leaked email Statistics

Org.	# Folders	# Emails	Size	# *Password*
Org-1	2509	315919	4.2 GB	6352
Org-2	505	38206	120 MB	237

As can be seen, a large number of emails are accessible to unauthorized users (especially at Organization-1). Also, the number of times the word "password" or its variants appear in these emails is alarming. Even though we understand that some of these occurrences might not be accompanied by actual passwords, by personal experience, distributing passwords via emails is by no means an uncommon event.

Browser Statistics

The second organization did not have the mozilla vulnerability since they had a more recent version of the browser installed, by which time the bug had been corrected. So the results shown in Table-7 have been obtained only from the first organization. Looking at the results, the amount of accessible private information is enormous. Table-8 contains a sample of the websites that had their passwords extractable and clearly most of these websites are extremely sensitive and a privacy breach of this sort is completely unacceptable.

Also as seen from Table-8, some obtained passwords were for accounts in other institutions and a few of them are likely to be *nix systems. Thus, it is conceivable that this password extraction can be used to **expand to other *nix installations** and thus be much more severe in scope than a single installation.

Table 7: Leaked browser Statistics at Organization-1

# Users with accessible `.mozilla`	311
# Users with readable password DB	149
# Passwords Retrievable	579
# Users with readable cookies DB	207
# Cookies Retrievable	19456
# Users with accessible caches	233
# Cached Entries	20907
# Users with readable browsing histories	256
# URLs in History	130,503

Table 8: Sample accounts with retrievable passwords

Financial Websites	**Personal Websites**
www.paypal.com	adultfriendfinder.com
www.ameritrade.com	www.hthstudents.com
www.bankofamerica.com	www.icers911.org
Email Accounts	**Other Institutions**
mail.lycos.com	cvpr.cs.toronto.edu
my.screenname.aol.com	e8.cvl.iis.u-tokyo.ac.jp
webmail.bellsouth.net	systems.cs.colorado.edu

Miscellaneous Statistics

Among few other applications at Organization-1, 17 users had their Opera [137] browser's cookies file readable and 497 users had their email address books, used by the Pine email client [142] and stored in ~`/.addressbook` readable. 18,308 email addresses could be obtained from these address books which can be potentially used for highly targeted spam!

2.6.3 Attack Severity

It is important to highlight the severity of this attack:

- **Low Technical Sophistication**: The attack is extremely low-tech; the commands used in a manual attack would be `cd`, `ls` and such. This aspect makes the threat significantly more dangerous than most other vulnerabilities.

- **Low Detection Possibility**: A version of the attack that targets only a few users a day and thus keeps overall disk activity normal has a very low probability of detection. Typical *nix installations do not keep extensive user activity logs and it is highly likely that such an attack will go unnoticed. Even if an individual user notices an unusual

last-access time on one of the files, without extensive logging, it is impossible to pin point the perpetrator.

- **No Quick Fix**: Unlike other security vulnerabilities like buffer overflows, this attack uses a **design** shortcoming combined with user/application carelessness and no patches would correct this problem overnight.

- **High Success Rate**: It is important to notice that the attack had a high success rate at installations where most users are computer literate. With increasing mainstream penetration of *nix systems, most users in the future would be ordinary users who cannot be expected to fully understand the vulnerabilities. This makes this attack a very potent threat.

2.7 Privacy Enhancements

In this section, we present two solutions that can be used independently or together to facilitate stronger privacy protection in xACCESS and other *nix systems. The first solution is a Privacy Auditing Tool that monitors the privacy health of an organization and can alert users/administrators of potential threats. The second solution is a new access control model, View-Based Access Control, that modifies the data sharing mechanisms for stronger protection. Using the two solutions together provides an excellent data sharing environment.

2.7.1 Privacy Auditing Tool

The aim of the privacy auditing tool is to periodically monitor user home directories and identify potential private data exposures. A similar approach is used by most enterprise security applications like [172] that audit user systems and enforce compliance to security policies, for example, requiring laptop owners to keep a boot-up password, or system administrators to enforce stricter password rules and so on. In a similar vein, the privacy auditing tool will scan user home directories and alert administrator or the users directly if their private data can be accessed by unauthorized users.

The design of such a tool is very similar to the design of the attack described in Section-2.6.2.1. A number of test accounts are created on the monitored system with different

group memberships, since it is possible that some user group might have access to more private data than others. The tool is then run from these test accounts to identify exposed private data. The auditing tool can be used either to obtain only higher level statistics, as described in the attack or more user-specific information which can alert users directly of their *private* directories that can be accessed by unauthorized users. A variant would be allowing users to themselves invoke an audit of their home directory. Yet another variant would be to allow the tool to automatically correct some of the obvious mis-configurations like emails.

Even though this solution does not solve the underlying access control problem, it has the following **advantages**:

- The privacy auditing tool does not require operating system or file system changes and so can be easily incorporated into enterprise infrastructures. The auditing solution is the quickest way of mitigating this vulnerability.
- Since existing security auditing tools operate in a similar mode, this tool can be easily added on as a privacy protection module to such tools.
- Even with a better access control model, the unavoidable and error-prone human involvement in protecting private data makes such a tool an important component of a secure enterprise.

Next, we discuss a more proactive solution to address the shortcomings.

2.7.2 View-Based Access Control (VBAC)

Our second solution is the design of a new access control mechanism called View-Based Access Control (VBAC). Similar to the *nix access control, VBAC is also a discretionary access control model, thus keeping security within the control of the data owner. VBAC is based on following design goals:

- *Act only on primary objects*: Private sharing in *nix in its current form requires a two step approach of (a) sharing desired data, and (b) protecting other unrelated data.

The second step, adequately protecting sibling directories or newly created directories, is unintuitive and should be removed.

- *Keep application data only in the owner's view*: A severe privacy breach occurred due to improper handling of application profile data. Such data should be viewed only by the owner and unless specifically allowed, should not be visible to other users.

- *Allow hiding of sibling directories from other users*: The POSIX ACLs increase granularity of protection for users, allowing data to be shared with individual users. Combining this with an approach that can completely hide sibling data from other users, thus protecting file metadata.

- *Allow extracting deep rooted directories*: In order to share deep rooted directories, it should be possible to simply pluck them from the file system tree and put them in the view of desired users. Also, it should be possible to share a different representation of the data without impacting the owners' view of the file system.

Based on these design goals, the VBAC access control model creates a new file system primitive called a "**view**". Informally speaking, a data owner can define a view of her home directory, dictating what another user gets to see when he attempts to access it. By adding only the data she wants to share into this view, other data remains protected. Also, it is possible to add a deep rooted directory directly to this view and it can be represented differently. Using such a mechanism, unless the owner explicitly adds her application data into a shared view, it will be always hidden from other users. More details follow in the next section.

2.8 VBAC: Design and Implementation

VBAC extends the *nix access control model by adding a *view* primitive, that presents a different file system structure to different users. For every user, there is one owner-view of the home directory[9], which is the same as the standard home directory in current systems.

[9]Not just home directories, any directory can be protected using VBAC, though its most relevant use for a multiuser enterprise setting appears to protect private data in home directories.

In addition, the owner can define new views of the home directory for other individual users, user groups or the *others* set. For example, a user *bob* can create a view of his home directory for a user *alice* and another view for his user group *faculty* and yet another view for all *other* users. He can then add desired data to appropriate views depending on what he wants to share. The added data could be a deep rooted directory and can be shared using a different name. Other users can access their view of *bob*'s home directory using the same ~`bob` or `/home/bob`. The underlying system **automatically routes** them to their appropriate view and users continue to see the view directory as ~`bob`

The VBAC model uses the same permission types as baseline *nix - *read*, *write* and *execute* and they have the same semantics. VBAC only adds another layer of access control by making a higher level decision of what a user gets to see or not. After that decision, whatever a particular user has in his/her view, it is access controlled using the baseline *nix permissions. We believe that this feature makes VBAC an elegant extension of the *nix model.

Also, a user can decide to **switch off** the additional VBAC layer providing other users with the same view as the owner-view (of course, access to data is controlled by the lower layer of *nix access control). This implies that VBAC can be incrementally introduced into a system without forcing all users to migrate to it immediately.

To avoid managing views at the granularity of individual users, we provide a technique for doing selective sharing of data by using the group views. Recall that we mentioned that in order to do selective sharing under an execute-only directory, a simple solution was to use tough-to-guess directory names. However, it was deemed infeasible since the owner would have to keep tens or hundreds of cryptic names. With the separation of views, an owner can now share a particular directory with a tough-to-guess name in the view, while keeping the original name in the home directory (owner-view). To achieve this, the other users' view is set to execute-only permissions and while adding a directory to the view, the owner also gives a *passphrase* which is used to encrypt the name of the directory being added. The encrypted cipher text is appended to the directory name and the resulting string is used as the target name in the view. Now, the owner authorizes a user to access this directory

by giving that user the original directory name and the *passphrase*. Notice the similarity with the authorization mechanism in original execute-only *nix - only one extra piece of information, the passphrase, is delivered using the same out-of-band channels like email. By using cryptographically tough-to-guess names, we are able to bring this user authorization definition much closer to the underlying system's definition, a shortcoming in the original *nix model.

2.8.1 In-kernel Implementation

In this section, we first describe our in-kernel filesystem implementation of the VBAC access control model. This in-kernel implementation can be used for general *nix systems and the centralized enterprise proxy storage-as-a-service model. Later in Section-2.8.2, we describe how xACCESS supports similar view based design for decentralized storage-as-a-service model.

Our in-kernel file system is called **viewfs** and is based on the Linux ext2 filesystem [30]. Most of ext2 functions like disk placement of data blocks are reused. viewfs is developed as a loadable kernel module and can be loaded into the kernel without kernel recompilation. viewfs was implemented on a Linux 2.6 kernel. Next, we explain the implementation of important VBAC features.

2.8.1.1 VIEW

The foremost VBAC concept is that of a view. From an implementation perspective, a view is a regular directory within the directory containing the owner's home directory (like `/home`). In other words it is a sibling directory to the owner home directory. However, the view directory has a special name of the form: "*.owner.uview.username*" or "*.owner.gview.groupname*" or "*.owner.oview*". This name is restricted, that is, users cannot use this name for naming other directories. This restriction helps to identify a directory being a view of another directory and is used to do automatic routing of users to their views. The restriction is enforced in the file system `mkdir` function implementation. The first type of name "*.owner.uview.username*" is used to create a view for an individual user. For example, if *bob* creates a view for *alice*, the view will be called "*.bob.uview.alice*". The

second type is for a user group and the third is for *other* users. Access is controlled to these views, for example, to prevent user *cathy* from accessing a view for *alice*, by ACLs set at view creation. The period ('.') before the view names keeps them hidden from plain view.

Next, we concretely describe the automatic routing to views. A directory lookup ocurrs in the following manner. Given a name to look up, the directory entry (or *dentry*) cache (or *dcache*) is looked up for the desired directory name. The dcache indexes cached dentries by hashes of their names. If there is a cache miss, the call passes onto the underlying filesystem for looking up that name. That is followed by the inode number lookup to find the inode number corresponding to the name and if it exists, the inode lookup that gets the object metadata.

We modify this filesystem lookup procedure by first checking if there exists an appropriate view directory. For example, if *alice* is looking for a directory `bob`, we check for the existence of a directory called "*.bob.uview.alice*". At this stage, we have the complete state necessary for completing this lookup (since view is the sibling to the home directory). If the view exists then the dentry associated with the view directory is returned and is cached in the *dentry* cache on return. In order to work with different user views in the cache, we modify the hash of the dentry by hashing the view name as opposed to the original directory name. This is feasible since the VFS hash function can be overridden by the underlying filesystem.

An important point to note is that this implementation does not interfere with the I/O paths at all, that is when file data blocks are being read or written. One potential performance impact of the implementation is that a single directory name lookup can cause multiple view name lookups. However, as our initial experiments with benchmarks show, the total overheads are still minimal. Secondly, we foresee *nix installation using viewfs only for user home directories. Therefore the vast majority of system lookups that are to standard OS and other infrastructure files contained in `/usr`, `/etc`, `/bin` are not affected at all.

As mentioned before, an individual owner can choose to switch off the VBAC model

for users accessing her home directory. This is accomplished by keeping an extended attribute [108] with the user home directory. The lookup described above first checks for this extended attribute and in case the user chooses to switch off VBAC, the normal ext2 file system lookup is performed providing the basic *nix model.

2.8.1.2 SHARING DATA

The next important viewfs implementation is its mechanisms for adding data to views. It allows adding deep rooted directories directly to user views and possibly with a different name. Also, changes made to the directory in one view should be immediately reflected in all other views. The first idea that comes to mind to facilitate this is directory hard links. A directory hard link is the same inode as the original directory but can have a different name and can be created at any location within the filesystem without worrying about the access along the path to the original directory (unlike a symbolic link). In fact there is no way to distinguish a hard link from the original directory.

However, directory hard links are not allowed by most operating systems including Linux even though it is not mandated by the POSIX standard. The reason is that many OS mechanisms like reference counting and locking consider the file system to be an acyclical tree and hard links can cause cycles. For example, for `a/b/c`, commands "*link e a/b*" and "*link e/c a*" cause a cycle. This will also break many existing applications that traverse the file system assuming it to be a tree. Symbolic links can also cause a cycle but they are easily identifiable since the link is a different inode that stores the complete path to the original pointed-to location. Hard links, as mentioned before, are unidentifiable and cycle detection for hard links can be expensive.

In the context of viewfs however, we can very easily prevent any cycle formation. We can do this by only allowing users to create a link from inside the view pointing outside and never in the other direction. This can be checked while adding data to a view (creating the hard link) and since views have restricted name, such a check would not be expensive. Even though we had a working implementation of this approach, there is an additional issue specific to the linux operating system and its implementation of the VFS dentry cache. This

view implementation can break in certain situations, for example, when a directory and its hard link are both cached and one is deleted [147]. It is unclear if the same holds for other variants like FreeBSD. However, we preferred linux and opted to take an alternate path.

Similar objectives can be achieved in linux using a bind `mount` [107]. A bind-mount mounts one portion of the filesystem tree at another location. Since it is a separate file system mount, the linux implementation issue discussed earlier does not apply [147]. This allows both sharing the deep rooted directory and sharing it with a different name. For viewfs, an additional requirement is to first create the mount point which will be the desired directory name in the view and is done while adding data to the view.

2.8.2 Integration with xACCESS

The viewfs design is easy to support in xACCESS based storage-as-a-service models. xACCESS, using its metadata and directory-table structures can support manipulation of file objects linked at different directory locations, similar to the bind mount and hardlinks as described above.

2.8.2.1 Views in xACCESS

From an implementation perspective, a view is just like any other directory. The in-kernel viewfs implementation only modified the lookup procedure so that users obtain access to the appropriate view. A similar technique already exists in the design of xACCESS because of the Cryptographic Access Control Primitives (CAPs) based design. In xACCESS, each user has a different *view* of any filesystem object based on the CAP associated with that user's access privileges. Thus, extending xACCESS to support views does not require design modifications. Now, when a user tries to access a particular directory (for example, ~`bob`), the xACCESS filesystem would access the view directory associated with that user. In fact, only the appropriate view's metadata encryption keys would be accessible to that user, thus providing complete security.

The only modification pertains to the location of the view directory. In the in-kernel viewfs design, the view directory is contained as a sibling directory (for example, view directory for `/home/bob` is contained within `/home`). Users use a setuid command to create

views, as it requires modifying the parent directory's directory table to which the user might not have `write` permissions (for example, `/home`). Since setuid's cannot be supported in the storage-as-a-service model, we use a different location for view directories. In the modified scheme, all views are contained within a single sibling *.uname.views* directory which is owned by the user *uname*. Thus, creating views requires modifying this directory's directory-table which is owned by the user creating the view and hence, in possession of the data signing key (DSK) that allows writes. The lookup process is also slightly modified to look for a particular user's view in this directory.

2.8.2.2 Sharing Data with Views in xACCESS

The viewfs filesystem defines new commands that allow users to create views, add data to views and so on. These commands can also be individually implemented for xACCESS. For example, to add a directory to a group view using the passphrase mechanism described above, we store the directory's metadata key encrypted with a keyed hash function [103] derived from the passphrase. This ensures that an authorized user who knows the passphrase can derive the key that decrypts the view directory's metadata key, thus obtaining access to its contents, if authorized.

2.9 Experimental Evaluation

In this section, we perform a detailed evaluation of xACCESS and the viewfs filesystems based on a number of filesystem benchmarks. First, we evaluate the xACCESS filesystem comparing it other related proposals for access control in decentralized storage services model in Section-2.9.1. Next, we evaluate the privacy enhanced filesystem based on view-based access control in Section-2.9.2.

2.9.1 xACCESS Evaluation

The core strengths of xACCESS design lie in its ability to provide an expressive UNIX-like access control model, using symmetric key cryptography for metadata operations and complete in-band management of keys. Through our experiments we want to evaluate the following two aspects:

- *Efficiency of secure decentralized storage-as-a-service model*: First, we want to evaluate how our choice of design features and aggressive security assumptions (of not trusting the SSP) impact the efficiency of the overall storage service model. Our design requires encryption and decryption for every metadata and data access and the CAP based design could also add overheads.

- *Impact of choice of cryptography*: Other proposals for access control in this area use public key cryptography for metadata operations [67, 2, 119]. We want to evaluate if using symmetric key cryptography in xACCESS provides significant benefits, else the public key approaches, which are easier to implement and understand, could be used.

In lieu of these two goals, we compared the xACCESS implementation with the following four implementations:

1. **NO-ENC-MD-D**: This implementation does not encrypt any metadata or data, and thus represents the baseline performance for the networking and other implementation overheads for a wide area file system. It is equivalent to a storage-as-a-service model in which the SSP is fully trusted for data confidentiality as well as access control enforcement.

2. **NO-ENC-MD**: This implementation does not encrypt metadata but encrypts data with symmetric key cryptography. This is equivalent to a system where the SSP is not trusted for data confidentiality but is trusted for metadata and access control enforcement.

3. **PUBLIC**: This implementation encrypts data using symmetric key cryptography and metadata objects with public key cryptography. This is representative of most other access control proposals for storage-as-a-service [67, 2, 119].

4. **PUB-OPT**: This is an optimized PUBLIC implementation which instead of encrypting the complete metadata object with public key cryptography, encrypts metadata objects with a symmetric key and then encrypts that (shorter) symmetric key with public key cryptography. As we discuss later, this optimization provides much better

performance than the original PUBLIC implementation. The data continues to be encrypted using symmetric key cryptography.

Next, we describe our experimental setup.

2.9.1.1 Setup

For most of our xACCESS evaluation, we set up the SSP in College of Computing, Georgia Tech, Atlanta, GA, USA. The server is a shared departmental SunOS server with four 1GHz processors and 8GB RAM. The client was set up in Birmingham, AL, USA, which is nearly 150 miles from Atlanta. The client machine is a Dell Inspiron 8200 laptop running Linux Fedora Core-5 with Pentium-4 1GHz processor and 512 MB RAM. The network connection is a regular DSL home connection with measured upload and download speeds of 850 Kbits/sec and 350 Kbits/sec respectively. This setup is a good exemplification of an actual storage-as-a-service usage scenario - with a non-exclusive SSP server and user accessing data remotely over a wide area network from a home DSL connection. We also evaluated xACCESS performance with different network characteristics (especially high bandwidth connections, representative of better connectivity between client enterprise and SSP) on Planet Lab [145] and results are described in Section-2.9.1.6.

For cryptographic operations, we used National Institute of Standards and Technology (NIST) [127] approved standards used for protecting personal information of federal employees [132]. Specifically, we use 128-bit AES [133] for symmetric key cryptography and 2048-bit RSA [150] for public key cryptography[10]. Finally, all experiments were repeated ten times and results were averaged.

In the next four sections, we present our results on comparing different implementations on a number of micro and macro benchmarks.

[10]A recent work by Shamir and Tomer [161] points out that only 3072-bit public key crypto schemes are equivalent to 128-bit symmetric key schemes. Therefore, by using 128-bit symmetric keys for metadata, xACCESS provides a greater level of security than the PUBLIC and PUB-OPT schemes that use 2048-bit public key cryptography.

2.9.1.2 *Create-and-List Benchmark*

In our first benchmark, we evaluate the core costs of encryption and decryption of metadata objects in the storage-as-a-service model. Metadata is encrypted when new objects are created and decrypted when a `stat` is performed on a filesystem object (which in turn executes the filesystem `getattr` function). For the encryption phase, we created 500 empty files in 25 directories and for the decryption phase we performed a recursive listing using a "*ls -lR*" operation, which `stat`s all files and directories. Figure-17 shows the results of this Create-And-List microbenchmark, with five implementations on the X-axis and time to create/list on Y-axis.

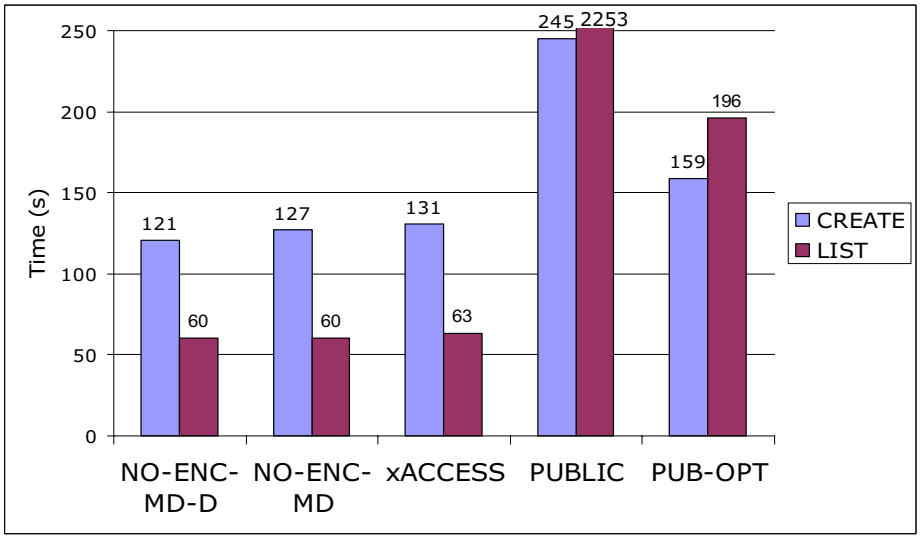


Figure 17: Create-And-List Benchmark

First, we would like to point out the extremely poor performance of the PUBLIC implementation proposed in [67, 119, 2]. For creating 500 files, it took 245 seconds, almost twice as much as the NO-ENC and xACCESS approaches. And importantly, for listing 500 files, it took 2253 seconds as compared to only 60 seconds for NO-ENC approaches. The reason for this huge disparity is the cost of using the **private** key during the list phase. Recall that in this approach, metadata is encrypted with the public keys of users that can access this object. Thus, for a `stat` operation, the metadata needs to be decrypted with the user's private key. As mentioned earlier in Section-2.1.3, operations with the private key are much more expensive. It is this asymmetry that makes the list phase prohibitively

expensive in the PUBLIC implementation.

The PUBLIC implementation can be optimized by avoiding encrypting and decrypting entire metadata objects. Instead, we use the PUB-OPT implementation which uses a symmetric key for encrypting metadata and then encrypts the symmetric key with the public keys of users that can access the object. Thus, now only a small 16-byte key is encrypted and decrypted using public key cryptography. However, as shown in Figure-17, even for this optimized implementation, the create phase is over 30% more expensive and the list phase is over 225% more expensive than the NO-ENC approaches.

In contrast, xACCESS has only 5-8% overheads as compared to NO-ENC approaches. This shows the superior efficiency of symmetric key cryptography in metadata operations. Between NO-ENC-MD-D and NO-ENC-MD approaches that differ in *data* encryption, the list phase is similar as only 25 data blocks are additionally decrypted (the directory-tables for 25 directories). In the create phase, for every new file created, the parent directory's directory-table is re-encrypted and sent to the SSP. This results in a 5% overhead for NO-ENC-MD approach.

This micro benchmark shows that xACCESS is highly efficient for metadata encryption and decryption operations. Next, we take a took at two macro benchmarks that evaluate the filesystem with different operations including data I/O.

2.9.1.3 Postmark Benchmark

Our second benchmark is the popular filesystem benchmark, called Postmark [95]. In this benchmark, 500 small files are created and then 500 randomly chosen transactions (`read`, `write`, `create`, `delete`) are performed on these files. It is a metadata intensive workload representative of web and mail servers. We used the default settings of file sizes ranging between 500 bytes and 9.77 KB. Figure-18 shows the results with varying sizes of the local cache (in percentage of total data size, on X-axis). The size of the cache influences the amount of cryptographic overheads, since for every metadata or data miss, encrypted data is obtained from the SSP and it is decrypted again. We do not compare the PUBLIC implementation and instead use its optimized version, PUB-OPT.

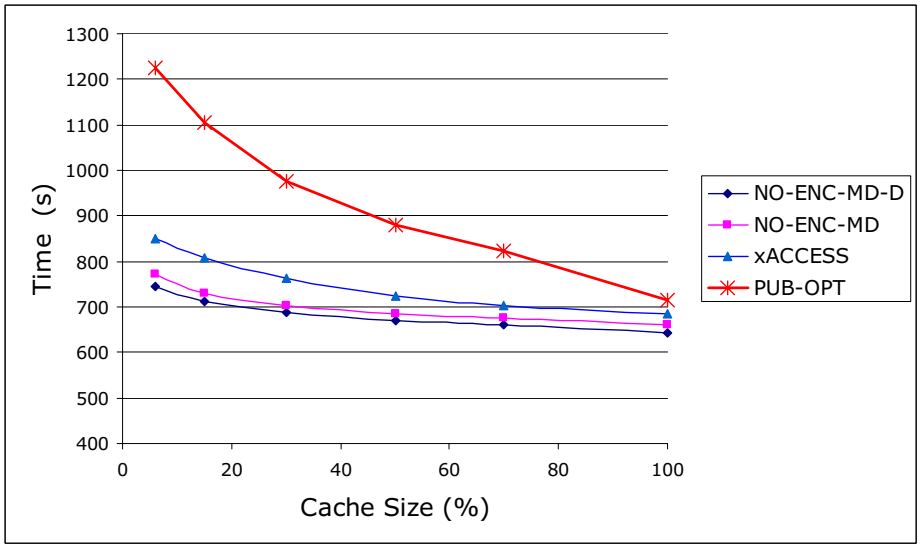


Figure 18: Postmark Benchmark

From the graph, notice that the optimized public key scheme is competitive only for an infinite cache size (100%). As the cache size becomes smaller (typical caches sizes would be in 10-20% range for individual clients), it quickly becomes much more expensive. For example for a 10% cache size, it is 64% more expensive than the NO-ENC-MD-D approach and 43% more than xACCESS. In contrast, xACCESS is always within 15% of the NO-ENC-MD-D approaches. This demonstrates he superior performance of xACCESS for a metadata intensive workload.

Also note that the NO-ENC-MD approach is very close to the NO-ENC-MD-D approach in performance (2-3%). This illustrates high performance of symmetric key cryptography as the two approaches only differ in data encryption. Through xACCESS, we are expecting a similar performance boost by using symmetric key cryptography for metadata objects.

Next, we evaluate xACCESS with a more generic filesystem benchmark – the Andrew Benchmark.

2.9.1.4 Andrew Benchmark

The widely used Andrew Benchmark [80] simulates a software development workload for filesystems. It has five phases: (1) creates subdirectories recursively; (2) copies a source tree; (3) examines the status of all the files in the tree without examining their data; (4)

Table 9: Cumulative performance for Andrew Benchmark

Implementation	Total Time (sec)	Overheads
NO-ENC-MD-D	239	–
NO-ENC-MD	248	3.7%
xACCESS	266	11%
PUB-OPT	384	60%

examines every byte of data in all the files; and (5) compiles and links the files. Phase-2 and Phase-4 are I/O intensive workloads, Phase-3 is similar to the recursive listing, evaluating the costs for the `stat` operation. Phase-5 is a computationally intensive workload in which the benchmark compiles some of the files in the source tree. Figure-19 plots the results for each individual phases and Table-9 lists the cumulative performance for all five phases.

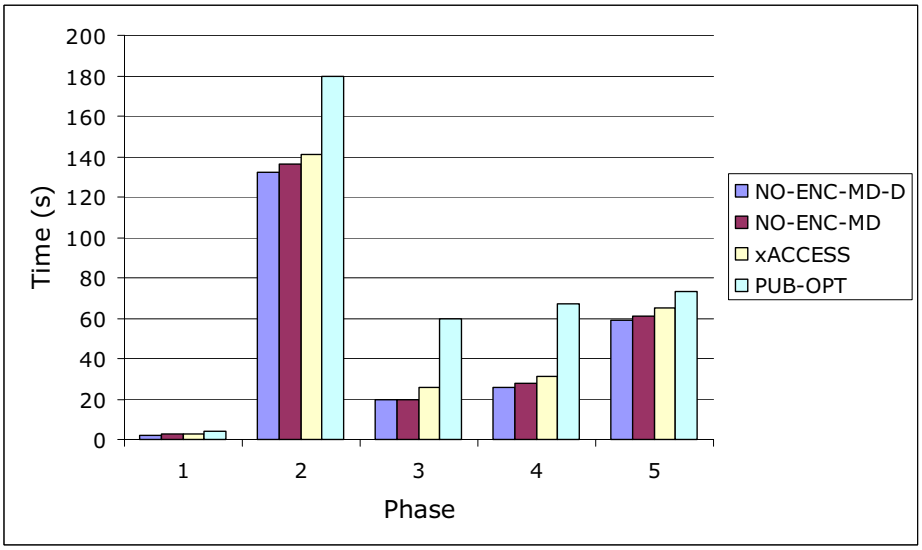


Figure 19: xACCESS Andrew Benchmark Results

From Figure-19, Phase-2 and Phase-4 results show that I/O overheads for xACCESS are minimal. This is because of the use of symmetric key cryptography for both data and metadata. In contrast, for the PUB-OPT approach, even though it uses symmetric key cryptography for data encryption, the metadata overheads are significant. In fact, the PUB-OPT overheads for Phase-2 and Phase-4 are almost equal to the Phase-3 overheads, which is the `stat` operation overheads. Thus, decryption with the private key during the `stat` operation is what makes PUB-OPT approach so expensive.

Cumulatively, xACCESS is only 11% more expensive than the NO-ENC-MD-D approach, which demonstrates that xACCESS delivers good performance for a generic filesystem workload as well.

2.9.1.5 Filesystem Operation Costs

We also analyzed the micro costs of various filesystem operations in xACCESS. We broke the costs into three components – (a) NETWORK: the network traffic costs, (b) CRYPTO: costs for cryptographic operations and (c) OTHER: all other costs. Figure-20 shows these costs for the `getattr`, `mkdir` (for different CAPs), and large file I/O (`read` and `write+close` of 1 MB files).

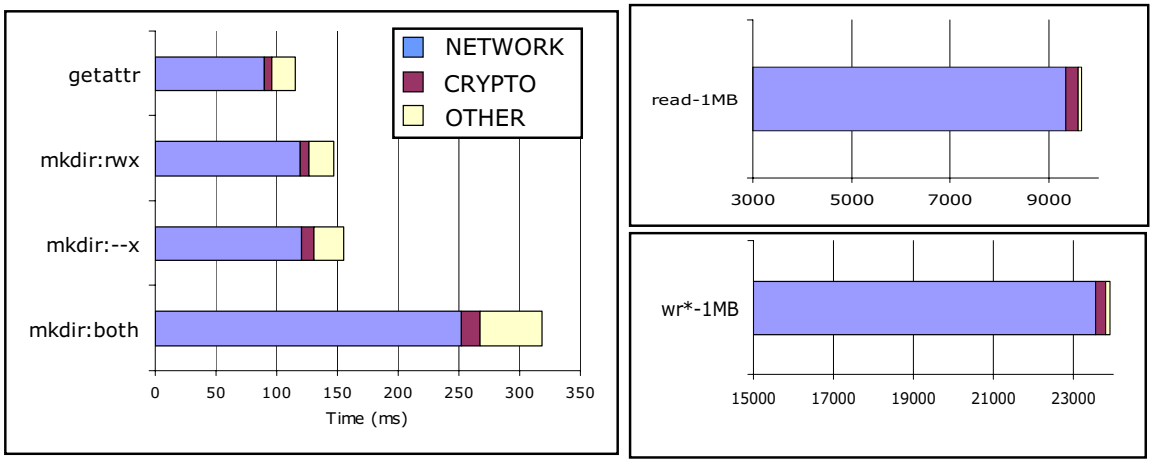


Figure 20: Filesystem Operation Costs

The `getattr` function obtains encrypted metadata from the SSP and decrypts it to access metadata attributes (see Figure-16). This operation completes in a little over 100 ms, with majority of the cost coming from the network component. In fact, the CRYPTO component is less than 7% for all filesystem operations. As seen from the figure, `mkdir` operation is slightly more expensive and its costs can vary based on the CAP required to be created. For example, creating an `exec-only` CAP is more expensive as it requires an additional encryption for the inner directory-table structure (Section-2.4.1). It is also possible that a single `mkdir` operation creates multiple CAPs and we show the costs of creating `read-write-exec` and `exec-only` CAPs. As mentioned earlier, these costs are the result of access control "*embedding*" in addition to core costs of data creation. Fortunately,

the maximum number of CAPs ever required is a small number (5 for UNIX directories).

We also evaluate xACCESS performance for large file I/O. As part of the experiment we read and wrote (+closed) 1 MB files. As the graph shows, xACCESS cryptographic costs are low (less than 7%) of the total costs and the majority of the cost is due to the wide area network communication.

2.9.1.6 Varying Network Characteristics

All experiments described above are representative of an enterprise user accessing data stored at the SSP through a home DSL connection. It is likely that many enterprise locations would have high bandwidth connections with the SSP. As the cryptographic costs and other filesystem overheads are constant, we need to measure the impact of network connectivity on overall xACCESS performance. To evaluate the impact of xACCESS for such connectivity, we evaluated various filesystem operation costs using Planet Lab [145].

For these experiments, we set up the client at a Planet Lab node hosted by Georgia Tech, Atlanta, GA, USA. Since the Linux kernel available with Planet Lab nodes was incompatible with our userspace filesystem toolkit, FUSE, we modified our client filesystem to operate completely in userspace - filesystem calls were simulated using direct function calls. We set up the SSP server at three different locations representing different network conditions. The first setup was a local area network (LAN) with the SSP server also on a Georgia Tech hosted Planet Lab node. For the second setup, we used a high-bandwidth wide area network (WAN) by setting up the SSP server at Duke University, Durham, NC, USA. The third setup was inter-continental (IC) with SSP server at ETH Zurich, Switzerland. Figure-21 and 22 plot the results of our experiments for various xACCESS filesystem operations.

Figure-21 shows the results of the metadata `getattr` and `mkdir`[11] operations. The X-axis shows the different Planet Lab setups, the Y-axis plots the percentage of costs as network, cryptographic and other overheads. The top of the figure also shows the total latencies of the operations in milliseconds.

Notice that the xACCESS cryptographic costs are significant only for the LAN setup.

[11] creating `read-write-exec` and `exec-only` CAPs

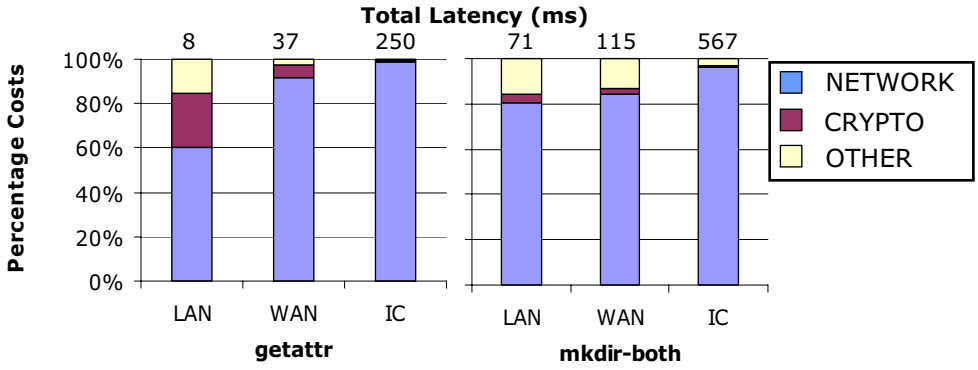


Figure 21: Core Metadata Operations

This is expected as the Planet Lab nodes running the client filesystem and the SSP server nodes are connected through a high speed ethernet connection which has high bandwidth and low latency. This reduces the total cost of the operation, for example 8ms for a `getattr` operation, thus increasing the contribution of cryptographic overheads. For the WAN and IC setups the cryptographic costs again form a small percentage of the total costs even though the total latencies are lower as compared to our earlier home connectivity based experiments. This demonstrates that xACCESS metadata cryptographic costs are minimal for high bandwidth wide area connectivity as well.

We also evaluated xACCESS performance for large file I/Os for the Planet Lab setups. Figure-22 plots the results of reading and writing(+closing) 1 MB files.

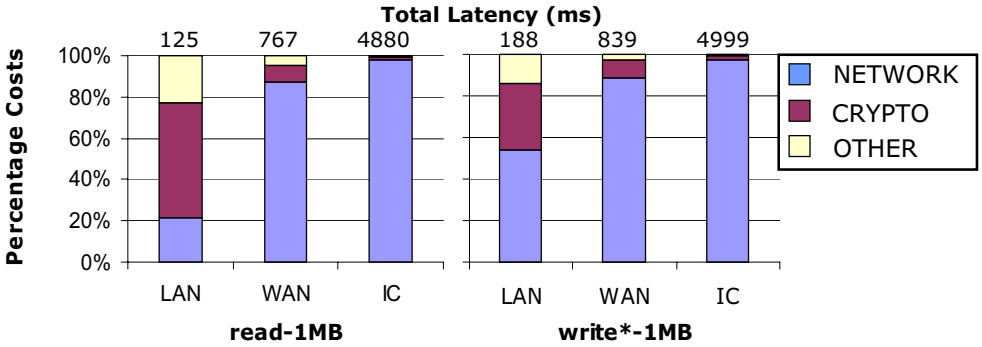


Figure 22: Large File I/Os

As earlier, the LAN setup has significant cryptographic overheads due to reduced overall costs as a result of low network latencies. This illustrates that using encryption in a local area network (*co-located* SSP) would have significant overheads as related to core data access

costs. However, for more representative SSP environments with WAN and IC setups, the cryptographic costs are a small percentage of the total cost. As a result, if it is acceptable for an enterprise to outsource storage to a SSP based on I/O performance, adding security through xACCESS cryptography adds little overheads to the overall cost.

2.9.2 Privacy Enhancements Evaluation

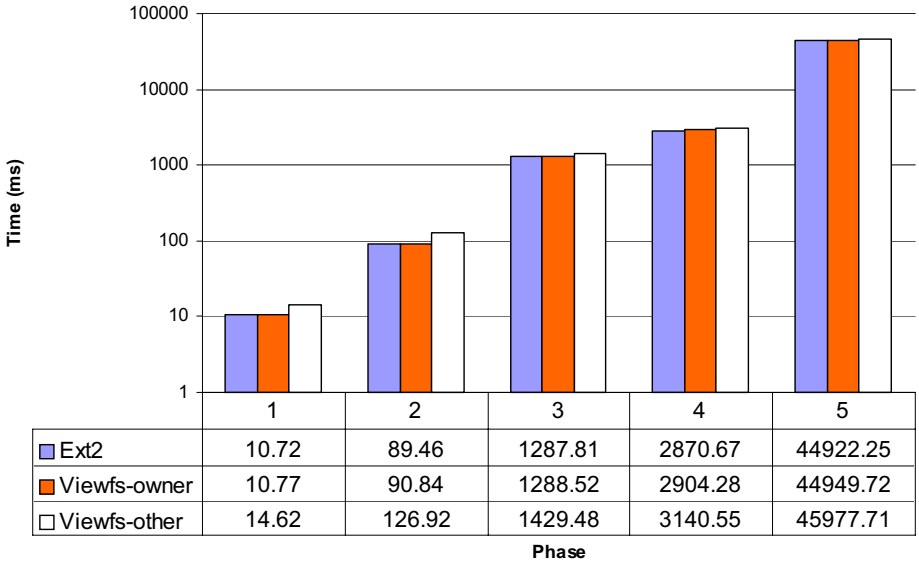
Next, we experimentally evaluate our privacy enhancements. Recall that xACCESS supports the view primitive automatically in its design and these enhancements do not add overheads to xACCESS. Therefore, in these experiments, we evaluate our in-kernel viewfs implementation and compare it with the baseline ext2 performance. The experiments were conducted on a P4 1.6 GHz Dell Inspiron 8200 with 512 MB RAM running RedHat Linux with kernel 2.6.11.3. All results were averaged over multiple runs. We used two viewfs scenarios - (1) *viewfs-owner*: when an owner is accessing her data, and (2) *viewfs-other* when a user is accessing the data from an *others* view. Note that additional view name lookups occur only in the latter scenario, so the former scenario is an indication of any other viewfs overheads, for example, of using bind mounts and would also be an estimate of impact on users with switched off VBAC model.

2.9.2.1 Andrew Benchmark

As for xACCESS, we evaluate viewfs on the Andrew benchmark [80], which emulates a software development workload. In the viewfs-other implementation the benchmark was run by an *other* user in the owner's view directory. Figure-23 plots the times (in ms) on log scale for each of the phases. Table-10 show the overheads of the viewfs-other implementation over ext2.

Table 10: Andrew Benchmark overheads of viewfs-other over ext2

Phase #	Overheads
1	36%
2	41%
3	11%
4	9%
5	2%
TOTAL	3%



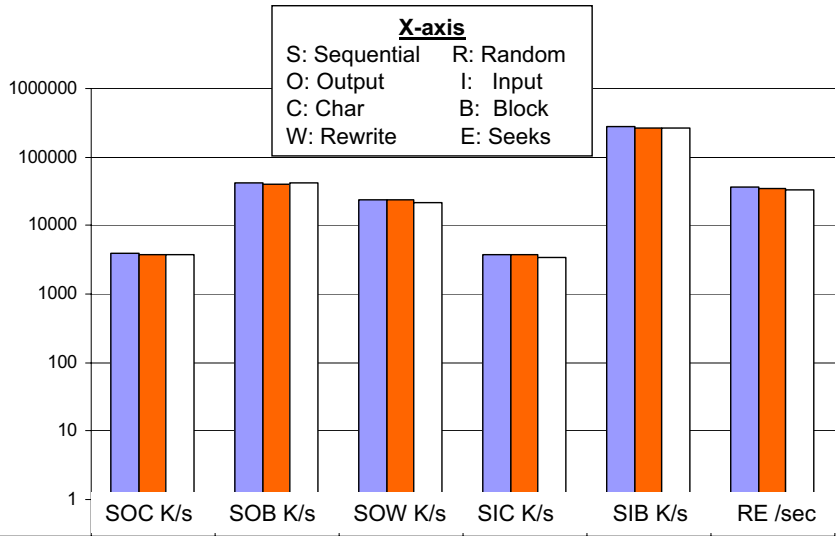
	1	2	3	4	5
Ext2	10.72	89.46	1287.81	2870.67	44922.25
Viewfs-owner	10.77	90.84	1288.52	2904.28	44949.72
Viewfs-other	14.62	126.92	1429.48	3140.55	45977.71

Figure 23: Viewfs Andrew Benchmark Results

Note that the viewfs-owner implementation performs very similar to the ext2 implementation. This implies there is little impact on performance of owners or users with switched off VBAC model. Also, the total overhead for viewfs-other over ext2 is only 3%, implying reasonable overheads of additional view-name lookups under this workload. The overheads in phases one and two are greater since their lookup costs are comparable to the total costs (less than 100 ms). For the later phases, other costs are more dominant.

2.9.2.2 Bonnie Benchmark

In order to test our thesis that viewfs does not impact I/O performance, we evaluated viewfs against ext2 on the Bonnie benchmark [23]. Bonnie tests the speed of file I/O using standard C library calls. It does reads and writes of blocks in random or sequential order and also evaluates updates to a file. The tests were run for a 400 MB file. Figure-24 shows the results for the three implementation for various I/O modes. The X-axis lists the I/O modes and the Y-axis plots the speeds for those operations. As can be seen, for all read/write modes, there is practically no difference between ext2 and viewfs. This proves that viewfs does not have I/O overheads.



	SOC K/s	SOB K/s	SOW K/s	SIC K/s	SIB K/s	RE /sec
Ext2	3916	41195	24020	3706	272410	35984
Viewfs-owner	3850	40327	23677	3691	262909	34591
Viewfs-other	3827	41044	21562	3500	265958	33282

Figure 24: Bonnie Benchmark

2.9.3 Summary of Results

The results of xACCESS and viewfs evaluation demonstrate the following:

- For metadata intensive workloads, where metadata cryptographic encryption and decryption costs could play an important role, xACCESS adds little overheads as compared to schemes with no security (within 10-15% for wide area home connectivity). On the other hand, even the optimized version of the public key implementation (PUB-OPT) had 50-220% overheads. This stems from the primary prohibitive cost of `stat` operations, when decryption with a *private* key is required.

- Even for generic filesystem workloads with data I/O, xACCESS has low overheads. For large file I/O operations, the total cost of xACCESS cryptographic operations added less than 3% of the total costs for wide area home connectivity. For high bandwidth connectivity emulated using Planet Lab, cost of cryptographic operations is significant only for LAN setups. The low percentage of cryptographic costs for other setups demonstrates xACCESS suitability even for dedicated enterprise and SSP connectivity.

Table 11: Comparing xACCESS with Related Work

Scheme	Unstrusted SSP (Confidentiality)	Untrusted SSP (Access Control)	Key Management	Access Control	Metadata Crypto
[200, 197] NASD [66]	Yes	No	In-band	Expressive	–
SNAD [119]	Yes	Partial	In-band	Restrictive	Public
Plutus [94]	Yes	Yes	Out-of-band	Restrictive	Symmetric
Sirius [67]	Yes	Yes	In-band	Restrictive	Public
Farsite [2]	Yes	Byzantine	In-band	Expressive	Public
xACCESS	Yes	Yes	In-band	Expressive	Symmetric

- When new metadata is created, for example with an `mkdir` operation, there can be additional costs for creating multiple CAPs. However, the total number of CAPs ever required to be created is typically a small number (for example, < 5 for UNIX access control model).

- The privacy enhancements add little overhead to the underlying filesystem. The additional lookup operations add up to less than 3% overheads and the I/O overheads are minimal as well.

2.10 Related Work

In this section, we survey other work related to xACCESS and its view based privacy enhancements. First, we differentiate xACCESS from the traditional encryption file systems, where data is also stored in encrypted form (Section-2.10.1). We discuss the technology and access control mechanisms supported by commercial storage service providers in Section-2.10.2 and describe related research access control projects in Section-2.10.3. Next, in Section-2.10.4 we compare our privacy enhancements to two pieces of related work in other security-enhanced *nix access control models and use of views as access control tools in other domains.

Table-11 shows a summary comparison of xACCESS with other approaches on a number of attributes. In general, it can be seen that for systems with similar security models as xACCESS, any system that provides in-band key management uses public key cryptography for metadata operations and provide only a restrictive access control model.

2.10.1 Encryption File Systems

There have been many encrypt-on-disk systems that store data on the media in an encrypted form. Some of the popular encryption file systems include CFS [19], CryptFS [200] and nCryptFS [197]. Most of such systems are local file systems usually aimed at protecting against physical theft/loss of storage media, or protecting against superuser snooping of data. Earlier systems like CFS [19] used a single key to encrypt an entire directory of files, thus restricting OS-like data sharing and requiring out-of-band key management. This would work only for the centralized enterprise proxy storage-as-a-service model discussed in Chapter-1. Later systems like nCryptFS [197] provided support for key management. However these systems always require a trusted kernel that is responsible for identifying users and is also trusted with filesystem metadata. Other networked storage encryption systems like NASD [66] also trust the storage device for authentication and authorization. In contrast, xACCESS provides access control in the untrusted SSP model.

2.10.2 Storage Service Providers

The commercial storage service providers provide a storage medium and management of stored data. Some of the popular enterprise storage service providers are Iron Mountain [93], Arsenal Digital [12], eVault [55] and Amazon S3 [6]. So far, the most attractive feature of these SSPs has been in the disaster recovery and continuous data protection (CDP) space, where they provide techniques to continuously protect data by taking incremental backups to the SSP sites. However, these systems do not provide any in-built access control mechanisms without trusting the SSP. Users can create administrator accounts for greater management functionality, but authentication and authorization is trusted with the SSP. Also, while users can choose to encrypt their data and metadata stored at these SSPs, they have to handle key management by themselves. This limits the usage for enterprises that do not want to trust the SSP for access control enforcement, the threat model assumed in this work.

2.10.3 Security with Untrusted Storage

In recent years, a number of research efforts have developed techniques for security and access control in the untrusted storage model. Plutus [94] describes a file system that aggregates files with similar access privileges into *filegroups* and encrypts them with a single key. Users are responsible for managing keys for different file groups that they can access and for sharing data, users are required to distribute keys out-of-band. Additionally, the access control semantics only provide *read* and *write* permissions at a file level and hierarchical directory-based permissions are not supported. Similar techniques are used in CNFS [76]. In contrast, xACCESS provides an in-band key management technique that provides UNIX-like access control semantics over untrusted SSP-managed storage. It provides full support for hierarchical directory-based permissions and can seamlessly transition local storage to the storage-as-a-service model.

Another effort, Sirius [67] described a filesystem that can provide access control in the untrusted storage model without modifying the storage server. They use public key cryptography for all metadata operations and for. This requires expensive private key based computation for every metadata operation. Also, it does not support any directory-level permissions. Similar public key cryptography technique was used in Farsite [2] which provides a file system over untrusted P2P storage using Byzantine fault tolerance and access control authorization. In contrast, xACCESS predominantly uses symmetric key cryptography and provides complete UNIX-like access control semantics. SNAD [119] also used a public key scheme for metadata key distribution and also trusted the SSP to perform certain verification operations.

Recently, Naor et al [126] have proposed a cryptographic primitive based on the Leighton-Micali [105] or Blom [20] schemes that can reduce public key cryptography costs in the storage-as-a-service model. They have not evaluated the performance of their schemes and also, do not provide an expressive access control model. Another important work in this domain is that of SUNDR [106]. It describes the levels of consistency that can be supported in the untrusted internet storage model (*fork-consistency* [106]). Their work is a complimentary contribution to our work and we are currently integrating their consistency

mechanisms with the xACCESS prototype.

2.10.4 Privacy Enhancement

Most of the research in *nix access control model aims to either improve the granularity of protection (for example, POSIX ACLs [72]) or counter the threat of malicious programs exploiting the `root` (superuser) privileges. One variant of Linux developed by National Security Agency [128], called the Security-Enhanced Linux (SELinux) [159] supports a mandatory access control model [18], in which an administrator sets a security policy which is used to determine the access granted to an object and users have limited control on their data. Another access control model is the Role-Based Access Control (RBAC) model [56, 156], supported by Sun Solaris operating system, in which security attributes can be assigned to user roles (a process or task). This helps reduce the threat of malicious programs exploiting the `root` privileges. In a similar vein, the Rule-Set Based Access Control (RSBAC) model [138] aims to protect against `root` vulnerabilities and improve granularity of protection. There has also been work on a privacy model [57, 58, 59] in access control. However, that work is aimed at guiding organizations on how to control their information flow to ensure privacy of collected user-data (for example, healthcare records). To the best of our knowledge this work is the first critical look at the privacy support for *data sharing in a multi-user *nix operating system.* Our proposed View-Based Access Control (VBAC) model is a specialized access control model aimed at providing stronger privacy protection in such environments.

The use of views as an access control tool has been primarily researched in the area of databases [162, 188]. Using database *views*, users are only shown the relevant data that they have access to. This is similar in concept to VBAC in which only data that needs to be shared is added to a user's view. A view-based access control model has also been used in networking to control access to management information in the Simple Network Management Protocol (SNMP) [193]. There is also an attempt of creating a new operating system called View-OS [180] that presents a different view of the system resources including the filesystem, to an OS process. Also, `chroot` [107] can be used to present a different *root*

(/) directory to a process. In contrast, our view based mechanism is a comprehensive privacy protection mechanism *preventing many kinds of privacy breaches* and *works with existing *nix systems.*

In summary, we consider xACCESS's in-band key management, seamless transitioning ability with an expressive UNIX-like access control and use of symmetric key cryptography for metadata to be our unique attributes. Also, our access model enhancements based on the *view* primitive are a first attempt at comprehensive improvements to the privacy characteristics of the data sharing mechanisms of *nix systems.

2.11 Summary

In this chapter, we described xACCESS, an access control system for decentralized storage-as-a-service model. xACCESS uses novel cryptographic access control primitives (CAPs) to "embed" access control into stored data and does not rely on the SSP for enforcement of security policies. xACCESS is able to support an expressive UNIX-like access control model and its in-band key management technology provides a seamless transition ability from local storage to the storage-as-a-service model with negligible user involvement. Additionally, by primarily using symmetric key cryptography, xACCESS delivers greater efficiency outperforming other proposed systems by over 40% on a number of filesystem benchmarks. We also analyzed our access control model for privacy support in its data sharing mechanisms and proposed enhancements to ensure more secure and convenient data sharing.

In the next chapter, we look at a similar access control problem in a multiuser keyword search application that provides filesystem search over SSP-hosted data.

CHAPTER III

SECURE MULTIUSER FILESYSTEM SEARCH

One insight from the results in the previous chapter is that the primary contributor to xACCESS filesystem operations cost is the wide area network latency. To minimize network traffic, enterprises would like to access only **relevant** files. The commonly used "*grep*" [107] operation to find a file that contains certain keywords, would be an extremely slow operation as it would have to download all files and read their content. Other than latency, there is an added incentive for enterprises to download only relevant content. The storage service providers (SSP) charge clients for the amount of data that is accessed, for example, the Amazon S3 [6] storage service costs $0.20 per month for every GB of data transferred. Thus, keyword search for files stored at the SSP would be a valuable service to an enterprise.

For the decentralized services model, it is also best suited if the search service is hosted at the SSP. Then, users can do keyword search for the files hosted at the SSP and based on the results, download only the relevant files. Searching over encrypted data, however, is a hard problem. Recent work in this area [169, 34, 21] have described some techniques for keyword search, but they only work for single user environments, where all files are owned and accessible to a single user. Enterprise filesystems, on the other hand, have multiple users with different privileges to data and access control needs to be enforced even while searching through the data. As a simple case in point, a user should not be able to search through data that is not accessible to that user. There are additional subtle requirements that surprisingly most of the enterprise search products do not satisfy. We will discuss these requirements later in Section-3.1. Additionally, we do not want to trust the SSP for access control enforcement during search.

We use a different scheme to address this problem. We require all data to be indexed

within a trusted enterprise domain before it is encrypted and stored at the SSP[1]. Access control is "*embedded*" into these indices during the indexing process using encryption and other techniques, ensuring that we do not have to rely on the SSP for access control enforcement during search. The indices are then shipped to the SSP and with slightly modified runtime search process, users can securely search over data. Figure-25 shows the architecture for our approach.

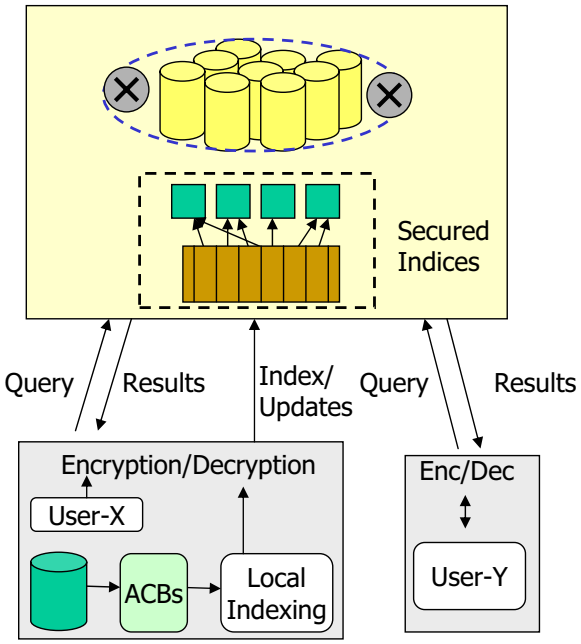


Figure 25: Secure Search over SSP Data

Our technique, usable even in traditional enterprise environments (without outsourced storage to SSP), couples keyword search and access-control into a unified framework to provide secure search. We use a novel building block called access control barrel (ACB) that ensures access control aware search. An ACB is a set of files that have the same access privileges for users and groups in the system and by (logically) dividing filesystem data into independent ACBs, we can ensure that the index for a user is only derived from the data accessible to that user in the underlying filesystem, thus satisfying the access control requirement. Further, by dividing data into independent barrels, data indexing can

[1]Similar technique is used by some data backup systems in which data is indexed before it is stored on a slower magnetic tape device [90].

be distributed to multiple machines for parallel processing. This can significantly reduce total indexing time. We have also developed two optimization techniques that ensure the scalability of our approach even in complex enterprise environments.

The rest of this chapter is organized as follows. We formally characterize the access control aware search problem in Section-3.1. In Section-3.2, we describe current search approaches and their limitations. We describe the design of our approach in Section-3.3 and its architecture and implementation in Section-3.4. We present a detailed experimental evaluation of our approach in Section-3.5. In Section-3.6, we describe the related work in this area. Finally, we summarize our contributions in Section-3.7.

3.1 Access Control Aware Search

As mentioned above, access control needs to be enforced while searching through filesystem data and a user should not be able to search through data that is not accessible to that user. There is an additional requirement. By looking at the results of a query, a user should not be able to *infer* any information that could not have been inferred by that user by accessing the filesystem directly. We refer to this principle as *Access Control Aware Search* or ACAS in short. Simply put, ACAS requires that no additional information can be extracted about the filesystem by using the search mechanism. More formally we define it as:

Definition: ***Access Control Aware Search (ACAS)***
Let $\mathcal{I}_F^U$ be the information that a user U can extract from the filesystem F by accessing it directly (dictated by access rights for U) and let $\mathcal{I}_S^U$ be the information that U can extract by searching on the indices over F over any period of time (based on the search mechanism). The access control aware search (ACAS) property requires that $\mathcal{I}_S^U \subseteq \mathcal{I}_F^U$.

Surprisingly most enterprise search products in the market, like Google Enterprise [70], Windows Enterprise Search [195], IBM OmniFind [89], do not satisfy the ACAS principle. These tools treat search and access-control as two disjoint components. In their approach,

a single system wide index is created for all users and it is queried using traditional information retrieval (IR) techniques (the *search* component). Finally the results (the list of files containing query keywords) are filtered based on access privileges for the querying user (*access control*). However, the ordering and relevance score of results, typically based on Term-Frequency-Inverse-Document-Frequency (TFIDF) measures [196], reveal information that violates the ACAS property. Intuitively, since the index was created based on the lexicon and documents of the complete system, simple post-processing of results would fail to adequately protect system-wide statistics against carefully crafted attacks. We describe this issue and demonstrate an example attack in Section-3.2.

A technique that satisfies ACAS can be found in common desktop search products like Google Desktop [69]. For multiple users on the desktop, these tools create distinct indices for each user on the system, with each user index including all files accessible to that user (the *access-control* component) and then querying only that index for the user (the *search* component). While this satisfies ACAS, it is inefficient due to shared files. We discuss the limitations of these existing search approaches in detail in the next section.

3.2 Limitations of Existing Approaches

In this section, we describe two existing search solutions for integrating access control and analyze their pros and cons from security and performance perspectives.

3.2.1 Index-per-User (IPU) Approach

Most desktop search products like Google Desktop [69], Yahoo Desktop [199] and MSN Toolbar [125] integrate access control during indexing. Each user has a separate index for accessible files, with duplication for files that are shared with other users. This ensures that each user has an index created only from data that was accessible to that user, satisfying the ACAS requirement. These indices can then be securely stored at the SSP by associating with each index a unique symmetric key made available only to that user (for example, by writing to a file in the xACCESS filesystem with permissions only for that user). We describe two technique for creating these secure indices later in Section-3.4.3.

However, this approach causes *additional* disk consumption (and thus monetary costs

in the SSP model) of $\sum(n_i - 1) * I_i$ where n_i is the number of users accessing file F_i and I_i is size of index for F_i. Additionally, each update to F_i causes updates to n_i indices. In an enterprise, where n_i could be in hundreds or thousands, such costs can be prohibitive.

3.2.2 Centralized Single Index (CSI) Approach

The enterprise search products like Google Enterprise Search [70], Coveo Enterprise Search [47] and IBM OmniFind [89] integrate access control at query runtime by creating a single system-wide index and filtering results based on access privileges of the querying user. This provides maximum space and update efficiency. This, however, requires the presence of a trusted access control engine at query run-time and thus would force the search service to be hosted at the client enterprise.

Also importantly, these products do not satisfy the ACAS requirement and by carefully crafting queries, a user can obtain information about the underlying filesystem which could not have been inferred otherwise. Below, we describe an example attack that can determine the total number of files containing a particular keyword even when the attacker does not have access to all files containing that keyword. For example, an attacker could monitor the enterprise filesystem to see the number of files containing the word "*bankruptcy*". A sudden increase in the number of such files could alert him/her to sell off company stock, practically amounting to insider trading. This violates the ACAS property, as this information could not have been determined by the attacker through the underlying filesystem directly.

We assume that the relevance score of a result file f_i is computed by the standard TFIDF measure

$$rel(f_i) = \sum_{t_j \in Q} w_{ij} * w_{Qj} \qquad (1)$$

where t_j are the terms in the query Q and w_{ij} is the normalized weight of term t_j in f_i given by

$$w_{ij} = \frac{o_{ij} * log\left(\frac{|N|}{n_{t_j}}\right)}{\sqrt{\sum_{t_k \in f_i}(o_{ik})^2 * \left(log\left(\frac{|N|}{n_{t_k}}\right)\right)^2}} \qquad (2)$$

where o_{ij} is the number of occurrences of term t_j in f_i, $|N|$ is the total number of files in

the system and n_{t_j} is the number of files that contain t_j. w_{Qj} is defined similarly.

The attacker, Alice, wishes to know the number of documents that contain the term t_q (e.g. "*bankruptcy*"). The attack works in three steps. First, Alice picks two *unique* terms t_1, t_2 (no file contains these terms) and creates two new files: f_1 containing terms $\{t_1, t_2\}$ and f_2 containing terms $\{t_2, t_q\}$. Note that after creating the files, o_{11}=o_{12}=o_{22}=o_{2q}=1, n_{t_1}=1 and n_{t_2}=2. In the second step, she queries for term t_1 and from (1) and (2) she can calculate $|N|$

$$|N| = 2^{\frac{1}{1-\sqrt{\frac{1}{rel(f_1)^2}-1}}} \qquad (3)$$

In the final step, Alice queries for term t_q and calculates n_{t_q} from (1), (2), (3). This completes the attack.

$$n_{t_q} = 2^{\frac{1}{\sqrt{\frac{1}{rel(f_2)^2}-1}}} * |N|^{\left(1-\frac{1}{\sqrt{\frac{1}{rel(f_2)^2}-1}}\right)}$$

Such attacks are possible on most TFIDF based measures including the popular measure Okapi BM25 [151]. Additionally, even when relevance scores are not returned as part of the result, good approximations to n_{t_q} can be obtained by exploiting ordering of the results [28]. A recent effort [28] describes an ACAS compliant approach that uses a complex (and trusted) query transformation at runtime for access control. Using such an approach would require a trusted search server implying that the service should be hosted at the client enterprise. Additionally, it requires to maintain access control lists for all files in the filesystem in-memory which is extremely inefficient for large enterprise environments.

Next, we describe our distributed enterprise search approach, which provides greater efficiency than the IPU approach and ACAS compliance unlike the CSI approaches. Additionally, our approach allows indices to be hosted at the SSP.

3.3 Distributed Secure Enterprise Search

The IPU approach is secure but inefficient for enterprise search, whereas CSI approaches are insecure in terms of ACAS and require enterprises to host the search indices themselves. In this section, we describe the design and implementation of our distributed approach to enterprise search based on the concept of access control barrels (ACBs). Our approach

provides security, efficiency and allows indices to be hosted at the SSP.

Before we get into the details of our approach, we briefly describe the access control notion of searchability for the UNIX permissions model (also followed by various other flavors like Linux and FreeBSD).

3.3.1 Searchability and Access Control

In an enterprise environment, not all data is accessible to all users. Access to data is controlled through the underlying filesystem's access control mechanisms. In this work, we follow the UNIX permissions model [148], which is also closely followed by xACCESS.

In context of indexing and search, there is a need for a notion of **searchability**. Clearly *read* permissions on a file allow it to be searched and *write* permissions do not influence searchability (and are ignored in the process). The *execute* permissions for directories have a non-obvious influence on searchability. Users with *execute-but-not-read* permissions on a directory can access its contents only if they know the exact names of subdirectories or files. As this out-of-band notion can not be adequately and safely captured, consistent with the UNIX and Linux `find/grep/slocate` [107] paradigms, we consider such directories as being not searchable. Formally, searchability is defined as:

Definition: ***Searchability*** – *A file, F is* ***searchable*** *by user u_i (or group g_m) if there exist* read ***and*** execute *permissions on the path leading to F and* read *permissions on F, for u_i (or g_m).*

3.3.2 Design Overview

The main design principle of our approach is to efficiently integrate access control into the indexing phase such that the indices used to respond to a user's query are derived only from the data accessible to that user. This will ensure that we satisfy the ACAS requirement and do not have to do access control based filtering on query results. It also helps in key management when these indices are eventually encrypted to be stored at the SSP. We accomplish this goal with a pre-processing step that (a) constructs a user access hierarchy for the users and user groups in the system (Section-3.3.2.2) and (b) logically

divides data into access-privileges based *access control barrels (ACB)*. We first provide a brief description of ACBs and then describe in detail how they work in conjunction with the user access hierarchy.

3.3.2.1 Access Control Barrels

An ACB is a set of files that share common searchability access privileges (as defined in Section-3.3.1). That is, all files contained within an access control barrel can be accessed (and thus searched) by the same set of users and user groups. For example, one barrel could contain files accessible to user *bob* and another for a user group *students*. Intuitively, the idea of barrels is that if we can efficiently create collections of files based on their access privileges, to provide secure search to a user, we can pick the collections that this user has access to and serve the query using only those indices.

This might sound similar to the index-per-user (IPU) desktop search approaches [69, 199, 125] where all files accessible to a user are grouped into a single collection and files accessible to multiple users are duplicated in their collections. ACBs avoid their inefficiencies by following an additional neat property of *minimality*. This property ensures that each file can be uniquely mapped to a single barrel, avoiding duplicating them in multiple collections (we defer the discussion of implementing such minimal ACBs to the next section). Now, files accessible to multiple users are grouped into *shared* collections and secure search for a user combines the user's private collections with these shared collections using distributed information retrieval [104]. This is efficiently accomplished using the user access hierarchy which is described next. We will also compare the ACB based approach with the index-per-user approach in detail in Section-3.5.5.

3.3.2.2 User Access Hierarchy

The user access hierarchy data structure has two main tasks: (1) provide a mechanism to map files to access control barrels, and (2) provide techniques to efficiently determine all barrels that contain files searchable by a user. In what follows, we first give a high level description of the data structure and later detail how the hierarchy is constructed for UNIX permissions model.

For most access control models a user is associated with two types of credentials: (i) a unique user identifier (*uid*), and (ii) one or more group identifiers (*gid*) corresponding to the user's group memberships. We represent the set of all such user and group credentials as a directed acyclic graph called Access Credentials Graph or ACG in short. For example, there could be a node for credential uid_{bob} or $gid_{students}$. Every node V_i in this graph is associated with a corresponding barrel ACB_i. Our example nodes uid_{bob}, $gid_{students}$ are associated with barrels containing files with searchability privileges to user *bob* and group *students* respectively. Now, mapping files to barrels is equivalent to assigning files to a node in ACG (the first task mentioned above).

For a node, V_u (associated with the *uid* credential of a user u), let V_u^* denote the set of all nodes in the *directed* graph ACG that are reachable from the vertex V_u. Our construction of the graph ACG will ensure that a file F is searchable for a user u if and only if F is assigned to some vertex $v \in V_u^*$. With this property, the results for user u's query can be computed by combining indices from barrels associated with nodes in V_u^*. The set V_u^* can be efficiently determined using a simple depth or breadth first search on the graph ACG. This accomplishes the second task of the user access hierarchy data structure.

Next, we explain in detail the process of constructing the graph ACG with aforementioned properties from UNIX-like user credentials. In a UNIX-like access control model a credential C can be expressed in Backus Naur Form (BNF) as:

$$\begin{aligned} C &= root \mid all \mid P \\ P &= uid \mid gid \mid P \wedge P \mid P \vee P \end{aligned} \qquad (I)$$

Note that *root* is a special user with super-user privileges and *all* indicates a credential for all users and groups. We need the $\vee$ operator on the principles to handle POSIX Access Control Lists [72] that allow associating multiple users and groups with a file F (as opposed to the usual $\{owner, group, others\}$ model [148]). We need the $\wedge$ operator on the principles to handle the implicit conjunction operation that occurs while traversing the directory hierarchy leading to file F (for example, directory `X/Y` where `X` has access only for user group *students*, and `Y` has access only for user group *grad-students*; only users that

belong to both groups can access data under `Y`). We define an implication operator $\Rightarrow$ which specifies if one credential can *dominate* another. For example, $\forall u$, $root \Rightarrow u$ says that *root* can access data that any other user can.

Permissions on a file can also be expressed based on a credential defined as above. For example, for a file F that allows access to users x, y and group z, we say that it has a credential $C_F = \{uid_x \vee uid_y \vee gid_z\}$ and is interpreted to say that access is allowed to users who have a credential that dominates this file's credential (user x has access to F since $uid_x \Rightarrow C_F$). Now, if we can create a barrel for each such file credential in the system, we can uniquely map a file to a barrel, achieving ACB minimality. While theoretically the total number of barrels (one for each possible access control setting, thus exponential in number of users and groups) can be very large, practically, this is hardly the case as many files in the system share common file credentials. Other studies have also made similar observations, for example, the filegroups concept of Plutus [94] uses this to ease key management for encrypted data storage. Regardless, in Section-3.3.3, we will describe two optimization techniques that address this potential scalability issue.

Finally, we construct the graph ACG as follows. First, the set of vertices and edges are initialized by adding vertices for all user and group credentials and adding edges for the simple $\Rightarrow$ relationships: $root \Rightarrow u$ $\forall u \in U$; $u \Rightarrow g$ $\forall g \in G(u)$; for any group g, $g \Rightarrow all$, where $G(u)$ denotes the set of all groups to which the user u belongs to. Formally,

$$\begin{aligned} V_{ACG} &= \{V_{root}, V_{all}\} \cup \{V_u \mid \forall u \in U\} \cup \{V_g \mid \forall g \in G\} \\ E_{ACG} &= \{V_{root} \rightarrow V_u \mid \forall u \in U\} \cup \{V_u \rightarrow V_g \mid \forall u \in U, \forall g \in G(u)\} \\ &\quad \cup \{V_g \rightarrow V_{all} \mid \forall g \in G\} \end{aligned}$$

where $\rightarrow$ indicates a directed edge in the graph.

Next, the $\vee$ or $\wedge$ nodes are added when we encounter files with such credentials. This is done during the pre-processing step while assigning files to their appropriate barrels (as described in Section-3.4). For each such file, we insert a new vertex V_C for the file's credential C. Then, we find the set of all vertices whose credentials minimally dominate the new credential C (say, $minDom(C)$) and for every such vertex, $V \in minDom(C)$, we add

an edge from vertex V to V_C. Note that this guarantees that the vertex V_C is reachable from all vertices that have a dominating credential, by the transitive nature of the $\rightarrow$ operator on the graph ACG. Similarly, we find the set of all vertices whose credentials are maximally submissive to the new credential C (say, $maxSub(C)$). For every such maximally submissive credential $V \in maxSub(C)$, we add an edge from the vertex V_C to V. Additionally, we remove redundant edges between vertices in $V_1 \in minDom(C)$ and $V_2 \in maxSub(C)$ if V_2 is now reachable from V_1 via the new vertex V_C. Formally, it can be represented as:

$$
\begin{aligned}
Dom(C) &= \{V_{C'} \mid C' \Rightarrow C\} \qquad\qquad (II) \\
minDom(C) &= \{V \in Dom(C) \wedge \neg\exists V' \in Dom(C), V \in Dom(V')\} \\
Sub(C) &= \{V_{C'} \mid C \Rightarrow C'\} \\
maxSub(C) &= \{V \in Sub(C) \wedge \neg\exists V' \in Sub(C), V \in Sub(V')\} \\
E_{ACG} &= E_{ACG} \cup \{V \rightarrow V_C \mid \forall V \in minDom(C)\} \cup \{V_C \rightarrow V \mid \forall V \in maxSub(C)\} \\
E_{ACG} &= E_{ACG} - \{V_1 \rightarrow V_2 \mid \forall\, V_1, V_2, V_1 \in minDom(C) \wedge V_2 \in maxSub(C) \\
&\quad \wedge V_1 \rightarrow C \in E_{ACG} \wedge C \rightarrow V_2 \in E_{ACG}\}
\end{aligned}
$$

Figure-26 shows an example ACG graph for a system with three users, two groups and group membership as shown in figure.

Using this access hierarchy, we can map all files to their appropriate barrels and also identify barrels searchable by a particular user (equivalent to finding V_u^* – a simple depth first search operation on ACG).

3.3.2.3 ACB Minimality

In this section, we formally state the minimality property satisfied by our ACB construction in Section 3.3.2.1. We use this claim 3.3.1 in Section-3.3.3 to present optimization techniques on our ACB approach.

Claim 3.3.1. *It is impossible to reduce the number of ACBs without either duplicating files in barrel indices or violating the ACAS property.*

Proof. We can prove this by a simple contradiction argument. Let ACB' denote a set of access control barrels such that the number of barrels in ACB' is smaller than ACB and it

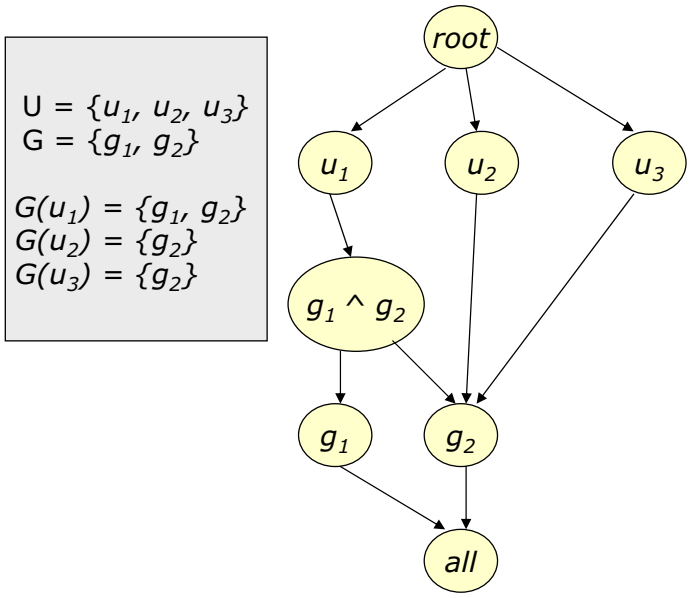


Figure 26: Example Access Credentials Graph

contains exactly one copy of each file in a barrel index and it respects the ACAS property. Since ACB' has smaller number of barrels, there exists a barrel $b' \in ACB'$ such that it has two files f_1, $f_2 \in b'$, where f_1 and f_2 belong to two distinct barrels b_1 and b_2 in ACB (by pigeon hole principle). Since f_1 and f_2 are in different barrels in ACB, the files must have different access control expressions. Hence, there may exist a user u such that u can access only file f_1 but not file f_2.

Now, for the file f_1 to be searchable by user u, there has to be some barrel bar in ACB' such that $f_1 \in bar$ and the barrel bar is reachable from the vertex V_u on the ACG. Since there is only one copy of the file index f_1 in ACB', the barrel b' must be reachable from V_u on the ACG for ACB'. Since the barrel b' is reachable from V_u all the files in the barrel b' must be accessible to the user u. Clearly, allowing the user u to search file f_2 violates the ACAS property. Thus, in order to reduce the number of ACBs, we have to allow either duplication of files indices in barrel or compromise the ACAS property. □

3.3.3 Scalability Optimizations

In most real enterprise environments the number of barrels per user is typically very small (6–8 on average in our observations for two enterprise filesystems; similar observation was

made in [94]). However, theoretically this number is exponential in the number of users and user groups. In this section we describe two optimization techniques that can preserve the scalability of our approach even in such rare hostile setups.

Our optimizations are based on our ACB minimality claim 3.3.1. Our first optimization trades off the number of barrels with the number of copies of a file index and the second technique trades off the number of barrels while allowing controlled violation of the ACAS property. Both techniques provide a control mechanism for administrators to choose appropriate trade-offs. The optimizations transform the access control graph (ACG) with the goal of decreasing the number of ACBs. However, such transformations must preserve *searchability*, that is, if a file f is accessible to user u then the file f must be searchable. Using ACBs this implies that if a file f is accessible to user u, then in any transformed ACG, the file f belongs to some barrel b such that b is reachable from V_u on the ACG ($b \in V_u^*$). We call this property *reachability* on the ACG.

3.3.3.1 File Index Duplication

Our first optimization reduces the number of ACBs and satisfies the ACAS property at the cost of maintaining duplicate file indices. Let us consider any vertex V_C in the ACG such that the credential $C \neq u$, for any user $u \in U$. One can eliminate the vertex V_C from the ACG (thereby decreasing the total number of barrels by one) by adding all the file indices in V_C to every vertex $v \in minDom(V_C)$, where $minDom$ is defined in Equation-(II) above. Note that if $C \neq u$, then $minDom(V_C) \neq \Phi$, that is, there exists at least one vertex $v \in minDom(V_C)$. If V_C is reachable from some vertex V_u (for user u and $C \neq u$) then at least one vertex $v \in minDom(V_C)$ is reachable from V_u; thus the above construction satisfies reachability on the ACG and thus preserves searchability. The construction preserves the ACAS property since the credential $domC$ associated with any vertex $v \in minDom(V_C)$ dominates the credential C ($domC \Rightarrow C$). Hence any user u that satisfies the credential $domC$ also satisfies the credential C. Hence, the above construction eliminates the vertex V_C at the cost of retaining $|minDom(V_C)|$ copies of file indices for each file in V_C.

In our implementation we define a tunable parameter $minf$ – the minimum number of

files per barrel such that if a larger $minf$ is chosen the number of barrels decreases at the cost of more duplication of files indices. Given the parameter $minf$ we present a greedy algorithm to reduce the number of barrels as follows:

1. Sort the barrels in increasing order of size (number of files) b_0, b_1, $\cdots$ b_k.
2. Pick the smallest i such that $b_i < minf$ and the credential associated with barrel b_i is not equal to u for any user $u \in U$. If there is no such barrel the procedure terminates.
3. Eliminate the barrel b_i by suitably replicating all its file indices (as described above).
4. Barrel elimination may change the size of other barrels; so resort the barrels according to their size and repeat the procedure.

Observe that by setting $minf = \infty$, the above procedure would terminate with $|U|$ barrels where each barrel b_u is the per-user index as generated by the index-per-user (IPU). This ensures that for any finite $minf$, our algorithm would have fewer file index duplicates when compared to the IPU approach.

3.3.3.2 Access Control Optimization

Our second technique reduces the number of ACBs while maintaining only one copy of each file index at the cost of violating the ACAS property. Let us consider any vertex V_C such that $C \neq all$. One can eliminate the vertex V_C from the ACG by adding all the file indices in V_C to some vertex $v \in maxSub(V_C)$, where $maxSub$ is defined in Equation-(II). Note that if $C \neq all$, then $maxSub(V_C) \neq \Phi$, that is, there exists at least one vertex $v \in maxSub(V_C)$. If V_C is reachable from some vertex V_u (for user u) then all vertices $v \in maxSub(V_C)$ is reachable from V_u; thus our construction satisfies reachability on the ACG and thus preserves searchability. However, there may exist a user u' such that a vertex $v \in maxSub(V_C)$ is reachable from $V_{u'}$ but not the vertex V_C. This is possible because the credential C dominates a credential $subC$ associated with vertex $v \in maxSub(V_C)$. Hence, the above construction maintains exactly one copy of every file index, but may violate the ACAS property. Unlike the centralized single-index approach that violates the ACAS

property for all the files in the file system, our approach allows us to control the number of such violations.

Given the parameter $minf$, the minimum number of files per barrel, the following greedy reduces the number of barrels:

1. Sort the barrels in increasing order on their size (number of files in the barrel) b_0, b_1, $\cdots$ b_k.

2. Pick the smallest i such that $b_i < minf$ and the credential associated with barrel b_i is not equal to all. If there exists no such barrel the procedure terminates.

3. Eliminate the barrel b_i by copying the file indices in b_i to at least one vertex $v \in maxSub(V_C)$, where C is the credential associated with barrel b_i.

4. Eliminating a barrel may have changed the size of other barrels; so we resort the barrels according to their size and repeat the procedure.

If $|maxSub(V_C)| > 1$, one can randomly pick a vertex v from $maxSub(V_C)$. However, we heuristically pick a vertex v such that it satisfies the $minf$ requirement while incurring only a small number of access control violations. Our first heuristic picks the vertex v that has the smallest barrel associated with it. This heuristic clearly favors our goal of achieving at least $minf$ files per barrel.

Our second heuristic attempts to reduce the number of files violating the ACAS property by picking a vertex v whose credential is the least *popular*. For example, let us suppose that the credential associated with v is a $\wedge$-group $cred = g_{i_1} \wedge g_{i_2} \wedge \cdots \wedge g_{i_k}$. We measure the popularity of the credential $cred$ as $pop(cred) = \prod_{j=1}^{k} pop(g_{i_j})$, where popularity of a group g is determined by the number of members in the group (normalized by the total number of users $|U|$). Similarly, we measure the popularity of a $\vee$-user credential $cred = u_{i_1} \vee u_{i_2} \vee \cdots \vee u_{i_k}$ as $pop(cred) = \frac{k}{|U|}$. Clearly, the less popular a credential $cred$, the smaller number of users that satisfy $cred$; hence, fewer users can reach the vertex v from V_u causing fewer violations. Note that every user u can that reach V_C can reach $v \in maxSub(V_C)$; hence the approach does not compromise on searchability. This approach attempts to minimize the

number of users u' that can reach $v \in maxSub(V_C)$, but not the vertex V_C itself, thereby reducing the number of ACAS violations.

These two optimizations can be used to improve the scalability of the system in rare environments where the variation in access control settings increases the number of ACBs per user. Further, we allow an administrator to make an intelligent decision on the choice of the optimization strategy. For instance, files with high update rates may use only the second optimization technique. This ensures that we have only one copy of the file index and thus keeps the update costs low. Similarly, files with lot of critical information may use only the first optimization technique. This ensures that there are no ACAS violations on the critical file data.

In the following section, we integrate our ACB based technique with the architecture of our indexing and search system.

3.4 Architecture and System Implementation

So far, we have introduced the concept of access control barrels (ACBs) and the user access hierarchy as a tool to (a) efficiently map files to ACBs and (b) determine accessible ACBs for a querying user. In this section, we will explain the overall architecture of our indexing and search system and how it fits into the enterprise infrastructure and the storage-as-a-service model.

For management of the distributed search environment during the indexing and securing phase, one enterprise machine is chosen as a global orchestrator. As mentioned earlier, the distributed nature of our approach allows us to use other underutilized enterprise machines as well. All such participating machines run a thin client version of the system and receive commands from the global orchestrator for indexing of access control barrels. Figure-27 shows the workflow for the process.

3.4.1 Pre-processing: Creating ACBs

As part of the pre-processing step, we first create the basic *ACG* from the user and user groups in the system as described in Equation-(I) above. For UNIX/Linux systems user and group information is obtained from */etc/passwd* and */etc/group*. Next, we initiate a

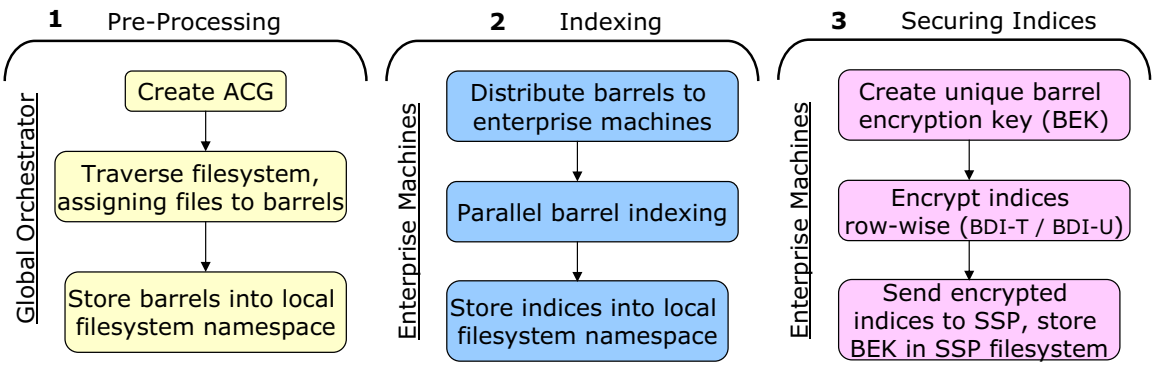


Figure 27: Secure Distributed Search Workflow

filesystem traversal for all data that needs to be indexed. This is required for mapping files to ACBs (represented by a vertex in ACG). As mentioned earlier, this process occurs before data is sent to the SSP, so we assume that all data is currently within the enterprise location.

During the traversal, we associate each file with a vertex in the directed ACG graph based on its searchability privileges. This mapping is done by finding the vertex that has the same credential as the file (e.g. V_{bob} for credential uid_{bob}). If the file has a $\vee/\wedge$ credential, a new node is added to the access hierarchy and the file is mapped to that node. At the end of this filesystem traversal, we have all barrels in the system and the list of files that are contained in each such barrel. These lists are written to per-barrel files, that are *securely* stored in the enterprise filesystem namespace with access privileges only to the superuser. This stored file is the embodiment of our abstract ACB concept. This completes the pre-processing step and is usually performed by a single enterprise machine – the *global orchestrator*, which stores the user access hierarchy.

3.4.2 Indexing

After creating ACBs, the next step is to index documents for each barrel. These ACBs can be indexed independently unlike the single index approach where the computation of TFIDF statistics requires centralized indexing of data. The global orchestrator distributes this barrel indexing task to participating enterprise machines. As the barrels are stored in a global namespace and accessible to all enterprise machines, the orchestrator only needs

to pass the barrel IDs to these machines. The orchestrator can easily optimize available resources by doing an intelligent distribution of barrels to machines (ensuring no single machine is overly loaded). As we show later in Section-3.5, this indexing task distribution provides excellent savings.

On receiving commands from the orchestrator, thin-client agents of enterprise machines retrieve the barrels from the filesystem namespace and start indexing documents. An index is typically comprised of: (a) vocabulary for words that appear in the documents and (b) a words to filename mapping along with their TFIDF statistics used later for ranking. Once indexed, these indices are stored back into the global namespace. Our access privileges based design of barrels provides a natural way of storing indices securely in this namespace. The index files are stored with the same privilege as the files contained in that barrel (all files in a barrel have the same privileges)[2]. This allows only the users that had access to files of a barrel (and thus can search through that barrel) to obtain these indices and provides a natural security mechanism for storing these indices using the underlying filesystem access control.

3.4.3 Secured Indices and Search

In order to secure the indices for hosting at the SSP, we need to encrypt the indices appropriately. There are two goals for this security process – (a) hide file names and vocabulary information from the SSP and (b) ensure that users only obtain results from indices that they can access (barrel indices associated with V_u^* in the ACG).

For a vocabulary of keywords, an index consists of multiple rows with a row for each keyword w. The row for w includes its TFIDF statistics from the files in barrel ACB_i and a list of files that contain w. We can hide the file names and vocabulary from the SSP using the following cryptography technique.

We associate each barrel ACB_i with a randomly generated barrel encryption key BEK_i.

[2]The $(u_i \vee u_j)$ barrels are handled by using POSIX ACLs for (u_i, u_j) on the indices and $(g_k \wedge g_m)$ are handled by keeping indices under directory hierarchy `X/Y` with `X`, `Y` having privileges for g_k, g_m respectively.

Only the users who have the required credentials to search the barrel ACB_i have the knowledge of BEK_i. This can be done by writing the key to a file stored in the filesystem namespace using the same permission settings as the barrel itself. Only the users that can access ACB_i can get this key. Note that the xACCESS system described in the previous chapter is the one that allows us to set such permissions on files in the storage-as-a-service model.

Now, given a barrel encryption key BEK_i, we replace each keyword w in barrel ACB_i's vocabulary with a keyed hash value $KH_{BEK_i}(w)$, where KH denotes a pseudo-random function like HMAC-MD5 or HMAC-SHA1 [103]. Next, we present two approaches that differ in the the computation/communication overhead incurred at the client at the risk of exposing certain information to the SSP.

As a first approach (referred to as BDI-T) we leave the TFIDF statistics unencrypted and encrypt *each* file name *separately* with the barrel encryption key BEK_i. Now, given a search query with keyword w from a user u, the user's client initiates a search for $KH_{BEK_i}(w)$ for all i such that ACB_i is searchable by the user u. Note that if a barrel j is not searchable by the user u, then the user u does not know the barrel encryption key BEK_j and thus cannot even guess the keyword $KH_{BEK_j}(w)$. The SSP performs a regular distributed IR search over user-accessible barrels for $KH_{BEK_i}(w)$ and returns a ranked list of encrypted file names which the user can decrypt with BEK_i. SSP can do this ranking as TFIDF statistics were left unencrypted.

However, this approach is vulnerable to a frequency inference attack (on the frequency of keywords in the index). A frequency inference attack attempts to infer a keyword from its popularity, say the number of files that contain the keyword, information which is contained within the unhidden TFIDF statistics. Such frequency inference attacks can be thwarted using statistics hiding approaches or by multiple SSPs. Please refer to [165] for one such technique.

A second approach, referred as BDI-U, is to hide the index statistics from the SSP as well. Similar to the first approach, we replace each keyword w in barrel ACB_i's vocabulary with a keyed hash value $KH_{BEK_i}(w)$, but instead of encrypting only the file names, we encrypt the entire row (including the TFIDF statistics) with the barrel encryption key

BEK_i. When the SSP receives a search query with keyword $KH_{BEK_i}(w)$ from a user u, it returns the encrypted rows in the index corresponding to the keyword $KH_{BEK_i}(w)$. Now, the client has to perform some computation to decrypt and merge the results obtained from different barrels and present a final ranked list of files to the user. This approach preserves the privacy of the index statistics along with file names and vocabulary by incurring some computation and communication overhead at the client.

3.4.4 Handling Updates

In an enterprise environment, there will be regular updates to data files and access privileges to data and the system needs to handle them appropriately. In the decentralized storage service model, these updates could occur through multiple clients and would require modifying indices stored at the SSP. As in xACCESS, we expect this process to be little expensive since we are not only modifying the data, but also "*embedding*" access control into it.

There are two options for this modification:

- *Immediate Client Modification*: In this scheme, the client that updates a file is responsible for updating the barrel index that contains the file. Most indexers like [112] can handle index updates in an incremental manner and not performing complete re-indexing. This approach has a big drawback of the impact on overall performance of data writes (requires modifying barrel indices) and the fact that the responsibility of this modification lies on individual enterprise users. Thus, we prefer the lazy approach described next.

- *Lazy Orchestrator Modification*: In the second scheme, the global orchestrator is responsible for handling all updates. All clients that update a file, add its name to a *barrel update list* stored at the SSP. The barrel update lists are encrypted with the barrel encryption key, thus ensuring that only users that have access to files in that barrel can modify the update list. The enterprise orchestrator periodically updates indices for all modified files. This approach eliminates index modification responsibility from clients at the risk of maintaining *old* indices.

Another operation requiring modification to indices is the case of user/group membership modification, in which case the access hierarchy needs to be adjusted. A user/group addition is handled by adding a new node and corresponding edges (as done during initial *ACG* construction). Group membership modification is handling by changing the edges in the directed graph. Finally, a user/group deletion is handled by removing the appropriate node and all edges coming into or out of that node and possible re-encryption of barrels. Since this operation is performed only by the `root`, the global orchestrator is responsible for performing these operations.

3.5 Experimental Evaluation

In this section, we present a detailed evaluation of our approach. In Section-3.5.1, we describe the datasets used in our indexing and querying experiments. Section-3.5.2 describes the indexing experiments including barrels pre-processing and Section-3.5.3 describes the querying and search related experiments. We compare our approach with the other ACAS-compliant index-per-user approach that allows hosting indices at the SSP in Section-3.5.5. Section-3.5.6 shows the effectiveness of our optimization algorithms.

All experiments were done on a Pentium-III Linux machine with 512 MB RAM and all storage mounted via NFS and results have been averaged over multiple runs.

3.5.1 Datasets

The first data set, called T14m, is a publicly available cleaned subcollection [78] of TREC Enterprise track (TREC 14) [176]. TREC 14 is a newly formed track specifically on enterprise search and includes data from the World Wide Web Consortium (W3C) enterprise filesystems. The T14m dataset characteristics are shown in Table-12. It includes emails (*lists*), web pages (*www*), wiki web pages (*esw*) and people pages (*people*). This dataset does not include any access control information.

A significant portion of the efficacy of our approach depends on actual filesystem structure and access privileges in the enterprise. In order to measure this, we collected statistics from a real multiuser enterprise installation, whose characteristics are shown in Table-13. Barrels per user statistics were also computed at a second enterprise (shown by *). It shows

Table 12: T14m dataset: Cleaned TREC 14 subcollections

Scope	Docs	Size	Avg. Doc Size
lists	173,146	485 MB	2.9 KB
www	45,975	1001 MB	23.8 KB
esw	19,605	80 MB	4.2 KB
people	1,016	3 MB	3.1 KB
Total	239,742	1569 MB	6.9 KB

the low average number of barrels per user which implies that our core ACB approach will function well (without any optimization tradeoffs).

We collected anonymized directory structure and access privileges information for 339,466 files arranged in 23,741 directories and replicated the structure in our test environment. The T14m data was used as content for the files (duplicating documents to fill all 339,466 files).

Table 13: Real Enterprise Dataset

Number of users	926		
Number of user groups	1203		
Number of files	339,466		
Number of dirs	23,741		
Max depth of dir structure	23		
Size of data	2.05 GB		
Number of barrels	2132		
Barrels per user	Max	Avg	Median
	25	6.31	4.26
	21*	5.78*	3.96*

3.5.2 Indexing Experiments

Indexing is perhaps the most important component of our approach. It includes a pre-processing step that creates the user access hierarchy and the access control barrels followed by actual content indexing of the files contained in ACBs.

3.5.2.1 Pre-processing

As pre-processing performance is entirely dependent on the enterprise infrastructure (users/groups and directory structure), we use the real enterprise dataset for these experiments. Table-14 shows the evaluation of our implementation.

Creating access hierarchy for 926 users and 1203 user groups took a total of 38.7 seconds,

Table 14: Pre-processing Performance for real enterprise dataset

Task	Performance
Access hierarchy creation	38.7 sec
Barrel creation	263.1 sec
# Files `stat`'ed	202,446 (60%)
# Dirs `stat`'ed	14,059 (59%)

which is a very small fraction of the total indexing time. It took another 263.1 sec to traverse the filesystem and create all ACBs. One $\wedge$-credential needed to be added to the access hierarchy during barrel creation. Additionally only 60% of the filesystem tree needed to be traversed to create all barrels as in many cases, a higher level directory was mapped to a restrictive credential (e.g. only uid_{bob} can access) in which case its contents are automatically added to that barrel without deeper traversal. Overall, these costs are only 10% of the distributed indexing approach and 6% of the centralized approach (shown later in Table-15).

3.5.2.2 Content Indexing

For indexing of documents, we used the *arrow* indexing and search component of the Bow Toolkit [112] developed at CMU. We modified the ranking algorithm of the toolkit to the distributed IR algorithm of [104]. For our experiments, we compared two architectures: (1) *Centralized Single Index (CSI)* - a centralized single index for the entire dataset analogous to the available enterprise search products[3], and (2) *Barrel-based Distributed Indexing (BDI)* - our barrels based distributed indexing approach. *BDI-m* denotes the case when m machines are used to index barrels in parallel. The Index-Per-User (IPU) approach is similar to the BDI approach as it also allows distributed indexing. We perform a direct comparison with the IPU approach later in Section-3.5.5.

Figure-28 shows the time to index different number of documents of the T14m dataset ranging from 25K to 240K. For BDI architectures, the documents were equally divided between the participating machines for indexing.

From the graph, CSI outperforms the BDI approaches when the number of documents is

[3]Recall that this architecture does not guarantee access control aware search

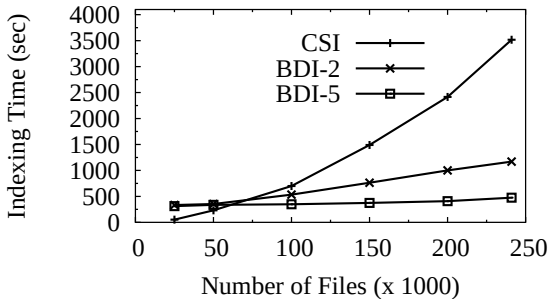


Figure 28: Indexing T14m dataset

small. This occurs due to the pre-processing costs that the BDI approaches incur. However, as the number of documents increases, BDI approaches quickly outperform the CSI approach. The distribution of data into ACBs has allowed us to exploit available enterprise machines for faster indexing time (an 85% improvement for 240K files).

The results for the T14m dataset above are a little optimistic as it considers a uniform size and distribution of barrels. However, in reality there could be a few barrels that are significantly larger than the others and dominate the indexing times. To evaluate this, we performed indexing for our real enterprise dataset. The results are reported in Table-15, where #Max-Docs is the number of documents in the largest barrel. As shown in Table-15, the barrel for the `all` node (files that can be read by all users) was significantly larger and took longer than all other barrels combined (and thus total time does not vary with number of machines). However, it was still 38% more efficient than the CSI approach. In general, distribution is most helpful when there are many such large barrels and we expect that to be true in an enterprise-scale environment.

Table 15: Indexing for real enterprise dataset

Type	#Max-Docs	Time (s)	Savings
CSI	339,466	4640	–
BDI	189,546	2902	38%

3.5.3 Searching Experiments

If the indices are hosted at the SSP, search would actually be performed by the SSP. However, as we are comparing our approach to the centralized index approach (CSI) scheme,

which does not support SSP index-hosting, we consider these experiments in the local enterprise setup. We evaluate the SSP index hosting approaches in Section-3.5.4.

Recall that searching in our approach requires combining multiple barrels. However, given the small number of barrels per user, overheads should not significantly deteriorate query performance. Secondly, since our approach does not have to perform access control at runtime (which the CSI approach does), there would be some savings in query runtime performance.

For the querying experiments we used 150 queries obtained from TREC 14 Email search. The queries had an average of 5.35 terms per query. The results for CSI and n-BDI (where n is the number of barrels combined) are reported in Table-16, where loading time is the time to load all indices in memory.

Table 16: Search Performance for TREC 14 *lists*

Type	Index size	Loading time	Avg. time / query
CSI	230 MB	2.5 s	131.12 ms
2-BDI	258 MB	3.37 s	112.89 ms
5-BDI	269 MB	5.68 s	130.68 ms
10-BDI	280 MB	6.90 s	149.90 ms

First notice that the BDI approaches have slightly larger indices. This is due to the fact that they have to store many words multiple times in different barrel vocabularies. Next, the time to load indices into memory also increases with the number of barrels as there are more file I/Os to gather the index data. However, this is only a one-time cost and once indices are cached, queries proceed normally. Finally, the average query time for BDI approaches is comparable to CSI with 2-BDI and 5-BDI even outperforming it by saving on the privileges check required at runtime in the CSI approach.

We also compared the ranking of the BDI approaches to CSI ranking. For this we evaluated the percentage of top-10 results of the CSI approach that occurred in top-100 of the distributed approach and their average ranks. Table-17 reports the results, where *10-in-100* is the percentage of CSI top-10 results in top-100 of x-BDI and *avg-rank* is the average rank of CSI top-10 results in x-BDI top-100. For our average case of 5 barrels per user, nearly 70% of top-10 results occurred in top-100 of the BDI approach with an

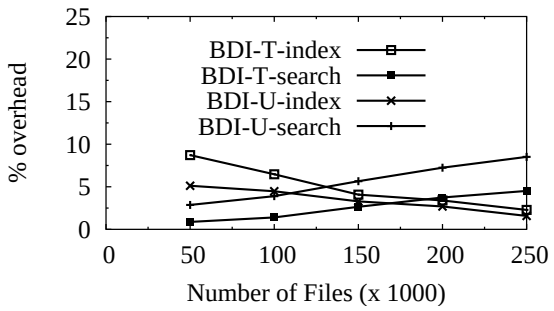


Figure 29: Computation Costs for SSP-hosted indices

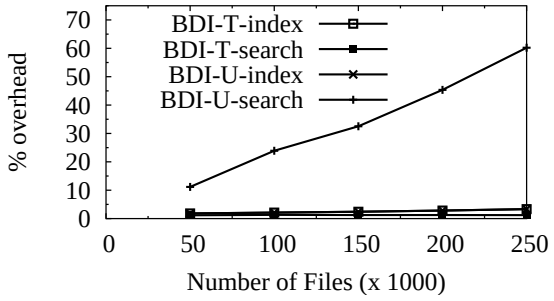


Figure 30: Communication Costs for SSP-hosted indices

average rank of 14. We believe that ranking can be further improved using more advanced distributed ranking techniques.

Table 17: Ranking comparison for TREC 14 *lists*

Type	10-in-100	Avg. Rank
2-BDI	75%	13
5-BDI	68%	14
10-BDI	61%	15

3.5.4 SSP Search Evaluation

In our SSP experiments, we compare the two approaches discussed in Section-3.4.3 – (a) BDI-T stores indices at the SSP with un-encrypted index statistics (TFIDF measures) and only hides file names and words, (b) BDI-U also encrypts the TFIDF statistics, but incurs higher communication and computation costs.

Figure-29 and 30 show the computation and communication overhead incurred by the two approaches over an approach wherein the SSP is completely trusted (nothing is hidden – used only for a baseline comparison).

The indexing cost for BDI-T is higher since we encrypt each filename separately, unlike BDI-U which encrypts the entire list of file names and index statistics for each keyword (encrypting file names separately requires each file name to be padded such that its length is an integer multiple of 16 bytes, a requirement of the encryption algorithms).

On the other hand BDI-U incurs a higher cost for search. Computation cost is higher because BDI-U requires client side computation to decrypt and merge the results from multiple barrels. Communication cost is high because in BDI-U the SSP sends the entire list of file names for a keyword (along with the index statistics) to the client rather than sending a short-listed set of ranked files. Note that the total number of files that match a keyword can be significantly larger than the result set, hence, the communication overhead in BDI-U is also larger than BDI-T.

We believe that the choice of approach would be a security based decision. In the BDI-T approach, the range of attacks are probabilistic frequency inference attacks which can be thwarted using techniques like multiple SSPs [165]. Thus, enterprises have to individually trade off this security issue with computation and communication costs with latter influencing monetary costs as well.

3.5.5 Comparison with Index Per User Approach

In this section, we compare the performance and scalability of the ACB approach against the index per user (IPU) approach. For a wider range of experiments and to better analyze the impact of large number of files and users, we use synthetic data for these experiments. The key parameters in our data are summarized in Table-18. Since the analysis is the same for both $\wedge$-groups and $\vee$-users, we consider only $\wedge$-groups in this section. We use Zipf(a, b) to denote a Zipf distribution with parameter $\gamma = 1$ that is truncated to the range (a, b).

We choose ngu the number of groups a user is a member of using a Zipf distribution on the range $(2, 10)$; we then choose ngu groups from the set G using Zipf(1, $|G|$) and without replacements. Similarly, we choose ngf the number of $\wedge$-groups per file using Zipf(2, 4); we then choose ngf groups from the set G using Zipf(1, $|G|$). The access control rule for the file is assumed to be an $\wedge$ over all the chosen groups. The number used in these experiments

Table 18: Parameters for IPU-ACB Comparison

Notation	Description	Default
$\|F\|$	Number of Files	10^7
$\|U\|$	Number of Users	10^3
$\|G\|$	Number of User Groups	32
pop	Group Popularity	Zipf(1, ng)
ngu	Number of Groups per User	Zipf(2, 10)
ngf	Number of $\wedge$-groups per File	Zipf(2, 4)
nuf	Number of $\vee$-users per File	Zipf(2, 4)

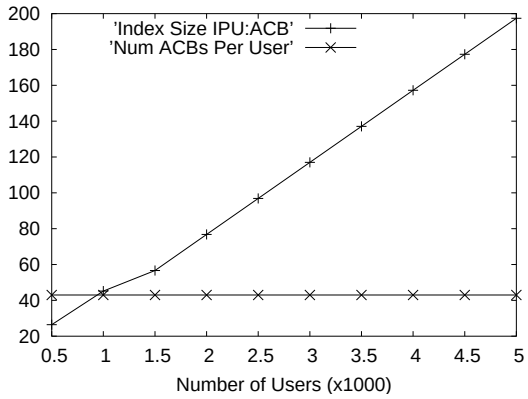


Figure 31: IPU-ACB Scalability Comparison: # Users

are based on typical enterprise infrastructure as observed earlier.

For our evaluation, we measure the total number of ACBs and the number of ACBs per user. We also compare the ratio of the size of indices maintained by the IPU and ACB approach. Figure-31 shows the scalability of our approach with the number of users $|U|$. In the IPU approach, the index size grows with the number of users in the system, typically because more users share a file. On the other hand, the ACB approach maintains exactly one copy of each file index and the total number of barrels (and thus the average number of barrels per user) is *independent* of $|U|$. Figure-32 shows the scalability of our approach with ngu the number of group memberships per user. As ngu increases, so does the number of users that share a file and thus the index size in the IPU approach. In the ACB approach, as ngu increases it results in a smaller increase in the number of ACBs per user.

One should observe that our core approach maintains exactly one copy of index for each file. Hence, when a file is updated at most one index needs to be modified. The IPU approach maintains multiple copies of each file index, one for every user who is permitted

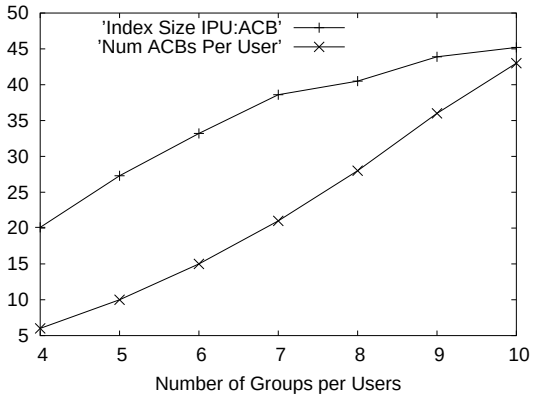


Figure 32: IPU-ACB Scalability Comparison: # Groups per user

to access that file. Hence, when a file is updated the IPU approach has to update several indices (an average of 45.2 using default settings in Table 18) thereby incurring high data access costs. In contrast the ACB approach incurs a small overhead of using distributed IR solutions to merge search results from a small number of barrels.

3.5.6 Optimization Techniques

In this section, we show the effectiveness of our optimization techniques in decreasing the number of barrels. Figure-33 shows the effectiveness of our first optimization technique that preserves the ACAS property while maintaining multiple copies of each file index. We plot the tunable parameter $minf$, the minimum number of files per barrel, on the x-axis. As described in Section-3.3.3 our index size increases with $minf$ and slowly reaches the index size of the IPU approach as $minf \rightarrow |F|$. This shows the flexibility of our technique in reducing the number of barrels while incurring significantly lower costs than the IPU approach.

Figure-34 and Figure-35 show the effectiveness of our second optimization technique that maintains exactly one copy of every file index while violating the ACAS property for some files. We have evaluated the effectiveness of our algorithm using three heuristics: `random` chooses a vertex v at random from $maxSub(V_C)$, `size` picks the vertex $v \in maxSub(V_C)$ that has the least number of files, and `pop` picks the vertex $v \in maxSub(V_C)$ that causes the least number of ACAS violations. Figures 34 and 35 show that popularity based approach performs best in terms of both minimizing the number of violations and the number of

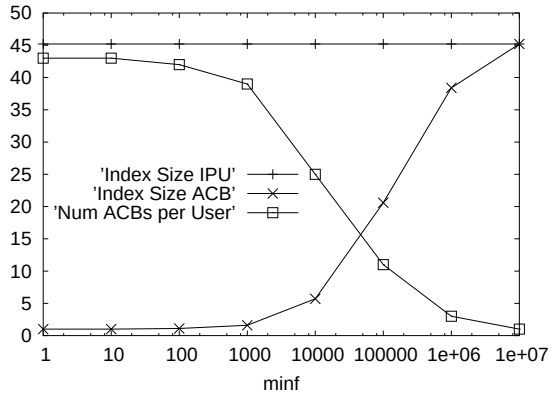


Figure 33: Optimization I: Reducing #ACBs by File Duplication

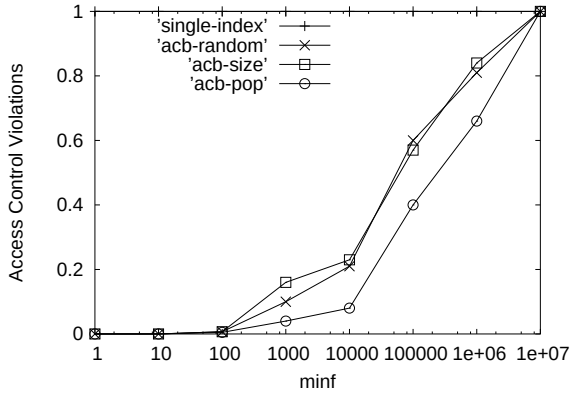


Figure 34: Optimization-II: Comparing Index Size

ACBs. As described in Section-3.3.3 the number of violations increases with $minf$ and finally equals that of the single-index approach. This shows the flexibility of our technique in reducing the number of barrels while violating the ACAS property for far fewer files than the single-index approach.

3.6 Related Work

3.6.1 Other Enterprise Search Approaches

We have already described existing work in enterprise search (Section-3.2) and how our approach tackles the problem differently. Table-19 summarizes this discussion and compares all approaches on various attributes. Specifically, our approach is unique in its ability to provide access control aware search using a distributed approach that allows index hosting at an untrusted SSP.

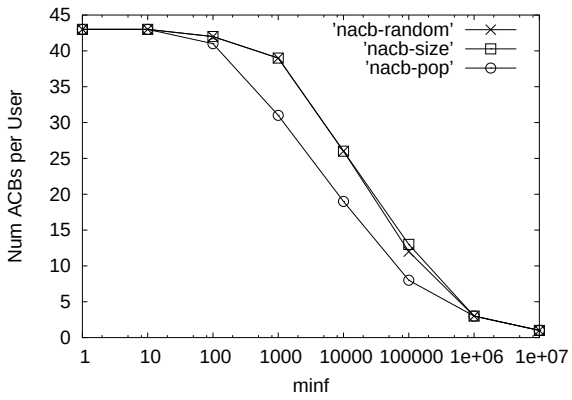


Figure 35: Optimization II: Comparing # ACBs

Table 19: Comparison of search approaches

	Desktop Search	**Enterprise Search**	**Wumpus** [28]	**Our Approach**
ACAS Requirement	Satisfied	Not satisfied	Satisfied	Satisfied
Architecture	Distributed & parallel	Centralized	Centralized	Distributed & parallel
Access Control Integration	Indexing	Query Runtime	Query Runtime	Indexing
Service Provider Index Hosting	No	No	No	Yes
Overheads	High space & update costs	Runtime privileges check	Runtime query transformation & privileges check	Barrels processing & runtime results merging

In the rest of this section, we discuss other related work in private search and keyword based search over encrypted data, similar to our SSP environment.

3.6.2 Private Information Retrieval

Private information retrieval (PIR) was first introduced as a problem by Chor et al [39] – a user wishes to retrieve the i^{th} bit in a database without revealing any information about i. PIR schemes often require multiple non-colluding servers, operate in multiple rounds, are resource-intensive and do not support keyword search. Hence, several authors have focused on efficient solutions and their security guarantees. Another direction of work has focused on running queries over encrypted data at an untrusted server [73, 169, 34]. These schemes require the user to know a secret key with which the searchable content of the document is

encrypted. They ensure that only the frequency profile of the queried keywords is revealed to the search service provider (similar to our BDI-T approach). However, these approaches do not consider a multiuser enterprise setting where in addition to keeping the data private from the SSP, one needs to enforce access control rules on the users. Our approach cleverly partitions the search problem into two parts: an access control problem that is handled by our barrels-based secure indices, and a privacy problem if such indices are hosted by a third-party search service provider. Indeed, we can leverage any approach [169, 21] that provide privacy preserving search over an untrusted service provider hosted index.

Bawa et al [16] present techniques for constructing a privacy preserving index on documents in a multi-organizational setting. Their goal is to construct a centralized index that can be made public without giving out any private information. Similar to other enterprise search techniques they apply access control at query runtime and incur higher overheads than our proposal. Our approach focuses on integrating access control with search in a single enterprise setting and is more efficient.

3.7 Summary

In this chapter, we presented an efficient and secure approach to enterprise search. This is a desired feature for data stored at a remote Storage Service Provider which charges client enterprises for the amount of data they access. We demonstrated the inadequacy of existing solutions at ensuring access control aware search for multiuser enterprise environments. We developed distributed techniques that elegantly "*embed*" access control semantics into search indices, using novel *access control barrel (ACB)* and *user access hierarchy* concepts. The distributed and parallel nature of our solution helps improve indexing efficiency and allow search indices to be hosted at the SSP. Our experimental evaluation on synthetic and real datasets shows improved indexing efficiency and minimal overheads for ACB processing.

CHAPTER IV

EFFICIENT IMPACT ANALYSIS WITH ZODIAC

The first part of the thesis addressed client security and privacy concerns in the storage-as-a-service model. As discussed earlier, another important challenge to this model stems from the complex management of the highly dynamic storage infrastructure of the service provider. The storage service provider has to provide on-demand storage and needs to accommodate client requests quickly and without adversely impacting the rest of the environment.

To scale to large amounts of data, SSPs use a Storage Area Network (SAN), typically based on Fibre Channel SCSI technology [43]. This SAN is hidden from client enterprises, thus preventing any management complexity at their end. Inside the SSP however, a number of administrators are required for managing such SANs, performing complex tasks like change analysis, provisioning, performance bottleneck analysis, capacity planning, disaster recovery planning and security analysis. In the context of an on-demand SSP infrastructure, one of the most critical tasks becomes change analysis, which is to pro-actively analyze the impact of an upcoming change on the rest of the storage area network. This is important since it is estimated that over 70-80% of all changes resulting in downtimes are initiated by people within the organization [114]. Such downtimes can be crippling for a storage service provider.

Typically, administrators perform change impact analysis manually, based on their past experience and rules of thumbs (best practices). For example, when a new host is added, the administrators make sure that Windows and Linux hosts are not put into the same logical zone or while adding a new workload, they ensure that the intermediate switches do not get saturated. Manually analyzing the impact of a particular change does not scale well as the size of the SAN infrastructure increases with respect to the number of devices, best practices policies, and number of applications.

Most change management tools have been reactive in their scope in that they keep snapshots of the previous state of the system, and the administrators either revert to or compare the current state with a previous state after encountering a problem. Only recently has proactive change management analysis begun to receive its share of deserved attention [157, 91, 97, 174]. In this chapter we describe our contribution in this area – the **Zodiac** framework. Zodiac enables system administrators to proactively assess the impact of changes on a variety of system parameters like resource utilizations and existing system policies, before making those changes.

The key aspect of our analysis framework is that it is tightly integrated with policy based storage management [3]. Policy-based management is being incorporated into most vendor's storage management solutions as it allows administrators to specify high level goals and requirements and let the management software handle implementation details. Zodiac allows administrators to specify their rules of thumb or best practices with respect to interoperability, performance, availability, security as *policies*. It then assesses the impact of user actions by checking which of these policies are being violated or triggered. Zodiac also assesses the impact of creating new policies.

We have also developed a number of optimizations that help to reduce the amount of SAN data examined during impact analysis, ensuring quick response and scalability for large SANs and large number of policies. One of the interesting results of the optimization design effort in Zodiac is that we have designed a new method for classifying SAN policies based on the optimization techniques they employ. This, in turn, can also be used by general SAN policy evaluation engines to optimize their evaluation mechanisms. During policy specification, policy designers can specify the policy type (as per this classification) as a hint to the policy engine to optimize its evaluation.

The rest of the chapter is organized as follows. Section-4.1 provides the necessary background with respect to policy definitions, and SAN operations. The overview of our architecture is presented in Section-4.2 followed by the details of our implementation in Section-4.3. In Section-4.4, we discuss three important optimization algorithms that help speed up the

overall analysis process. The experimental framework and the results evaluating these optimizations are presented in Section-4.5. We describe related work in Section-4.6. Finally, we summarize our contributions in Section-4.7.

4.1 Background

This section presents the necessary background material for this chapter. Section-4.1.1 contains a discussion on the type of SAN policies considered in this chapter. Section-4.1.2 provides the details of the storage resource models and Section-4.1.3 presents a list of what-if operations that one can perform.

4.1.1 Policy Background

The term *policy* is often used by different people in different contexts to mean different things. For example, the terms *best practices, rule of thumbs, constraints, threshold violations, goals, rules and service classes* have been referred to as policies by different people. Currently, most standardization bodies such as IETF, DMTF, and SNIA refer to policy as a 4-field tuple where the fields correspond to an *if* condition, a *then* clause, a *priority or business value* of the policy and a *scope* that decides when the policy should be executed. The *then* clause portion can generate indications, or trigger the execution of other operations (*action* policies), or it can simply be informative in nature (write a message to the log). [3] describes the various SAN policies found relevant by domain experts. In this chapter, within the storage area network (SAN) domain, we deal with the following types of policies:

- **Interoperability:** These policies describe what devices are interoperable (or not) with each other.
- **Performance:** These policies are violation policies that notify users if the performance of their applications (throughput, IOPs or latency) violate certain threshold values. For a storage service provider this implies the performance guarantees given to the client enterprise.

- **Capacity:** These policies notify users if they are crossing a percentage (threshold) of the storage space has been allotted to them.

- **Security and Access Control:** Zoning and LUN masking policies are the most common SAN access control policies. Zoning determines a set of ports of host, switches and storage controllers that can transfer data to each other. Similarly, LUN masking controls host access (via its ports) to storage volumes at the storage controller.

- **Availability:** These policies control the number of redundant paths from the host to the storage array.

- **Backup/Recovery:** These policies specify the recovery time and recovery point objectives and copy frequency to facilitate continuous copy and point-in-time copy solutions.

4.1.2 Storage Resource Model

In order to perform impact analysis, storage resource models are used to model the underlying storage infrastructure. A storage resource model consists of a schema corresponding to various **entities** like hosts, host bus adapters (HBAs), switches, controllers, the entity **attributes** (e.g. vendor, firmware level), container relationships between the entities (HBA is contained with a host), and connectivity between the entities (fabric design). These entities and attributes are used during the definition of a policy as part of the *if-condition* and the *then clause*. For a specific policy, the entities and the attributes that it uses are called its *dependent* entities and *dependent* attributes respectively. The SNIA SMI-S [168] model presents a general framework for naming and modeling storage resources.

In addition to the schema, a storage resource model also captures the behavioral aspects of the entities. The behavioral aspects, called **metrics**, represent how a resource behaves under different workload and configuration conditions. The behavioral models are either analytically specified by a domain expert [178], or deduced by observing a live system [7] or a combination of both.

The defined resource model gets populated by discovering and monitoring a real SAN.

SMI-S utilizes the client-server CIM architecture [50] to allow for the retrieval of storage management information from the storage resources. In this chapter we assume that we have a CIM client that populates a SMI-S compliant SAN resource model database.

Figure-36 shows the basic SAN resource model that we consider in this chapter. A single SAN path is highlighted and shaded ports represent zone containment. Our resource model consists of hosts, HBAs, ports, switches, storage controllers, zones, and volume entities, and host to HBA, port to HBA, port to zone containment relationships and port to port connection relationships. In addition, there exists a *parent entity class* called *device*, which contains all the SAN devices. The *device* entity class can be used to define global policies like *all devices should have unique WWNs.*

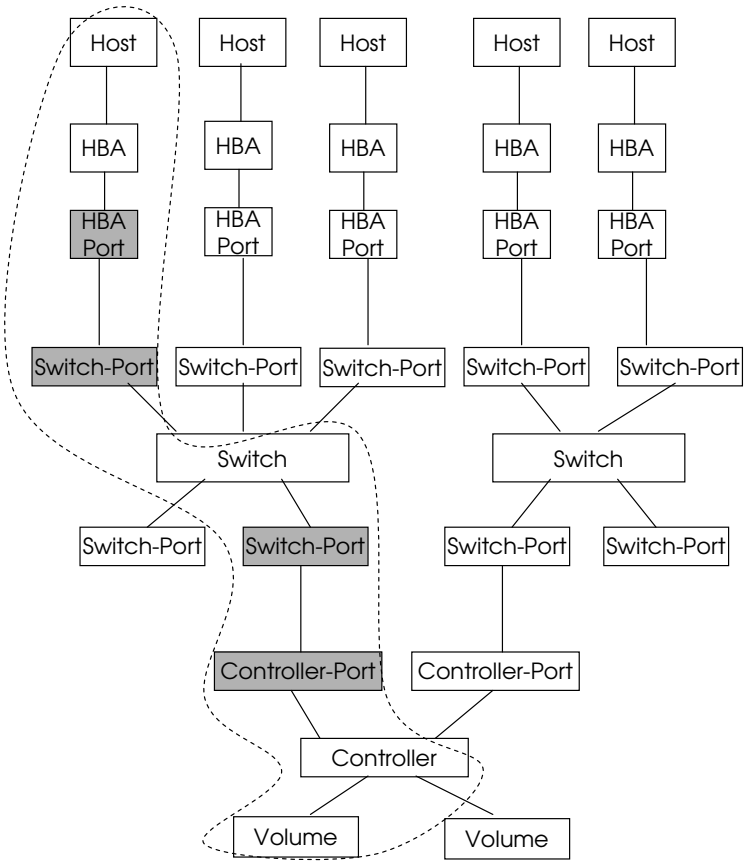


Figure 36: SAN Resource Model

4.1.3 SAN Operations:

Using Zodiac, the following types of operations can be analyzed for impact.

- *Addition/Deletion of Physical Resources*: Hosts, switches, HBAs, and storage arrays can be added or removed (due to failure). Similarly, devices can be interconnected or disconnected.

- *Addition/Deletion of Logical Resources*: Volumes and zones can be created and removed. Ports belonging to storage devices can be made active or inactive.

- *Access control operations*: Adding or removing ports to zones to control which host can access which storage ports. Similarly, LUN masking operations can be performed to determine which host ports can access which storage volumes.

- *Addition/Deletion of Workloads*: The requirements (throughput, latency) of a workload can also be modified. We represent a workload as a set of flows. A flow can be thought of a data path (Figure-36) between a host and a storage controller. A flow starts at a host port, and goes through intermediate switch ports and ends at a storage controller port.

- *Addition/Deletion of Policies*: Zodiac can also evaluate the impact of adding or deleting a particular policy. However, please note that we do not perform conflict detection analysis within policies, rather only simulate how the real system is likely to behave.

4.2 Architecture Overview

In this section, we provide an overview of the Zodiac architecture and its interaction with other SAN modules. We start with an example illustration of the impact analysis process.

4.2.1 Impact Analysis Illustration

Consider an administrator trying to deploy 10 hosts, running a new application, to the SAN. The SAN already has many critical applications deployed, with stringent performance requirements. In addition, the SAN is also required to adhere to many best practices (e.g. *all hosts should be from the same vendor*) and lab tested configuration guidelines (*Vendor-A host with Vendor-H HBA works well with Vendor-S storage*). These and many other policies are stored in a policy database. Before applying the actual change, the administrator wishes

to use Zodiac to analyze the impact of the change on existing infrastructure. The various steps in the impact analysis process are as follows:

- First, the administrator would initialize a Zodiac impact analysis **session**, in which operations can be "virtually" applied to the SAN to analyze their impact. Zodiac starts off by taking a snapshot of the state of the SAN, which is used as the base state for impact analysis.

- As part of the analysis, Zodiac would first **identify the set of relevant policies** that need to be checked before adding the hosts. For example, it would check if the hosts' vendor is the same as the existing hosts in the SAN and if the host connections are consistent with the lab configuration guidelines.

- It is possible that some of the policies get violated. Based on the actions defined in the *then* clause of those policies, Zodiac could just indicate the violation (policy only required notifying the administrator), or **perform automatic operations** (policy initiates a migration job, on violation).

- These policy actions are treated as basic SAN operations (Section-4.1.3) and analyzed similarly. Such "chaining" actions might require many **more policies to be checked**. For this reason, Zodiac incorporates many optimization algorithms that let it efficiently recognize the set of relevant policies from the huge policy database and the region of SAN relevant for evaluation (Section-4.4).

- After initial policy evaluation, Zodiac **simulates the new network traffic** generated due to the workload and evaluates new metrics for the SAN resources using device resource models. Such evaluation could also trigger various performance policies and their actions are also analyzed for impact.

- The user can also analyze the impact at a **future point in time**. For example, if all storage is backed up at 3 AM, the administrator might want to ensure that the workloads are able to meet their performance requirements during that time.

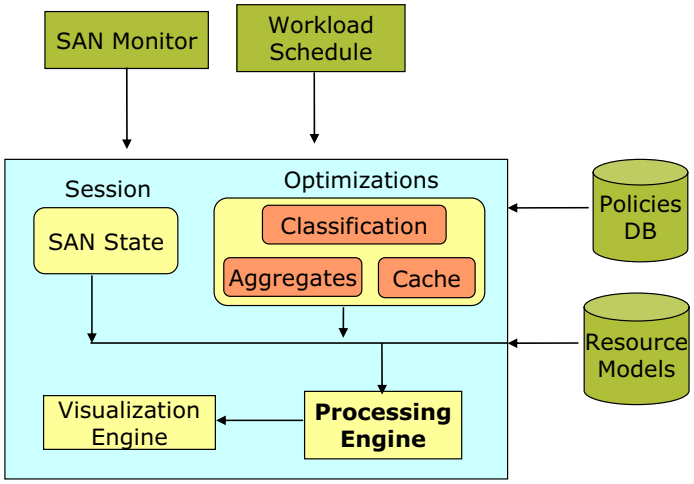


Figure 37: Zodiac Architecture

As evident from the example, impact analysis is a complex task and policy evaluation is a very **critical** component in its performance (multiple policies can be triggered multiple times in a single impact analysis operation). We believe Zodiac to be an important step in solving this tough problem.

4.2.2 Zodiac: Big Picture Design

The goal of the impact analysis engine, like Zodiac, is to predict the state and behavior of the SAN once a desired operation is performed. In order to evaluate the new state, the engine needs to interact with various SAN modules to get the relevant information, like device attributes, policies. The overall picture of such an eco-system is shown in Figure-37. In this eco-system, Zodiac interacts with the following modules:

- *SAN Monitor:* The foremost input requirement is the state of the SAN, which is obtained from a SAN Monitor like [54, 81, 88, 45]. It consists of the physical configuration (fabric design), resource attributes (HBA vendor, number of HBAs in a host) and logical information like Zoning/LUN-Masking.

- *Workload Schedule:* In order to predict the behavior of the SAN, Zodiac also needs to know the schedule of the workload. For example, if a backup job is scheduled for 3 AM, then the engine needs to account for the additional traffic generated due to the backup during that duration.

- *Policy Database:* A unique characteristic of the Zodiac impact analysis engine is its integration with policy based management. The policies are specified in a high level specification language like Ponder [49] or XML [3] and stored in a policy database.

- *Resource Models*: As described earlier, Zodiac uses a model based approach to evaluate the behavior of SAN resources. For this, we require a resource models database that provides such behavioral models. There has been significant work in the area of modeling and simulation of SAN resources [178, 7, 153, 189, 44, 121, 17, 26, 194] and we leverage that. Note that the design of Zodiac is independent of the resource models and can work with any approach.

Given these modules, Zodiac takes as input the operation that the administrator wants to perform and the time at which the impact needs to be measured (immediate or after n hours) and initiates the analysis.

4.2.3 Zodiac: Internal Design

Internally, Zodiac engine is composed of the following primary components:

- **SAN-State:** In Zodiac, the impact analysis occurs in a *session*, during which an administrator can analyze the impact of multiple operations incrementally. So, a first operation could be - *what happens if I add two hosts?* After the engine evaluates the impact, an administrator can perform an incremental operation - *what if I add* another *two hosts?* The SAN state component maintains the intermediate states of the SAN, so that such incremental operations can be analyzed. When an analysis session is initialized, the SAN state is populated by the current snapshot of the SAN, obtained from the SAN Monitor.

- **Optimization Structures:** As mentioned earlier, for efficient policy evaluation, Zodiac maintains intelligent data structures that optimize the overall evaluation. These three primary structures (caches, policy classes and aggregates) are explained in detail in Section-4.4.

- **Processing Engine:** The processing engine is responsible for efficiently evaluating the impact of operations using the SAN state and the rest of the internal data structures. It is the main work horse of Zodiac.

- **Visualization Engine:** Another important component of Zodiac is its visualization engine. The visualization engine primarily provide two kinds of output. First, it can provide an overall picture of the SAN, with various entity metrics and can highlight interesting entities, e.g. the ones that violated certain policies. Secondly, with the incorporation of temporal analysis, an administrator can plot interesting metrics with time.

4.2.4 Operation Modes

Zodiac uses a number of internal data structures, that are used to optimize impact analysis. These data structures are derived from the actual SAN and it is important to keep them consistent with the state of the SAN across multiple impact analysis sessions. This is facilitated by Zodiac via the following modes of operation:

- *SAN Management Software (SMS):* The ideal setting for an impact analysis engine is for it to be a part of the SAN management software. In this mode, the internal data structures are automatically updated by the management software and thus, no special operations are required.

- *Distinct Component with Bootstrapping (DCB):* Another mode could be the generation of all required data structures, every time the analysis engine is run. Though this makes the bootstrapping process expensive, it keeps the impact analysis engine completely independent of the SAN management software.

- *Distinct Component with Event Listener (DCEL):* It is also possible for the impact analysis engine to contain an event listener which acts as a "sink" for events generated by the SAN (standardized under SNIA SMI-S specification) and keeps its structures updated. This would require the analysis engine to be running at all times.

Our current prototype is implemented to operate in the SMS mode, though the design does not have any inherent restrictions and can be modified to operate in any mode.

4.3 Zodiac: System Details

In this section, we provide the details about the internal data structures being used by Zodiac to represent SANs (Section-4.3.1) and how the policy evaluation framework uses these data structures (Section-4.3.2). In Section-4.3.3, we describe the inadequacy of the current evaluation approach before proposing various optimizations in the next section.

4.3.1 SAN Representation

For efficient impact analysis, it is critical that SAN is represented in an optimal form. This is because all policies and resource metric computations would obtain required data through this SAN data structure. In Zodiac, the SAN is represented as a graph with entities as nodes and network links or containment relationships (HBA is contained within a host) as edges. A sample SAN as a graph is shown in Figure-36. A single SAN "path" has been highlighted. Note that it is possible to have more than one switch in the path.

Each entity in the graph has a number of attribute-value pairs, e.g. the host entity has attributes like vendor, model and OS. In addition, each entity contains pointers to its immediate neighbors (Host has a pointer to its HBA, which has a pointer to its HBA-Port and so on). This immediate neighbor maintenance and extensive use of pointers with zero duplication of data allows this graph to be maintained in memory even for huge SANs (1000 hosts).

There are two possible alternatives to this kind of immediate-neighbor representation of the SAN. We discuss the alternatives and justify our choice below:

1. **Alternative-Paths**: Assume a best practices policy requiring a Vendor-A host to be only connected to Vendor-S controller. Its evaluation would require a traversal of the graph starting at the host and going through all paths to all connected controllers. In fact many policies actually require traversals along "paths" in the graph [3]. This could indicate storing the SAN as a collection of paths and iterating over the relevant paths

for each policy evaluation, preventing costly traversals over the graph. However, the number of paths in a big SAN could be enormous, and thus, prohibitive to maintain the path information in-memory. Also, the number of new paths created with an addition of a single entity (e.g. a switch) would be exponential, thus making the design unscalable.

2. **Alternative-SC**: Even without paths, it is possible to "short-circuit" the traversals by keeping information about entities further into the graph. For example, a host could also keep pointers to all connected storage. While this scheme does work for some policies, many interoperability policies, that filter paths of the graph based on some properties of an intermediate entity, cannot be evaluated. For example, a policy that requires a Vendor-A host, *connected to a Vendor-W switch*, to be only connected to Vendor-S storage, cannot be evaluated efficiently using such a representation, since it is still required to traverse to the intermediate entity and filter based on it. However, this idea is useful and we actually use a modified version of this in our optimization schemes described later.

4.3.2 Policy Evaluation

In current policy evaluation engines, policies are specified in a high level specification language like Ponder [49], XML [3]. The engine converts the policy into executable code that can evaluate the policy when triggered. This uses an underlying data layer, e.g. based on SMI-S, that obtains the required data for evaluation. It is this automatic code generation, that needs to be heavily optimized for efficient impact analysis and we discuss various optimizations in Section-4.4.

In Zodiac, the data is obtained through our SAN data structure. For evaluating a policy like *all Vendor-A hosts should be connected to Vendor-S controllers*, a graph traversal is required (obtaining storage controllers connected to Vendor-A hosts). In order to do such traversals, each entity in the graph supports an API that is used to get to *any* other connected entity in the graph (by doing recursive calls to immediate neighbors). For

example, hosts support a *getController*() function that returns all connected storage controllers. The functions are implemented by looking up immediate neighbors (HBAs), calling their respective *getController*() functions, aggregating the results and removing duplicates. The neighbors would recursively do the same with their immediate neighbors until the call reaches the desired entity (storage controller). Similarly for getting all connected edge switches, core switches or volumes. This API is also useful for our caching optimization. It caches the results of these function calls at all intermediate nodes for reuse in later policy evaluations.

However, even this API suffers from the limitation of the Alternative-SC scheme presented above. That is, how to obtain controllers connected only through a particular vendor switch. To facilitate this, the entity API allows for passing of filters that can be applied at intermediate nodes in the path. For our example, the filter would be *Switch.Vendor="W"*. Now, the host would call the HBA's *getController*() function with the filter *Switch.Vendor="W"*. When this call recursively reaches the switch, it would check if it satisfies the filter and only the switches that do, continue the recursion to their neighbors. Those that do not satisfy the filter return null.

The use of filters prevents unnecessary traversals on the paths that do not yield any results (e.g. paths to the controllers connected through switches from other vendors). The filters support many comparison operations like $=, \neq, >, \geq, <, \leq, \in$ and logical OR, AND and NOT on filters are also supported. The caching scheme incorporates filters as well (Section-4.4.2). The Alternative-SC presented above, can not use this filter based scheme since the possible number of filters can be enormous and thus always storing information in-memory for each such filter would be infeasible.

Notice that not all filters provide traversal optimizations. The filters that are at the "edge" of a path do not help. For example, a best practices policy - *if a Vendor-A host connected to a Vendor-W switch accesses storage from a Vendor-S controller, then the controller should have a firmware level* $> x$. In this case, the policy requires getting controllers with the filter *Switch.Vendor="W" AND Controller.Vendor="S"*. While the first term helps reduce the number of paths traversed, the second term does not – we still have to

check every controller connected through the appropriate switches. Therefore, we prefer not to apply the filters at the edge, instead obtaining all edge entities (controllers in this case) and then checking for all conditions ($Vendor$ and $FirmwareLevel$). This helps in bringing more useful data into the entity caches.

It is also important to mention that the traversal of the graph can also be done only for logical connections (due to zoning). This is facilitated by providing equivalent API functions for traversing links with end-points in particular zone, e.g. $getControllerLogical(Z)$ obtains all connected controllers in Zone Z, i.e. all controllers reachable through a path containing ports (HBA ports, switch ports, controller ports) in zone Z.

Given the above framework, we next discuss why the current policy evaluation approach is inefficient for impact analysis.

4.3.3 Impact Analysis: Inadequacies of Current Approach

During impact analysis, a SAN operation can trigger multiple policies to be evaluated. For example, a host being added into the SAN would require evaluation of intrinsic host policies (policies on basic attributes of the host - *all hosts should be from a single vendor*), various host interoperability policies with other connected devices, zoning policies, and so on. With the popular policy-based autonomic computing initiative, it is highly likely that the number of policies in a SAN would be very large. So it is imperative that only the relevant set of policies are evaluated. For example, for the host-addition case, a switch and controller interoperability policy should not be evaluated.

The current policy evaluation engines [3] use a coarse classification of **scopes**. In such a scheme, each policy is designated a *Scope* to denote the class of entities, it is relevant to. In addition, it is possible to sub-scope the policy as *intra-entity* - evaluation on attributes of a single entity class or *inter-entity* - evaluation on attributes of more than one entity class [3]. The motivation for such classification is to allow administrators, to do a policy check only for a select class of entities and policies in the SAN. Unfortunately, this form of classification is not efficient for impact-analysis due to the following reasons:

- **Lack of granularity**: Consider the policy which requires a Vendor-A host to be connected only to Vendor-S storage controller. Naively, such a policy would be classified into the Host scope and the Storage scope. Thus, whenever a new host is added to the SAN, it will be evaluated and similarly, when a controller is added. However, consider the addition of a new link between an edge and a core switch. Such a link could cause hosts to be connected to new storage controllers, and thus the policy would still need to be evaluated and so, the policy also needs to be added to the Switch scope. With only scope as the classification criteria, *any* switch related event would trigger this policy. It is possible to further *sub-scope* the policy to be an *inter-entity*. However, it still will be clubbed with other switch inter-entity policies, which will cause un-necessary evaluations.

- **Failure to identify relevant SAN region**: The current scoping mechanism fails to identify the region of the SAN that needs to be traversed for policy evaluation. Consider the two policies: (a) *All Vendor-A hosts should be connected to Vendor-S storage*, and (b) *All hosts should have atleast one and atmost four disjoint paths to controllers*. Both the policies would have identical scopes (host, controller and switch) and sub-scopes (inter-entity). Now, when a switch-controller link is added to the SAN, evaluation of (a) should traverse *only* the newly created paths – ensure that all new host-storage connections satisfy the policy; there is no need to traverse a path that has already been found to satisfy that policy. However, the same is not true for (b). Its evaluation would require traversing many old paths. The current scoping mechanism fails to identify policies of type (a) and would end up evaluating many old paths in order to provide a correct and general solution.

The current policy evaluation engines also **fail to exploit the locality of data** across various policies. For example, two distinct policies might require obtaining the storage controllers connected to the same host. In such a scenario, it is best to obtain the results for one and cache them to use it for the other. To the best of our knowledge, the current policy engines do not provide such caching schemes and rely on the underlying SMI-S data layer [3]

to do this caching (which could still require evaluating expensive *join* operations). This is, in part, due to the low number of policies in current SANs and the fact that currently, the policy checking process is primarily a non-real-time, scheduled task with periodic reporting (typically daily or weekly reporting periods). As we show in Section-4.5, a caching scheme has significant performance benefits and helps in interactive real-time analysis.

Such an efficiency is critical especially in the presence of **action policies**. Such policies when triggered initiate automatic operations on the SAN ("then" clause of the policy). These are typically designed as compensating actions for certain events and can do rezoning, introduce new workloads, change current workload characteristics, schedule workloads and more. For example, a policy for a write-only workload like *if Controller-A utilization increases beyond 95%, write the rest of the data on Controller-B*. When the policy is triggered, a new flow is created between the host writing the data and Controller-B, and policies related to that event need to be checked. The action might also do rezoning to put Controller-B ports in the same zone as the host and so, all zoning related policies would end up being evaluated. Overall, such a chain of events can lead to multiple executions of many policies. The caching scheme, combined with the policy classification, significantly helps in these scenarios.

4.4 Zodiac Impact Analysis: Optimizations

In this section, we present various optimizations in the Zodiac architecture that are critical for the scalability and efficiency of impact analysis. Zodiac uses optimizations along three dimensions.

1. *Relevant Evaluation*: Finding relevant policies and relevant regions of the SAN affected by the operation. This is accomplished using policy classification and is described in Section-4.4.1.

2. *Commonality of Data Accessed*: Exploiting data locality across policies or across evaluation for different entity instances. This is achieved by using caching, described in Section-4.4.2.

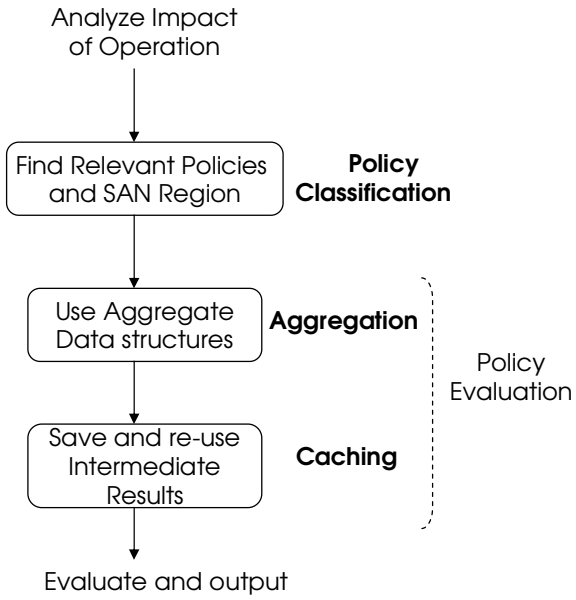


Figure 38: Zodiac Impact Analysis Process

3. *Aggregation*: Efficient evaluation of certain classes of policies by keeping certain aggregate data structures. This scheme is described in Section-4.4.3.

All three optimizations are independent of each other and can be used individually. However, as we show later in our results, the best performance is achieved by the combination of all three optimizations.

Figure-38 presents the flow of impact analysis process and how all optimizations fit in.

4.4.1 Policy Classification

The first policy evaluation optimization in Zodiac is policy classification. Policy classification helps in identifying the relevant regions of the SAN and the relevant policies, whenever an operation is performed. In order to identify the relevant SAN region affected by an operation, we classify the policies into four categories described below. Only the "if" condition of the policy is used for classification. Also, each policy class has a set of operations, which are the ones that can trigger the policy. This mapping of operations to policies can be made easily due to our classification scheme and is used to find the relevant set of policies.

1. **Entity-Class (EC) Policies**: These policies are defined only on the instances of a single entity class. For example, *all HBAs should be from the same vendor*, and

all Vendor-W switches must have a firmware level $> x$. Such policies do not require any graph traversals, rather a scan of the list of instances of the entity class. The relevant operations for this class of policies are addition/deletion of an entity-instance or modification of a "dependent" attribute of an instance like changing the firmware level of a switch (for our second example above). Additionally, EC policies can be subdivided into two types:

- *Individual (EC-Ind) Policy*: A policy that holds on every instance of the entity class. For example, *all switches must be from Vendor-W*. This class of policies has the characteristic that whenever an instance of the entity class is added/modified, the policy only needs to be evaluated on the new member.
- *Collection (EC-Col) Policy*: A policy that holds on a collection of instances of the entity class. For example, *the number of ports of type X in the fabric is less than N* and also *all HBAs should be from the same vendor*[1]. This class of policies might require checking all instances for final evaluation.

2. **Single-Path (SPTH) Policies**: These policies are defined on more than one entity on a single path of the SAN. For example, *all Vendor-A hosts must be connected to Vendor-S storage*. Importantly, SPTH policies have the characteristic that the policy is required to hold on *each* path. In our example, each and every path between hosts and storage must satisfy this policy. This characteristic implies that on application of **any** operation, there is no need to evaluate this policy on old paths. Only new paths need to be checked. The relevant operations for these policies are addition/deletion/modification of paths or modification of a "dependent" attribute (vendor name) of a "dependent" entity (storage controller) on the path.

3. **Multiple-Paths (MPTH) Policies**: These policies are defined across multiple paths of the SAN. For example, *all hosts should have atleast two and atmost four*

[1]This policy is also a collection policy since in order to evaluate the policy for the new instance, it is required to get information about existing instances.

disjoint paths to storage, and *a Vendor-A host should be connected to atmost five controllers*. MPTH policies cannot be decomposed to hold on individual paths for *every* operation. For the examples, adding a host requires checking only for the new paths created, whereas adding a switch-controller link requires checks on earlier paths as well. We are working on developing a notion distinguishing between the two cases[2]. In this chapter, we consider MPTH policy as affecting all paths. The relevant operations for these policies are addition/deletion/modification of paths or modification of a "dependent" attribute of a "dependent" entity on the path.

4. **Zoning/LUN-Masking (ZL) Policies**: These policies are defined on zones or LUN-Mask sets of the SAN. For example, *a zone should have atmost N ports*, and *a zone should not have both windows or linux hosts*. For our discussion, we only use zone policies, though the same approach can be used for LUN-Masking policies. Notice that these policies are similar to EC policies with entity-class being analogously replaced by zones or LUN-Mask sets. Just as EC policies are defined on attributes of entity instances, ZL policies are defined on attributes of zone instances. Also similar to EC policies, Zone policies can be collection policies, requiring evaluation over multiple zones, (e.g. *the number of zones in the fabric should be atmost N*)[3] and individual policies, requiring evaluation only over an added/modified zone (e.g. *all hosts in the zone must be from the same vendor*). Also, within a zone, a policy might require evaluation over only the added/modified component (*Zone-Member-Ind*) or all components (*Zone-Member-Col*). An example of a *Zone-Member-Ind* policy is *all hosts in the zone should be windows*, and an example of *Zone-Member-Col* policy is *a zone should have atmost N ports*. The relevant operations for this class of policies are addition/deletion of a zone instance or modification of an instance (addition/deletion of ports in the zone).

[2]Informally, typically an operation affecting only the "principal" entity of the policy (host in the examples) does not require checking old paths.

[3]Such a policy is required since the switches have a limit on the number of zones they can handle

Note that the aim of this classification is not to semantically classify all conceivable policies, but rather to identify the policies that can be optimized for evaluation. Having said that, using our classification scheme, it was indeed possible to classify all policies mentioned in [3], the only public set of SAN policies collected from administrators and domain experts. The basic difference between the classification scheme in [3] and our scheme stems from the fact that it classifies policies based on specification criteria, while we use the internal execution criteria for the classification. This helps us in generating optimized evaluation code by checking only the relevant regions of the SAN.

As evident from our experiments (described in Section-4.5), the classification technique helps in significantly reducing the total cost of policy impact analysis.

4.4.2 Caching

The second optimization we propose, uses a caching scheme to cache relevant data at *all* nodes of the SAN resource graph. Such a scheme is extremely useful in an impact-analysis framework due to the commonality of data accessed in the following scenarios:

- *Multiple executions of a single policy*: A single policy might be executed multiple times on the same entity instance due to the chaining of actions, defined in the *then* clause of the triggered policies. Any previous evaluation data can be easily reused.

- *Execution of a single policy for different instances of entities*: For example, consider an operation of adding a policy like *all Vendor-A hosts should be connected to Vendor-S storage*. For impact analysis, the policy needs to be evaluated for all hosts. In our immediate-neighbor scheme, for the evaluation of this policy, a host, say Host-H, would call its HBA's $getController()$ function, which in turn would call its ports' $getController()$ function, which would call the edge switch (say Switch-L) and so on. Now, when any other host connected to Switch-L calls its $getController()$ function, it can reuse the data obtained during the previous evaluation for Host-H. Note that with no replacement, the caching implies that traversal of any edge during a policy evaluation for all entity instances is done *atmost* once. This is due to the fact that

after traversing an edge {u,v} once, the required data from v would be available in the cache at u, thus preventing its repeated traversal.

- *Locality of data required across multiple policies*: It is also possible, and often the case, that multiple policies require accessing different attributes of the same entity. As mentioned earlier, we do not apply filters to the "edge" entities (e.g. controllers for a *getController*() call) and retrieve the full list of entities. Now, this cached entry can be used by multiple policies, even when their "dependent" attributes are different.

As mentioned earlier, the caching scheme incorporates filters as well. Whenever an API function is called with a filter, the entity saves the filter along with the results of the function call and a **cache hit** at an entity occurs only when there is a complete match, i.e. the cached entry has the same API function call as the new request and the associated filters are also the same. This condition can be relaxed by allowing a partial match, in which the cached entry is for the same function call, but can have a more general filter. For example, assume a cache entry for *getController*() with the filter *Switch.Vendor="W"*. Now, if the new request requires controllers with the filter *Switch.Vendor="W" AND Switch.FirmwareLevel* $> x$, the result can be computed from the cached data itself. We leave this for future work. Also, the current caching scheme uses LRU for replacement.

4.4.3 Aggregation

It is also possible to improve the efficiency of policy execution by keeping certain aggregate data structures. For example, consider a policy which mandates that *the number of ports in a zone must be atleast M and atmost N*. With every addition/deletion of a port in the zone, this policy needs to be evaluated. However, each evaluation would require counting the number of ports in the zone. Imagine keeping an aggregate data structure that keeps the number of ports in every zone. Now, whenever a port is added/deleted, the policy evaluation reduces to a single check of the current count value.

We have identified the following three classes of policies that have simple aggregate data structures:

- *Unique*: This class of policies require a certain attribute of entities to be unique. For example, policies like *the WWNs of all devices should be unique*, *all Fibre Channel switches must have unique domain IDs*. For these class of policies, a hashtable is generated on the attribute and whenever an operation triggers this policy, the policy is evaluated by looking up that hashtable. This aggregate data structure can provide good performance improvements especially in big SANs (Section-4.5). Note that such an aggregate is kept only for EC and ZL policies (where it is easier to identify addition/deletion). However, there does not appear to be any realistic SPTH or MPTH unique policies.

- *Counts*: These policies require counting a certain attribute of an entity. Keeping the count of the attribute prevents repeated counting whenever the policy is required to be evaluated. Instead, the count aggregate is incremented/decremented when the entity is added/deleted. A count aggregate is used only for EC and ZL policies. While SPTH and MPTH count policies do exist (e.g. *there must be atmost N hops between host and storage* and *there must be atleast one and atmost four disjoint paths between host and storage* respectively), maintaining the counts is tricky and we do not use an aggregate.

- *Transformable*: It is easy to see that the policy evaluation complexity is roughly of the order $EC\text{-}Ind \approx Zone\text{-}Member\text{-}Ind < EC\text{-}Col \approx Zone\text{-}Member\text{-}Col < SPTH < MPTH$. It is actually possible to transform many policies into a lower complexity policy by keeping additional information about some of the dependent entities. For example, consider a policy like *all storage should be from the same vendor*. This policy is an EC-Col for entity class - Storage. However, keeping information about the current type of storage (T) in the system, the policy can be reduced to an equivalent EC-Ind policy – *all storage should be of type* T. Similarly, a Zone-Member-Col policy like *a zone should not be both windows and linux hosts* can be transformed into multiple Zone-Member-Ind policies *there should be only type* T_i *hosts in zone* Z_i, where T_i is

the current type of hosts in the Z_i. For these transformed policies, a pointer to the entity that provides the value to aggregate is also stored. This is required, since when the entity is deleted, the aggregate structure can be invalidated (can be re-populated using another entity, if existing).

For all other policies, we currently do not use any aggregate data structures.

4.5 Experimental Setup and Results

In this section, we evaluate our proposed optimizations as compared to the base policy evaluation provided by current engines. We start by describing our experimental setup beginning with the policy set.

4.5.1 Microbenchmarks

With the policy based management being in a nascent state so far, there does not exist any public set of policies that is used in a **real** SAN environment. The list of policies contained in [3] is indicative of the *type* of possible policies and not an accurate "trace" for an actual SAN policy set. As a result, it is tough to analyze the overall and cumulative benefits of the optimizations for a real SAN. To overcome this, we try to demonstrate the benefits of optimizations for different categories of policies individually. As mentioned earlier, since we have been able to classify all policies in [3] according to our scheme, the benefits would be additive and overall useful for a real SAN as well. In addition, this provides a good way of comparing the optimization techniques for each policy class.

We selected a set of 7 policies (Figure-39) as our working sample. Four of them are EC policies, two are path policies (SPTH and MPTH) and one is a zone policy. All 7 policies are classified according to the classification mechanisms presented in Section-4.4.1. Any aggregates that are possible for policies are also shown. For this set of policies, we will evaluate the effectiveness of our optimizations individually.

4.5.2 Storage Area Network

An important design goal for Zodiac was scalability and ability to perform impact analysis efficiently even on large SANs of 1000 hosts and 200 controllers. Since it was not possible to

#	Policy	Classification
1	Every HBA that has a vendor name V and model M should have a firmware level either n1, n2 or n3	EC
2	No two devices in the system can have the same WWN (World Wide Name)	EC Unique
3	The number of ports of type X in the fabric is less than N	EC Counts
4	The SAN should not have mixed storage type such as SSA, FC and SCSI parallel	EC-Col to EC-Ind (Transform)
5	An ESS array is not available to open systems if an iSeries system is configured to array	SPTH
6	A HBA cannot be used to access both tape and disk drives	MPTH to SPTH (Transform)
7	No two different host types should exist in the same zone	Zone-Member-Col to Zone-Member-Ind

Figure 39: Policy Set

construct such large SANs in the lab, for our experiments, we emulated four different sized SANs. Please note that in practice, Zodiac can work with any real SAN using an SMI-S compliant data store.

In our experimental SANs, we used hosts with two HBAs each and each HBA having two ports. Storage controllers had four ports each. The fabric was a core-edge design with 16-port edge and 128-port core switches. Each switch left certain ports unallocated, to simulate SANs constructed with future growth in mind. The four different configurations correspond to different number of hosts and controllers:

- **1000-200**: First configuration is an example of a big SAN, found in many data centers. It consists of 1000 hosts and 200 controllers with each host accessing all controllers (full connectivity). There were 100 zones.
- **750-150**: This configuration uses 750 hosts and 150 controllers with full connectivity. There were 75 zones.
- **500-100**: This configuration has 500 hosts, 100 controllers and 50 zones.
- **250-50**: This configuration is a relatively smaller SAN with 250 hosts, 50 controllers and 25 zones.

4.5.3 Implementation Techniques

For our experiments, we evaluate each of the policies in the policy-set with the following techniques:

- **base**: This technique is the naive implementation, in which there is no identification of the relevant region of the SAN. Only information available is the vanilla scope of the policy. Due to lack of classification logic, this implementation implies that the evaluation engine uses the same logic of code generation for all policies (check for all paths and all entities). Also, there is no intermediate caching and no aggregate data structures are used.

- **class**: This implementation uses the classification mechanism on top of the base framework. Thus, it is possible to optimize policies by only evaluating over a relevant SAN region, but no caching or aggregation is used.

- **cach**: This implementation technique caching on top of the *base* framework. No classification or aggregation is used.

- **agg**: This implementation technique only uses aggregate data structures for the policies (where ever possible). There is no caching or classification.

- **all**: This implementation uses a combination of all three optimization techniques.

Using these five classes of implementation, we intend to show (a) inadequacy of the base policy, (b) advantages of each optimization technique and (c) the performance of the **all** implementation. Zodiac is currently running on a P4 1.8 GHz machine with 512 MB RAM.

4.5.4 Policy Evaluation

In this section, we present our results of evaluating each policy 100 times to simulate scenarios of chaining and execution for multiple instances (e.g. adding 10 hosts). The policies are evaluated for all four SAN configurations (X-axis). The Y-axis plots the time taken to evaluate the policies in milliseconds. The results have been averaged over 10 runs.

4.5.4.1 Policy-1

"Every HBA that has a vendor name V and model M should have a firmware level either n1, n2 or n3"

The first policy requires a certain condition to hold on an HBA entity class. We analyze the impact of the policy when an HBA is added. The *base* implementation will trigger the HBA scope and try to evaluate this policy. Due to its lack of classification logic, it will end up evaluating the policy afresh and thus, for all HBA instances. The *class* implementation would identify it to be an EC-Ind policy and only evaluate on the new HBA entity. The *cach* implementation does not help since there is no traversal of the graph. The *agg* implementation also does not help. As a result, *all* implementation is equivalent to having only *class* optimization. Figure-40 shows the results for the different SAN configurations.

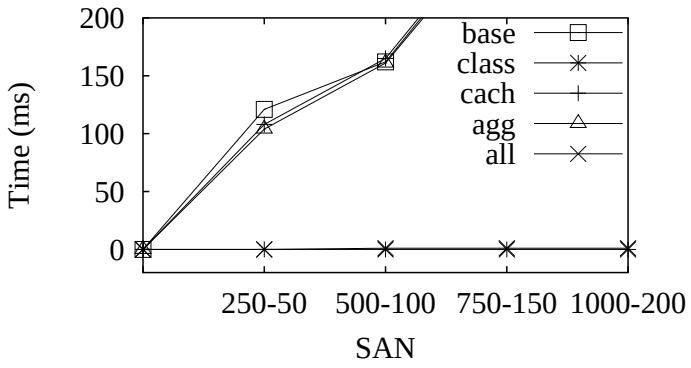


Figure 40: Policy-1. *class*, *all* provide maximum benefit

As seen from the graph, there is a significant difference between the best optimized evaluation (*all*) and the *base* evaluation. Also, as the size of the SAN increases, the costs for the *base* implementation increase, while the *all* implementation stays the same, since irrespective of SAN size, it only needs to evaluate the policy for the newly added HBA.

4.5.4.2 Policy-2

"No two devices in the system can have the same WWN."

The second policy ensures uniqueness of world wide names (WWNs). We analyze the impact when a new host is added. The *base* implementation will trigger the *device* scope

and evaluate the policy. Again, without classification logic, it will check that all devices have unique WWNs. The *class* implementation will only check that the new host has a WWN different from other devices. The *cach* implementation does not provide any benefit and performs similar to *base*. The *agg* implementation will create a hashtable, and do hashtbale lookups. The *all* implementation also uses the hashtable and checks only the new host for uniqueness.

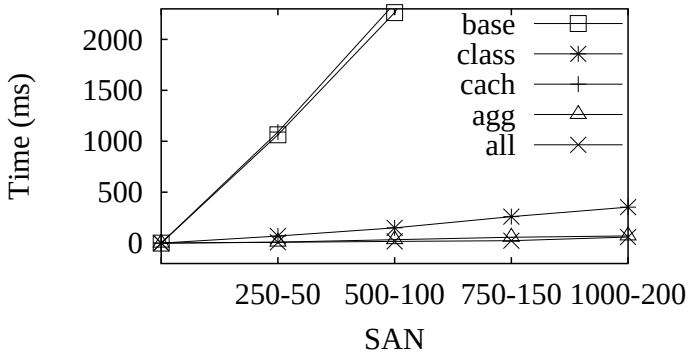


Figure 41: Policy-2. *agg*, *all* provide maximum benefit

As Figure-41 shows, *agg* and *all* perform much better than the *base* implementation. *class* performs better than *base* by recognizing that only the new host needs to be checked.

4.5.4.3 Policy-3

"The number of ports of type X in the fabric is less than N."

The third policy limits the total number of ports in the fabric. We analyze the impact of adding a new host with 4 ports to the SAN. For each added port, the *base* implementation will count the total number of ports in the fabric. The *class* implementation performs no better, since it is an EC-Col policy. The *cach* implementation also does not help. The *agg* implementation keeps a count of the number of ports and only increments the count and checks against the upper limit. The *all* implementation also exploits the aggregate counts.

As can be seen from Figure-42, *agg* and *all* perform significantly better due to the ability of aggregating the required information for the policy evaluation.

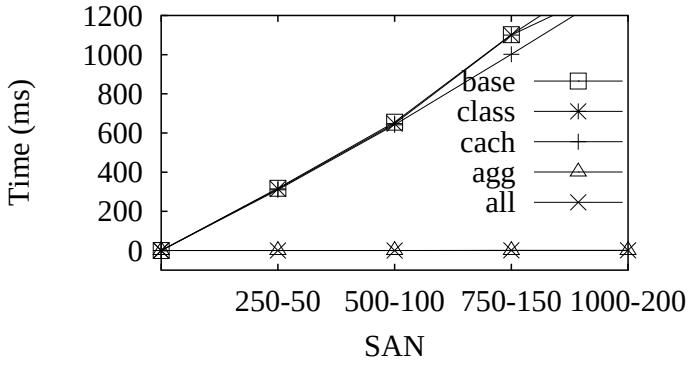


Figure 42: Policy-3. *agg*, *all* provide maximum benefit

4.5.4.4 Policy-4

"The SAN should not have mixed storage type such as SSA, FC and SCSI parallel"

The fourth policy ensures that the SAN has uniform storage type. For this policy, we analyze the impact of adding a new storage controller. The *base* implementation will trigger the storage scope and evaluate the policy ensuring all controllers are the same type. The *cach* implementation will not help. The *class* implementation only checks that the newly added controller is the same type as every other controller. The *agg* implementation will transform the policy to an EC-Ind policy by keeping an aggregate value of the current controller type, T in the SAN. However, without classification, it would end up checking that all controllers have the type T. The *all* implementation combines the classification logic and the aggregate transformation to only check for the new controller.

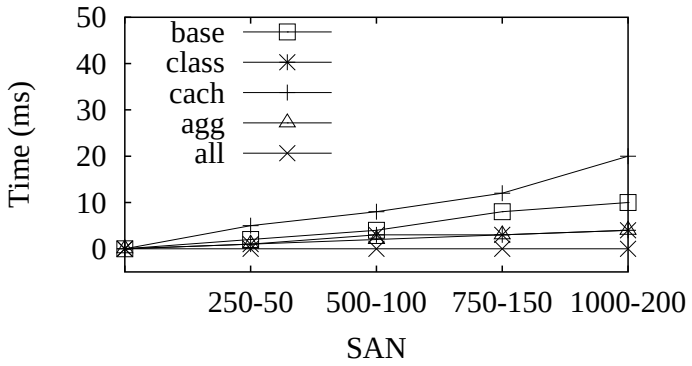


Figure 43: Policy-4. *all* provides maximum benefit

Figure-43 shows the result with *all* performing the best, while *class* and *all* doing better

than *base* and *cach*. The difference between the best and poor implementations is small since the total number of controllers is small.

4.5.4.5 Policy-5

"An ESS array is not available to open systems if an iSeries system is configured to array." The fifth policy is an SPTH policy that checks that an iSeries open systems host does not work if an ESS array is used with the controller. We analyze the impact of adding a new host to the SAN for this policy. The *base* implementation ensures that all iSeries open systems hosts do not have any ESS controllers connected to them. This requires calling the *getController*() API functions of the host entities and will cause traversals of the graph for all host-storage connections. The *class* implementation identifies that it being an SPTH, only the new created paths (paths between the newly added host and the connected storage controllers) need to be checked. The *cach* implementation will run similar to *base*, but will cache all function call results at intermediate nodes (As mentioned before, it would mean that each edge will be traversed atmost once). The *agg* implementation does not help and the *all* implementation would use both the classification logic and the caching.

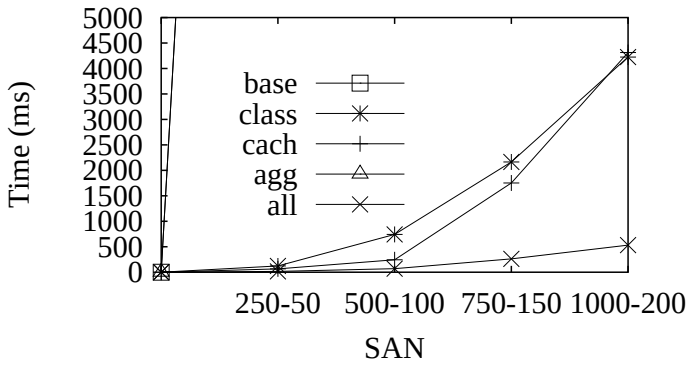


Figure 44: Policy-5. Only *all* provides maximum benefit

As shown in Figure-44, the *base* and *agg* implementation perform extremely poorly (multiple orders of magnitude in difference) due to multiple traversals for the huge SAN graph. On the other hand, *class* and *cach* are able to optimize significantly and their combination in the *all* implementation provides drastic overall benefits. It also scales extremely well with the increasing SAN size.

4.5.4.6 Policy-6

"A HBA cannot be used to access both tape and disk drives."

The sixth policy is an MPTH policy which requires checking that each HBA is either connected to tape or disk storage. We analyze the impact of adding a host with 2 HBAs to the SAN. The *base* implementation would check the policy for all existing HBAs. The *cach* implementation would do the same, except the caching of results at intermediate nodes. The *class* implementation does not optimize in this case since it considers it an MPTH policy and checks for all paths (Section-4.4.1). The *agg* implementation transforms the policy to an SPTH by keeping aggregate for the type of storage being accessed, but checks for all HBAs due to the lack of classification logic. The *all* implementation is able to transform the policy and then use the SPTH classification logic to only check for the newly added HBAs.

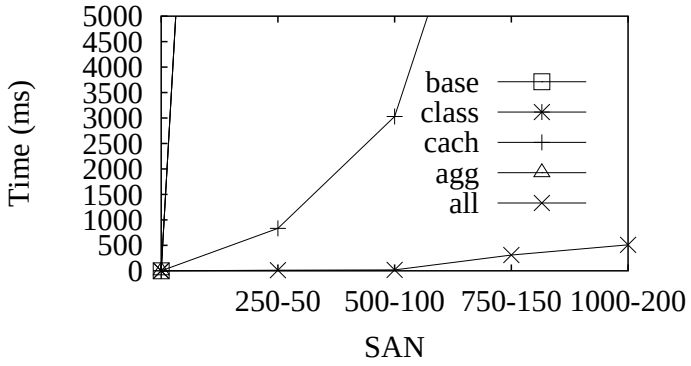


Figure 45: Policy-6. Only *all* provides maximum benefit

Figure-45 shows the results. The *base*, *class* and *agg* implementation perform much poorly then the *cach* implementation, since the *cach* implementation reuses data collected once for the other. The *all* implementation performs the best by combining all optimizations.

4.5.4.7 Policy-7

"No two different host types should exist in the same zone."

The eighth policy requires that all host types should be the same in zones. We analyze the impact of adding a host HBA port to a zone. The *base* implementation would check

that all hosts in each zone are of the same type. The *class* implementation would check only for the affected zone. The *cach* implementation would be the same as *base*. The *agg* implementation would keep an aggregate host type for each zone and check the policy for all zones. The *all* implementation would combine the aggregate with the classification and only check for the affected zone, that the new host has the same type as the aggregate host type value. Figure-46 shows the results. Again *all* implementation performs the best, though the difference between all implementations is small.

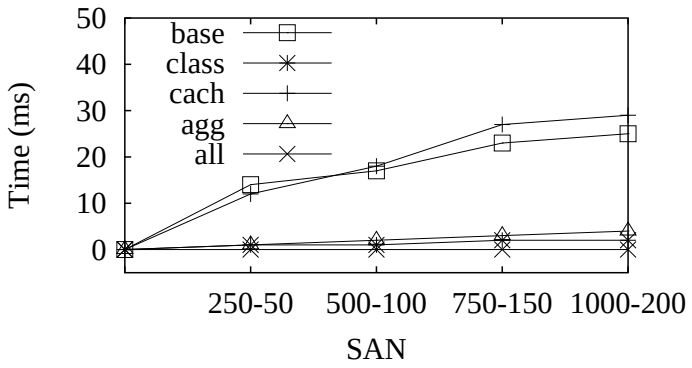


Figure 46: Policy-7. *all* provides maximum benefit

4.5.5 Summary of Results

Through these micro-benchmarks, we have analyzed the performance of various optimization implementations for a variety of real SAN policies. Based on the results, we find that each optimization – classification, caching and aggregation – has a niche of evaluation scenarios where it is most effective. For example, caching helps the most during the evaluation of path policies. Overall, a combination of the optimization techniques – *all* implementation yields maximum benefits.

Our evaluation shows that use of optimizations allows our approach to scale to large sizes of the SANs with thousand servers and hundreds of storage controllers. Additionally, it maintains good performance and is able to obtain results quickly. Thus, it will serve as an important tool for administrators to analyze impact of changes efficiently and quickly.

4.6 Related Work

With the growth in amount of storage resources, there has been a strong initiative for automating various management tasks and making systems self-sufficient [5, 9, 63, 36]. Most of this research has focused on various planning tasks - capacity planning, including Minerva [5], Hippodrome [9], Ergastulum [10]; fabric planning like Appia [190, 191], and disaster recovery planning [96, 97].

Impact analysis, also referred to as "what-if" or change-management analysis, is another closely related management task. Some of the planning work described above can actually be used for such analysis. For example, Ergastulum [10] can be used to analyze storage subsystems and Keeton et al's [97] helps in analyzing disaster recovery scenarios. Another recent work by Thereska et al. [174] provides what-if analysis using the Self-* framework [63]. There also exist tools and simulators like [91, 194] that provide impact analysis for storage controllers. Most of the what-if approaches utilize device and behavioral models for resources. Significant amount of research has been done both in developing such models [178, 7, 153, 189, 194] and using those models [52, 44, 121, 17].

Zodiac is different from existing impact analysis work, due to its close integration with policy based management. Using Zodiac, an administrator can analyze the impact of operations not only on system resources but also on system policies. In addition, the analysis accounts for all subsequent actions triggered by policy executions. As we describe later, efficient analysis of policies is non-trivial and critical for overall performance.

The Onaro SANscreen product [157] provides a similar predictive change management functionality. However, from the scarce amount of published information, we believe that they only analyze the impact for a small set of policies (mainly security) and do not consider any triggered policy actions. We believe this to be an important shortcoming, since typically administrators would specify policy actions in order to correct erroneous events and would be most interested in analyzing the impact of those triggered actions. The EMC SAN Advisor [155] tool provides support for policy evaluations, but is not an impact analysis tool. Secondly, it pre-packages its policies and does not allow specification of custom policies.

In the policies domain, there has been work in the areas of policy specification [49,

14], conflict detection [61] and resource management [111]. The SNIA-SMI [168] is also developing a policy specification model for SANs. To the best of our knowledge, there does not exist any SAN impact analysis framework for policies. [3] proposed a policy based validation framework, which is typically used as a periodic configuration checker and is not suitable for interactive impact analysis.

4.7 Summary

In this chapter, we presented Zodiac - an efficient impact analysis framework for storage area networks. Zodiac enables system administrators to do proactive change analysis, by evaluating the impact of their proposed change before actually applying it to the SAN. It includes the impact on SAN resources, existing policies and also, due to the actions triggered by any of the violated policies. For greater efficiency, we proposed three optimizations - classification, caching and aggregation, for impact analysis. A combination of these optimizations makes impact analysis fast and scalable, meeting the demands of dynamic and large SSP SANs.

The objective of the Zodiac framework is only to perform impact analysis. For storage service provider SANs, this would allow administrators to immediately analyze the effects of a client-enterprise's workload change and plan to integrate those changes adequately without adverse impact on other workloads. However, Zodiac does not address this problem of *planning* to integrate the proposed change. This could involve resource allocation or deallocation, for example, provisioning of new servers or storage and migration of applications or storage. In the next chapter, we describe the SPARK framework which addresses this problem using modern server virtualization technologies.

CHAPTER V

EFFICIENT RESOURCE ALLOCATION IN VIRTUALIZED SANS

The primary advantage of the storage-as-a-service model for an enterprise is its ability to reduce storage management costs. By offloading the storage hardware and management tasks to a storage service provider (SSP), enterprises are freed of any involved complexities. On the other hand, SSPs are able to use economies of scale to amortize the costs across their various clients. However, they have to be very efficient in allocation of resources in their data centers and can not rely on the usual technique of gross over-provisioning.

Resource allocation becomes a complex problem especially when dealing with a dynamic environment, such as that of a SSP. Dealing with workload surges, node failures, downtimes, growth and other typical data center pain points require manual planning and active administrator involvement. This is prone to errors and can result in suboptimal placement decisions when selecting nodes where affected applications can be placed. In addition, it is a bottleneck in dealing with scenarios, where a quick response is required.

Doing fast, high quality placement planning autonomically in a modern data center is a challenging problem. Unlike previous research, it requires handling storage and computational resources in a coupled manner. For example, moving an application to another physical node in response to a workload surge needs to take into account the "affinity" of the new node to application's storage. Similar constraints exist when migrating application storage. Complexity is introduced into this problem due to the heterogeneity in data centers. The hardware and Storage Area Network (SAN) fabric in data centers are incrementally built over time and disparate resources co-exist in the same environment. This results in non-uniform affinities between computational and storage nodes.

Further contributing to this non-uniformity are the recent trends in SAN hardware. Many vendors have introduced specialized SAN devices that include processing power at non-traditional locations in the SAN. For example, Cisco MDS 9000 switches [41] and IBM

DS8000 storage controllers [86] have available processing power that has been successfully used for application deployment [40, 85, 87]. Using virtualization technologies, proximity of these processing nodes to storage can be effectively exploited by careful placement of applications (for example, an I/O intensive application at the controller processor can access its storage faster).

In this chapter, we describe a framework to handle this challenging problem. We extensively use server virtualization technologies to create an adaptive SAN data center environment. Advances in virtual machine (VM) technologies with the emergence of VMware [184] and Xen [15] are playing an important role in shaping the modern data center. By isolating applications into independent virtual containers that can run concurrently on a single physical machine, virtualization reduces hardware, space and maintenance costs and improves resource utilization. Other recent innovations allow live migration of application VMs from one physical machine to another without much downtime [129, 42]. By combining these advancements with similar data migration technologies [110, 8, 75, 100], we aim to create a highly adaptive data center environment, suitable for efficiently delivering storage as a service to enterprises.

Our framework, called Stable-Proposals-And-Resource-Knapsacks or simply SPARK aims to find an optimal placement of applications' storage and CPU on the resource nodes in the SAN. Its primary uniqueness lies in accounting for affinities (or lack thereof) between storage and CPU nodes during placement decisions. In its design, SPARK is based on a novel combination of two well-studied problems – the Stable Marriage problem and the 0/1 Knapsack problem. It yields high quality placement decisions, yet is intuitive and easy to understand. In our synthetic experiments, SPARK is **within 4% of the optimal values** (lower bounds) computed using LP formulation of the coupled placement problem for a wide variety of workloads and SAN environments[1]. It also consistently outperforms natural candidate algorithms by 30-40%. It is more **scalable** and **fast** – being an order of magnitude faster than the next best quality candidate algorithm (PAIR-GR, ref. Section-5.3) for large

[1] Due to limits imposed on LP solvers [60] by large number of variable and constraints, LP solutions could be obtained, and compared against, for small and medium problem sizes only.

instances of the problem.

Its other most salient feature is its **versatility**. SPARK can naturally handle many real-life data center constraints on SAN nodes that are suitable candidates for placing an application, based on system policies and application preferences. It can also **iteratively** improve any existing configuration by appropriately accounting for VM and data migration costs (ref. Section-5.3.5). This built-in versatility makes SPARK a superior solution to handle the coupled placement challenge in SSP environments.

The rest of the chapter is organized as follows. We describe relevant technologies and related work in Section-5.1. In Section-5.2, we describe the model SAN environment considered in this work and the architecture of our solution framework. We describe the design of the SPARK algorithm in Section-5.3. Detailed experimental evaluation of our algorithm and its comparison with other techniques is presented in Section-5.4. We finally summarize our contributions in Section-5.5.

5.1 Background and Related Work

In this section, we describe the necessary background and related research to our work. We start with the recent advances in virtualization technologies and current hardware trends that are shaping the modern SAN data center environment. Then, we describe the state of the art for planning in virtualized data centers and articulate the uniqueness of our work.

5.1.1 Advances in Virtualization Technologies

The concept of Virtual Machine technology, first introduced in the 1960s [68], has been widely exploited in recent years for consolidating an ever-expanding and unmanageable hardware infrastructure in data centers [152]. New virtualization technologies like VMware [184] and Xen [15] allow applications to run in isolated containers without interfering with each other. Applications which run on single machines can now be moved to virtual machines; these virtual machines in turn can be moved to fewer physical machines increasing resource utilization and reducing space and hardware management costs. Table-20 lists the virtualization technologies and the ongoing work in the area.

One most notable recent development is *live migration* technologies like VMotion [129]

Table 20: Virtualization Advancements

Technology	**Related Work**
Architecture – Paravirtualization – Hosted VMs	 Xen [15], Denali [192] VMware [184, 170]
Live Migration	VMotion [129], Xen Migration [42]
VM Functioning and Optimizations	Memory Management [187], VMM-bypass I/O [109], Optimizing Network Virtualization in Xen [116]
Software Deployment using VMs (*Virtual Appliances*)	Collective [33], Internet Suspend / Resume [24]
Planning for VM environments	VMware DRS (Distributed Resource Scheduler) [181]

and [42] for Virtual Machines. This enables applications to be moved from one machine to the other in real-time with minimal service downtime. This can be an extremely useful tool for administrators of data centers as it facilitates load-balancing, fault-tolerance and system maintenance. The live migration technology is a critical enabler for the SPARK framework discussed in this work.

5.1.2 Current Hardware Trends

Recent developments in hardware technology aim to make devices that can better use virtualization technologies and vendors have begun rolling out innovative products. IBM has implemented a pool of virtualization technologies on the POWER5 servers [1] using the *Logical Partitioning* (LPAR) technology in which multiple resources can be placed under the exclusive control of a given logical partition. The IBM DS8000 storage controllers support POWER5 logical partitioning which makes them suitable for hosting applications [87]. IBM recently demonstrated running a DB2 application in an LPAR, using which most of the computation associated with ad-hoc queries in OLAP workloads can be offloaded to the storage controller. [82] also describes a system that speeds up search of unindexed data by running "searchlets" at the storage system.

Another similar application is the offloading of data access functions to switches. Cisco MDS 9000 switches [41] have multiple Intel x86 processor blades and can be fitted with

multiple application modules. Cisco has already displayed the feasibility of this idea by implementing *IBM TotalStorage SAN Volume Controller* [84] and *Veritas Storage Foundation for Networks Solution* [179] on the switches.

Such hardware innovations are increasing the level of heterogeneity in modern SAN environments by making available specialized processing power at non-traditional locations. If used appropriately, this can have a significant impact on overall application performance. These trends point towards the need of an intelligent application placement technique. The SPARK algorithm described in this chapter exploits this specialization by doing coupled placement of storage and CPU resources, accounting for application affinities for such specialized nodes.

5.1.3 Planning in Virtualized Data Centers: State of the Art

Current work in the area of planning deals either with storage resource optimization or server performance optimization. Recent products like VMware Infrastructure-3 [182] and its Dynamic Resource Scheduler (DRS) [181] continuously monitor CPU resource utilization in a virtualized data center and based on certain policies can prioritize how CPU resources are allocated to virtual machines. While it is a good starting point, it only concentrates on optimization of CPU resources for virtual machines staying oblivious of the underlying storage system. Oceano [11] and Muse [35] are also resource provisioning tools in data center environments focusing on allocating CPU resources. However, they also do not account for the underlying storage system and its affinity to a CPU node.

Similar tools at the storage end of the spectrum like Ergastulum [10], Hippodrome [9] and Minerva [5], attempt to find optimized storage system configurations taking into account only the storage requirements of applications. In contrast, SPARK attempts to optimize overall application performance taking storage as well as CPU requirements into account. Our experiments reveal how SPARK's coupled placement is better than independent storage and CPU placement.

Other related planning work includes algorithms for migrating data objects from one configuration to another [110, 8, 75, 100]. Assuming that the final configuration is known,

these systems attempt to reconfigure the storage system using minimum number of data migration steps. Our work is complementary to theirs and seeks to *identify* appropriate storage and CPU placement locations and can leverage these efforts while executing the final plan.

5.2 SPARK Architecture

In this section, we describe the architecture of our framework and its various components. We start with a description of a model SAN data center environment.

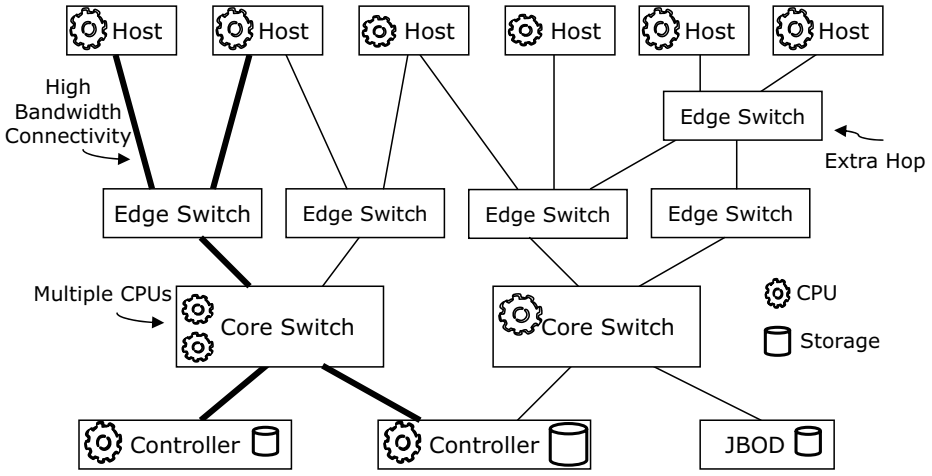


Figure 47: Modern SANs with heterogeneous resources

A storage area network in a data center consists of multiple devices – application servers, edge and core switches, different types of storage devices from high end controllers to Just-a-Bunch-Of-Disks (JBOD). These devices are connected through a high speed network, usually referred to as the *fabric*. Figure-47 shows an example core-edge design SAN topology. As mentioned earlier, modern SANs are heterogeneous in nature with resources differing in bandwidth connectivity, resources capacities and capabilities. Additionally, many non-traditional devices now include significant processing power (Figure-47 shows core switches and storage controllers with CPU nodes). Note that these CPUs are as powerful as ones available in traditional application servers, if not more.

An important characteristic of this modern SAN environment is its support for virtualization platform deployment. Most CPU architectures today have been virtualized

including the ones available on storage controllers and switches. For example, Cisco MDS 9000 switches ship with Intel x86 processors, virtualized by VMware, Xen and IBM DS 8000 controllers have PowerPC processors, virtualized through the Logical Partitioning (LPAR) technology [1]. With increasing push towards standardization of VM formats [183], these processors are equal candidates for addition to virtualized resource pools that would normally contain only application servers. In fact, these nodes can be even more effectively utilized when hooks exist for specialized applications that can communicate with fast-path APIs available at these nodes. For example, an application running at the controller storing its database, can significantly improve table scan performance by directly communicating with the controller storage [86].

Such a virtualized environment can be used to realize *utility computing*, where all resources are aggregated into pools and workloads are dynamically mapped to use resources from these common pools. We discuss this goal in the next section.

5.2.1 Destination: Utility Computing

Consider the SAN environment in Figure-47 where a virtualization platform (say, VMware Server [185]) has been installed at all CPU nodes and all applications have been encapsulated into VMware virtual machines. Administrators can choose to run multiple application VMs at each CPU node for efficient resource utilization. Now, suppose an application gets hit with a sudden workload surge and its VM begins using extra CPU resources. This might impact performance of all applications running on that node.

To correct this, the administrator simply moves one of the application VMs on this physical node (*source*) to another VMware server on a *target* node using the VMware migration technology – VMotion [129]. The VM state is encapsulated into regular files and stored in the SAN. The target server accesses these files concurrently and the active memory and execution state of the VM is transmitted over a high speed network. Since the network is also virtualized, the virtual machine retains its network identity and connections, ensuring a seamless migration process. Please note that use of VMware technologies is for illustrative purposes only and similar migration technology also exist for Xen [42], which can migrate

VMs running *interactive* applications in only tens of milliseconds.

While this technology seemingly brings the utility computing dream within reach, the human involvement in planning for selecting an appropriate VM to move and the appropriate target node for migration can lead to errors, sub-optimal decision making and is also a bottleneck in situations that require immediate response. As mentioned earlier, autonomically performing such coupled placement decisions accounting for storage-cpu affinities, is complex and SPARK aims at addressing this challenge.

5.2.2 Planner Architecture

In this section, we describe the architecture of our framework detailing its internal components and its interaction with external entities. Figure-48 shows the framework.

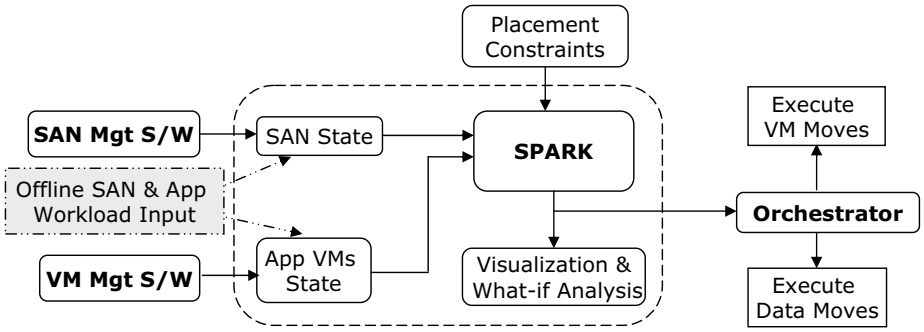


Figure 48: SPARK Planner Architecture

External Components: Planning requires detailed information of the underlying storage SAN, which is obtained from *SAN management tools* like EMC Control Center [54] and IBM Total Storage Productivity Center [88]. These tools keep real-time information for all devices and events in the SAN. Additionally, information about the state of application VMs is required to obtain utilizations and availabilities at processing nodes. This is obtained from *VM management suites* like VMware Infrastructure-3 [182] and similar products [198, 4, 13, 146]. Relevant SAN and VM state information can also be input in an offline mode, especially when doing what-if analysis (see below). *Constraints and preferences* for placing application VMs and storage on certain nodes are also provided as input to the planning framework. Furthermore, the planning engine is connected to an *Orchestrator* that can execute VM and data migration workflows using storage management and VM

management tools.

Internal Components: Internally, the framework consists of a *SAN State* component that maintains relevant SAN state required during planning, for example, SAN topology information. Similarly relevant *VM State* information is maintained. The SPARK algorithm takes this information along with placement constraints to generate a placement plan of application storage and CPU on resource nodes. This plan can be viewed through a *visualization engine* and can also be used as a *what-if analyzer*, through which an administrator can proactively assess the behavior of the system in response to certain dynamic scenarios (for example, what-if App-A's workload surges by 20%?). Finally, the placement plan is executed through the orchestrator (if desired, the plan can be verified by an administrator).

Modes of Operation: This framework can be deployed both in an *offline* and an *online* setup. In an offline mode, an administrator can input relevant information (maybe imported from a snapshot of a live system state) and visualize the plans generated by the framework and perform what-if analysis for his/her current setup. In an online mode, the framework can continuously monitor the SAN and VM state to identify workload surges or failures, automatically initiate planning in response and actually execute the plan through the orchestrator.

5.3 Solution

The previous sections have discussed how innovations in virtualization technologies and SAN node specialization have laid the foundation for a *utility computing* platform. An important missing piece in the current systems is the planning scheme that produces placement of workloads to resources maximizing overall performance. Such a scheme has to be fast, scalable and robust to handle a wide variety of workloads and SAN environments. In this section, we describe a first step in this direction through our algorithm SPARK and start with the mathematical problem formulation.

5.3.1 Problem Formulation

A SAN in the data center has multiple types of nodes – application servers, switches, storage controllers – with available CPU and storage resources. Let the set of CPU resource nodes in the data center be denoted by $\mathcal{P} = \{P_k : 1 \leq k \leq |\mathcal{P}|\}$. This includes all CPU resources in the SAN where an application can be run – for example, an application server, a switch with a virtualizable CPU or a high-end storage controller. Each such CPU resource P_k has an associated limit on how much processing capacity it has, say in processor cycles per second, which we denote by $cap(P_k)$.

Similarly let the set of storage resource nodes in the data center be denoted by $\mathcal{S} = \{S_j : 1 \leq j \leq |\mathcal{S}|\}$. Each such storage resource S_j has an associated limit on how much storage capacity it has, say in GBs, denoted by $cap(S_j)$. These resources are connected to each other directly or indirectly based on the SAN topology (for example, Figure-47).

Along with these, there is a set of applications, say $\mathcal{A} = \{A_i : 1 \leq i \leq |\mathcal{A}|\}$ that need to run in the data center. These applications are packaged into virtual containers and can be executed on compatible virtualization platforms deployed at CPU nodes in the SAN. Each such application VM has a cpu resource requirement and a storage resource requirement. In general an application may require multiple cpu resources (for example, a cluster) and multiple storage resources (different storage for *log* data and *temp* data). However for ease of exposition we first present for the case of single cpu VM and single storage requirement for each application. That illustrates several of the key aspects and challenges of the problem. We later show in Section-5.3.5 how the algorithm works for the multiple resource case as well. We use *App.StgReq*, *App.CpuReq*, and *App.IOVol* to denote the amount of storage required (GB), the amount of CPU required (processor cycles per second), and the data I/O volume between the storage and CPU (post-cache) for the application *App*. Also, *App.Stg* and *App.Cpu* denote the storage and CPU nodes that host application data and VM respectively.

Cost Function: Given these, the question is where to allocate cpu and storage for each application, that is where should each application be run among the available CPU resources

and where should its data be placed among available storage resources. For each application, we use $\boldsymbol{Cost(A_i, S_j, P_k)}$ to denote the cost of running application A_i on storage resource node S_j and CPU resource node P_k. We also refer to it as the "*distance*" between S_j and P_k for A_i.

This cost function is used to capture the affinities, or lack thereof, between various application, storage and CPU groups. For example, if an application has preference for storage nodes of a certain type (for example,, RAID5) then the costs of assigning it to that storage node could be set lower. Also, if an application has a hard latency requirements then all storage-CPU pairs that cannot satisfy the latency bound have their cost set to infinity. In general, the cost function gives us a flexible way to account for multiple environmental characteristics:

- *SAN Fabric Characteristics*: The cost function can capture SAN characteristics like high bandwidth connectivity (through fabric or being on the same physical resource, for example, cpu and storage nodes on a controller).

- *Application QoS Requirements*: In order to restrict placement for an important application to specific nodes, the cost function can be adjusted to allow only those nodes that meet application QoS requirements.

- *Application Preferences*: For example, whether an application can use processors at a node in a specialized manner or conversely, if application VM is not compatible with the hypervisor available at a particular CPU node.

- *Cost of Storage*: The cost function can also be used to bias utilizing cheaper storage.

Many of these characteristics can be obtained automatically from the underlying SAN infrastructure and policy databases. Even a distinct *Cost Estimator Module* can be used. The SPARK algorithm and the framework presented below are flexible enough to work with any cost function provided as input. This makes SPARK extremely versatile in dealing with many peculiarities of a real SAN data center environment (see Section-5.3.5).

Given this cost function for all applications and resources, the goal of the placement is to select locations for the cpu and storage of each application so as to keep the overall cost for all applications small, that is,

$$min \sum_i Cost(A_i, A_i.Stg, A_i.Cpu) \quad (I)$$

while ensuring feasibility by not exceeding storage and CPU resource node capacities:

$$\forall j \sum_{i:A_i.Stg=S_j} A_i.StgReq \leq cap(S_j)$$

$$\forall k \sum_{i:A_i.Cpu=P_k} A_i.CpuReq \leq cap(P_k)$$

As mentioned above, SPARK can work with any cost function, but for concreteness in this chapter we use a cost function that captures desired features and complexity and yet is easy to describe. It tends to keep applications with high I/O volume requirement on storage-CPU pairs with small distances thus lowering the traffic on the SAN and increasing room for surges and other traffic. It is based on the I/O volume of the application and the distance between its cpu and storage nodes. It is given by $A_i.IOVol * dist(A_i.Stg, A_i.Cpu)$, where latter is the physical inter-hop distance between $A_i.Stg$ and $A_i.Cpu$ and is independet of applications. This function notably captures some of the desired features discussed earlier. For example, if a CPU at a storage controller is available, it would have low distance between its storage and cpu node, and thus an application with high I/O volume would be favored to go there, which improves performance and reduces the load on the SAN network.

Complexity and Relation to Other Problems: This problem (I) captures the basic questions inherent in placing CPU and storage in a coupled manner. The NP-Hard nature of the problem can be established by reducing to the 0/1 Knapsack problem. Even if a simpler case of our problem – involving two CPU nodes (one is a catch-all node of infinite capacity and large cost) and fixed application storage – can be solved, it can be used to solve the Knapsack problem[2]. Having to decide coupled placements for both storage and cpu with general cost/distance functions makes the problem more complex.

[2]Basically by making the second cpu node correspond to the knapsack and setting the costs and cpu requirements accordingly.

If either CPU or storage locations are fixed and only the other needs to be determined then there is related work along the lines of (a) File Allocation Problem [51, 32] placing files/storage assuming CPU is fixed; (b) Minerva, Hippodrome [5, 9] assume CPU locations are fixed while planning storage placement; (c) Generalized Assignment problems [158, 163]: Assigning tasks to processors (cpus) – they have been well-studied with several heuristics proposed but they do not consider the coupled storage and cpu allocation.

If the storage and cpu requirement for applications can always be split across multiple resource nodes, then one could also model it as a multi-commodity flow problem [46] – one commodity per application, introduce a source node for the cpu requirement of each application and a sink node for the storage requirement, with the source node connected to cpu resource nodes, storage resource nodes connected to the sink node and appropriate costs on the storage-cpu resource node pairs. However multi-commodity flow problems are known to be very hard to solve [102] in practice even for medium sized instances. And if the splitting is not justifiable for applications (for example, it requires sequential processing at a single server), then we would need an unsplittable flow [38] version for multi-commodity flows, which becomes even harder in practice.

Another important aspect of the problem is non-uniform costs. In a modern virtualized data center these costs vary depending on various factors like application preferences, storage costs, node heterogeneity and distances. If these variations were not present, i.e. costs for each application A_i were the same for all (S_j,P_k) pairs then the problem could be simplified to placing storage and cpu independently without coupling. In the next section, we discuss an algorithm INDV-GR that follows this approach. The evaluation and discussion of its performance in the general data center environment is given in the experimental section.

5.3.2 Algorithm

In this section, we begin by outlining two simpler algorithms – a greedy individual placement algorithm INDV-GR that places cpu and storage of applications independently in a natural greedy fashion and a greedy pairs placement algorithm PAIR-GR that considers applications

in a greedy fashion and places their cpu, storage pair simultaneously. The pseudocode for these is given as Algorithm-1 and Algorithm-2.

The INDV-GR algorithm (Algorithm-1) first places application storage by sorting applications by $\frac{IOVol}{App.StgReq}$ and greedily assigning them to storage nodes sorted by *BestDist*, which is the distance from the *closest* CPU node. Intuitively, INDV-GR tries to place highest I/O volume applications (normalized by their storage requirement) on storage nodes that have the closest CPU nodes. In the next phase, it will similarly place application VMs on CPU nodes.

Algorithm 1 INDV-GR: Greedy Individual Placement

```
1:  RankedAppsStgQ ← Apps sorted by IOVol/StgReq // decreasing order
2:  RankedStgQ ← Storage sorted by BestDist // increasing order
3:  while RankedAppsStgQ ≠ ∅ do
4:    App ← RankedAppsStgQ.pop()
5:    for (i=0; i<RankedStgQ.size; i++) do
6:      Stg ← RankedStgQ[i]
7:      if (App.StgReq ≤ Stg.AvlSpace) then
8:        Place App storage on Stg
9:        break
10:     end if
11:   end for
12:   if (App not placed) then
13:     Error: No placement found
14:   end if
15: end while
16: Similar for CPU placement
```

However, due to its greedy nature, a poor placement of application can result. For example, it can place an application with 600 units storage requirement, 1200 units I/O volume at a preferred storage node with capacity 800 units instead of choosing two applications with 500 and 300 units storage requirement and 900, 500 units I/O volume (cumulative volume of 1400). Also, INDV-GR does not account for storage-cpu affinities beyond using a rough *BestDist* metric. For example, if A_i storage is placed on S_j, INDV-GR does not especially try to place A_i cpu on the node closest to S_j.

This can potentially be improved by a greedy simultaneous placement. The PAIR-GR algorithm (Algorithm-2) attempts such a placement. It tries to place applications sorted by $\frac{IOVol}{CpuReq*StgReq}$ on storage, cpu **pairs** sorted by the distance between the nodes of the pair.

With this, applications are placed simultaneously into storage and cpu buckets based on their affinity measured by the distance metric.

Algorithm 2 PAIR-GR: Greedy Pairs Placement

```
1: RankedAppsQ ← Apps sorted by IOVol / (CpuReq*StgReq)
2: RankedPairsQ ← {Storage x CPU} sorted by distance
3: while RankedAppsQ ≠ ∅ do
4:    App ← RankedAppsQ.pop()
5:    for (i=0; i<RankedPairsQ.size; i++) do
6:       Stg ← RankedPairsQ[i].storage()
7:       CPU ← RankedPairsQ[i].cpu()
8:       if (App.StgReq ≤ Stg.AvlSpace AND App.CpuReq ≤ CPU.AvlCpu) then
9:          Place App storage on Stg, App cpu on CPU
10:         break
11:      end if
12:   end for
13:   if (App not placed) then
14:      Error: No placement found
15:   end if
16: end while
```

Notice that PAIR-GR also suffers from the shortcomings of the greedy placement where an early sub-optimum decision results in poor placement. Ideally, each storage (and cpu) node should be able to select application *combinations* that best minimize the overall cost value of the system. This hints at usage of **Knapsack**-like algorithms [46]. Secondly, an important missing component of these greedy algorithms is the fact that while applications have a certain preference order of resource nodes they would like to be placed on (based on the cost function), the resource nodes would have a different preference determined by their capacity and which application combinations fit the best. Matching these two distinct preference order indicates a connection to the **Stable-Marriage** problem [62] described below.

5.3.3 The SPARK algorithm

The above discussion about the greedy algorithms suggested an intuitive connection to the Knapsack and Stable Marriage problems. These form the basis for the design of SPARK. So we begin by a brief introduction of these problems.

Knapsack Problem[46, 143]: Given n items, a_1 through a_n, each item a_j has size s_j and a profit value v_j. The total size of the knapsack is S. The 0-1 knapsack problem asks for the

collection of items to place in the knapsack so as to maximize the profit. Mathematically:

$$max \sum_{j=1}^{n} v_j x_j \;\; subject\ to \;\; \sum_{j=1}^{n} s_j x_j \leq S$$

where $x_j = 0$ or 1 indicating whether item a_j is selected or not. This problem is known to be NP-Hard and has been well-studied for heuristics of near optimal practical solutions [83].

Stable Marriage Problem [62, 113]: Given n men and n women, where each person has a ranked preference list of the members of the opposite group, pair the men and women such that there are no two people of opposite group who would both rather have each other than their current partners. If there are no such people, then the marriages are said to be "stable". This is similar to the US residency-matching problem for medical graduate applicants where each applicant submits his ranked list of preferred medical universities and each university submits its ranked list of preferred applicants.

The Gale-Shapely *Proposal algorithm* [113] is the one that is commonly used in such problems. It involves a number of "rounds" (or iterations) where each man who is not yet engaged, "proposes" to the next most-preferred woman in his ordered list. She then compares the proposal with the best one she has so far and accepts it if it is higher than her current one and rejects otherwise. The man who is rejected becomes unengaged and moves to the next woman in his preference list. This iterative process is proved to yield stable results [113].

Notice that placing an application together on storage, cpu resource pair (S_j, P_k) (as done by PAIR-GR) would impact resource availability in all overlapping pairs (S_j, P_l) and (S_m, P_k). This *overlap* can have cascading consequences. This indicates that perhaps placing storage and CPU separately, yet coupled through affinities would hold the key to solving this problem. Combining this observation with knapsacks and the stable proposal algorithm leads us to SPARK.

SPARK: Consider a general scenario where say, the cpu part of applications has been placed and we have to find appropriate locations for storage. Each application A_i first constructs an ordered preference list of storage resource nodes as follows: Let P_k be the processor node where the CPU of A_i is currently placed. Then all S_j, $1 \leq j \leq \mathcal{S}$ are ranked in increasing

order of $Cost(A_i, S_j, P_k)$, or with our example cost funcation, $A_i.IOVol * dist(S_j, P_k)$[3]. Once the preference lists are computed, each application begins by proposing to the first storage node on its list (like in the stable-marriage scenario). On the receiving end, each storage node looks at all the proposals it received. It computes a profit value for each such proposal that measures the utility of that proposal. How to compute these profit values is discussed in Section-5.3.4. We pick the storage node that received the highest cumulative profit value in proposals and do a knapsack computation for that node [4]. This computation decides the set of applications to choose so as to maximize the total value without violating the capacity constraints at the storage resource. These chosen applications are considered accepted at the storage node. The other ones are rejected. The rejected ones move down their list and propose to the next candidate. This process repeats until all applications are accepted. The pseudocode for this part is given in Algorithm-3.

We assume a dummy storage node S_{dummy} (and similarly a dummy cpu node P_{dummy}) of unlimited capacity and large distance from other nodes. These would appear at the end of each preference list ensuring that the application would be accepted somewhere in the algorithm. This catch-all node provides a graceful termination mechanism for the algorithm.

Given these storage placements, the algorithm then decides the cpu placements for applications based on the affinities from the chosen storage locations. The pseudocode for the CPU part is similar to the one in Algorithm-3.

An illustration for the working of the SPARK-Stg and SPARK-Cpu rounds is given in Figure-49. SPARK-Stg brings $A_2.Stg$ closer to $A_2.Cpu$ and SPARK-CPU further improves by bringing $A_2.Cpu$ closer to $A_2.Stg$. Knapsacks help choose the A_1+A_2 combination over A_3 during placement.

Though the combination of SPARK-Stg and SPARK-Cpu address many possibilities well, they are not equipped to deal with scenarios like the one shown in Figure-50. Here a move of one end during a round of placement (either storage or cpu) doesnt improve the

[3]In case CPUs have not been placed (for example, when doing a fresh placement) we use the *BestDist* metric.

[4]Though knapsack problem is NP-Hard, there are known polynomial time approximation schemes for it [37] which work reasonably well in practice to give not exactly optimal but close to optimal solutions. We use one such package [144] here.

Algorithm 3 SPARK-Stg: Storage placement in SPARK

```
1: for all App in AppsQ do
2:    Create storage preference list sorted by distance from current App CPU
3:    Propose to best storage
4: end for
5: while (All apps not placed) do
6:    MaxStg ← StgQ[0]
7:    for all Stg in StgQ do
8:       Compute proposals profit
9:       MaxStg ← max(MaxStg.profit , Stg.profit)
10:   end for
11:   Knapsack MaxStg
12:   for all Accepted apps do
13:      Place App stg on MaxStg
14:   end for
15:   for all Rejected apps do
16:      Propose to next storage in preference list
17:   end for
18: end while
```

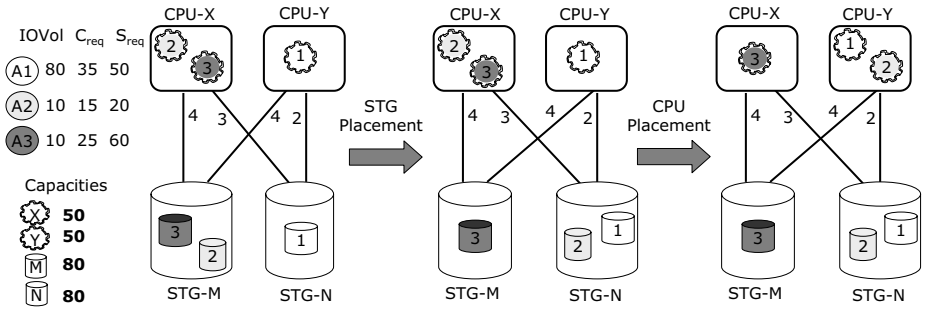


Figure 49: Placement in rounds.

placement but moving of both simultaneously does. This is where the SPARK-Swap step comes in. It takes two applications A_i and $A_{i'}$ and exchanges their cpu and storage locations if that improves the cost while still being within the capacity limits. For the scenario in the figure, individual rounds cannot make this move - during STG placement, M and O are equally preferable for A_1 as they have the same distance from A_4-cpu (X). Similarly during CPU placement. The SWAP step exchanges the STG/CPU pairs between A_4 and A_5.

Combining these insights, the SPARK algorithm is summarized in Algorithm-4. It proceeds iteratively in rounds. In each round it does a proposal-and-knapsack scheme for storage, a similar one for CPU, followed by a Swap step. It thus improves the solution iteratively, until a chosen termination criterion is met or until a local optimum is reached.

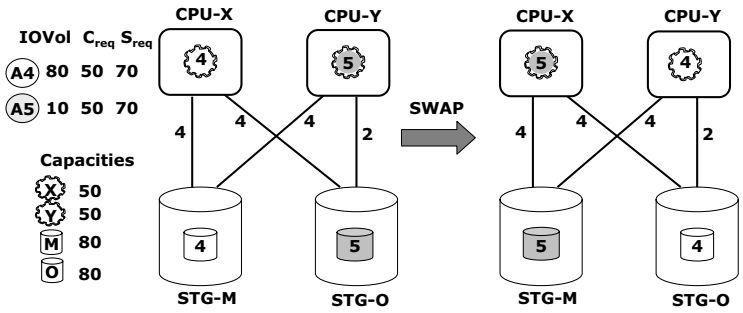


Figure 50: Swap exchanges STG/CPU pairs between A_4, A_5.

Algorithm 4 SPARK: Proposals-and-Knapsacks

```
1: MinCost ← ∞, SolutionCfg ← Ø
2: loop
3:    SPARK-Stg()
4:    SPARK-Cpu()
5:    SPARK-Swap()
6:    Cost ← 0
7:    for all App in AppsQ do
8:       Cost ← Cost + App.IOVol * dist(App.Stg, App.Cpu)
9:    end for
10:   if (Cost < MinCost) then
11:      MinCost ← Cost
12:      SolutionCfg ← current placement solution
13:   else
14:      break // termination by local optimum
15:   end if
16: end loop
17: return SolutionCfg
```

5.3.4 Computing Profit Values

One of the key steps in the SPARK algorithm is how to compute the profit values for the proposals. Recall that when a storage node S_j receives a proposal from an application A_i it first determines a profit value for that proposal which it then uses in the knapsack step to determine which ones to accept.

We distinguish two cases here based on whether A_i currently has a storage location or not (for example, if it got kicked out of its location, or it has not found a location yet). If it does, say at node $S_{j'}$ ($S_{j'}$ must be below S_j in A_i's preference list, otherwise A_i would not have proposed to S_j.), then the receiving node S_j would look at how much the system would save in cost if it were to accept A_i. This is essentially $Cost(A_i, S_{j'}, P_k) - Cost(A_i, S_j, P_k)$ where P_k is the current (fixed for this storage placement round) location of A_i's cpu. This

is taken as the profit value for A_i's proposal to S_j.

On the other hand, if A_i does not have any storage location or if A_i has storage at S_j itself, then the receiving node S_j would like to see how much more the system would lose if it did not select A_i. If it knew which storage node $S_{j'}$, A_i would end up, if not selected then the computation is obvious. Just taking a difference as above from $S_{j'}$ would give the necessary profit value. However where in its preference list A_i would end up if S_j rejects it, is not known at this time.

In the absence of this knowledge, a conservative approach is to assume that if S_j rejects A_i, then A_i would go all the way to the dummy node for its storage. So with this, the profit value can be set to $Cost(A_i, S_{dummy}, P_k) - Cost(A_i, S_j, P_k)$.

An aggressive approach is to assume that A_i would get selected at the very next storage node in its preference list after S_j. In this approach, the profit value would then become $Cost(A_i, S_{j'}, P_k) - Cost(A_i, S_j, P_k)$ where $S_{j'}$ is the node immediately after S_j in the preference list for A_i. The reason this is aggressive is that $S_{j'}$ may not take A_i either because it has low capacity or it has much better candidates to pick.

In this work we assume the conservative approach described above. Experiments show that the solutions computed by the SPARK algorithm are very close (within 4%) to the optimal with this approach for a range of scenarios. In future, we plan to examine sophisticated approaches including estimating probabilities that a given item would be accepted at a particular node based on history from past selections.

5.3.5 Features of Solution

In the previous section and examples of Figure-49, 50 we described how the proposals and knapsacks framework addresses various combinations and scenarios. In this section we discuss other salient features of this approach.

1. Being iterative: The SPARK algorithm proceeds in rounds iteratively improving the solution, thus offering a natural time-quality tradeoff if one is desired for time-constrained domains. The experimental section shows that even after one round SPARK yields better results than other greedy algorithms and multiple rounds further improve the value.

2. Flexible cost measure for (Application, Storage, CPU) triple: The algorithm and the framework allows the cost for each triple to be set independently. At any internal step, say if we have fixed the storage location S_j for A_i and trying to look for better candidate locations for cpu, then the algorithm looks at the $Cost(A_i, S_j, P_k)$ for all k, sorts the CPUs by increasing order of that Cost and proceeds with the proposals. The profits for the proposals are computed from these costs and the knapsacks are based on those profits. Thus, the algorithm is not tied to a particular cost function. This allows the user or administrator to capture special affinities between application, storage, CPU triples by controlling the cost function.

3. Ability to improve existing configurations, accounting for migration costs: Another important characteristic of SPARK is its ability to start from any existing configuration and account for migration costs in making placement decisions. This helps design placement plans that do not require extensive data and VM migration. This is an extremely desired feature and is in contrast to many other planning algorithms[5] that make decisions oblivious to current setups. SPARK accommodates this by instrumenting the $Cost$ function in the following manner. For each application A_i with storage at S_j and cpu at P_k, we add a factor $(\psi * A_i.StgReq * C_{S_j \rightarrow S_{j'}})$ to every $Cost(A_i, S_{j'}, P_*)$ value where $0 \leq \psi \leq 1$ is a customizable parameter that modulates the bias in deciding to migrate or not (ψ=0 implies no bias and each application is free to move) and $C_{S_j \rightarrow S_{j'}}$ is the cost of migrating storage from S_j to $S_{j'}$. Similarly for job migration costs from P_k. We envision ψ to be decided based on the desired speed of orchestrating the final SPARK plan. For example, to handle a workload surge in a rapid manner, a high ψ value would prevent extensive VM and data migration.

4. Handling dynamic scenarios: SPARK is extremely quick in generating placement decisions (see Section-5.4). This makes it especially adept at handling SAN dynamic scenarios. Below we briefly describe how it can plan to handle workload surges, planned downtime, growth and node failures.

[5]For example, for an LP formulation of this problem, the increased number of constraints would make the tractable problem size even smaller.

- **Surges:** Once a workload surge is identified using the VM and SAN management tools (Section-5.2), SPARK is initiated to plan application storage and cpu placement with the new amplified workload. The ψ can be set to a high value to limit the amount of VM/data migration suggested by SPARK. Once SPARK outputs a placement plan, the orchestrator can automatically execute necessary migrations stabilizing the system.

- **Planned Downtimes:** If planning sufficiently in advance for scheduled downtimes of hardware (for maintenance or upgrades), it might be preferable to have a more optimized solution than to limit migration. For such planning, ψ is set low and the SPARK is set to plan only with those SAN resources that will be up during that time. This output plan can then be executed and the system adequately prepared for the node downtimes.

- **Planned Growth:** Similar to downtimes, growth in application storage capacities or CPU requirements can be planned adequately in advance.

- **Node Failures:** SPARK can also similarly handle node failures as long as information about node state (for example, for a CPU resource node, the state of the VM) can be recreated. For stateless applications and when technologies exist to recreate state, SPARK can output a plan fast and minimize migration by setting a high ψ value.

5. Policy Constrained Placements: Typically data center administrators have many policies dictating that certain kind of applications can or cannot be placed on certain types of nodes. These constraints can be automatically encapsulated in the cost function by giving such incompatible application, storage, cpu triples a very high cost value and conversely highly preferred triples a low cost value. Incorporating such constraints in other algorithms like INDV-GR, PAIR-GR is complex as their greedy approaches are not flexible enough to account for such cost metrics.

6. Applications requiring multiple cpu and storage nodes: With the inherent flexibility in SPARK, it is easy to incorporate applications with multiple cpu and storage

node requirements. Such scenarios are common when multiple instances of an application are run concurrently (say on CPU nodes $C_k^1, \ldots C_k^m$) and data is shared between these instances. Further this data can also be split into multiple storage nodes (say $S_j^1, \ldots S_j^n$) for example, different locations for *log* and *temp* data. SPARK accounts for such situations by computing an aggregated cost value for all instances of a single application, given by

$$\sum_{p \in 1 \ldots m} \sum_{q \in 1 \ldots n} Cost(A_i, S_j^q, C_k^p)$$

whenever any instance of the application is considered for placement. This aggregated cost function captures the dependency across application instances as moving a single instance can affect the affinities for the entire application group. The greedy algorithms will struggle to account for this.

7. Based on well-studied problems: The fact that SPARK is fundamentally grounded in well-studied problems gives us hope that it will work well in a variety of situations and offer an opportunity for a more concrete theoretical analysis in future. The experimental section discussed next validates the belief of superior performance of SPARK in optimization quality as well as running time.

5.4 Experimental Evaluation

In this section, through a series of simulation based experiments, we evaluate the performance of SPARK for various workload and SAN environments. We also compare it with other candidate algorithms and Linear Programming based optimal solutions. Section-5.4.1 describes the experimental setup followed by the results. We summarize our findings in Section-5.4.6.

5.4.1 Setup

To evaluate SPARK, we simulated storage area networks of varying sizes and application workloads with different CPU, storage requirements and I/O volume rates.

As in any realistic SAN environment, the size of the SAN is based on the size of the application workload. We used simple ratios for obtaining the number of application servers, controllers and switches. For example, for a workload with 1000 applications (one CPU,

storage node per application), we used 333 (Ratio=3) application servers, 50 (Ratio=20) high end storage controllers (with CPUs) and 40 (Ratio=25) regular storage devices (without CPUs). We used the popular core-edge design of the SAN with storage controllers connected to application servers through three levels of core and edge switches. For the above example, we had 111 edge switches, 16 mid level core switches and 5 core switches (with CPUs). We used uniform CPU and storage capacities for nodes.

All these parameters are encapsulated into a single metric called **Problem Size** which is equal to the product of number of applications, number of CPU nodes and number of storage nodes. It roughly represents the complexity of the problem.

The CPU and storage requirements for applications are generated through a normal distribution with varying mean and standard deviation. The I/O volume rates were also varied using a normal distribution. We describe the exact mechanism used to obtain these values in Section-5.4.1.2 and evaluate our algorithm with different values in Section-5.4.3 and Section-5.4.4.

Another important input is the application cpu-storage distance matrix. In our experiments, we used same distance values independent of applications derived using an exponential function based on the physical inter-hop distance; the closest CPU-storage pairs (both nodes inside a high end storage controller) are set at distance 1 and for every subsequent level (core switch CPU and storage and then hosts and storage) the distance value is multiplied by a *distance-factor*. We present experiments with varying distance factor in Section-5.4.5.

5.4.1.1 Algorithms and Implementations

We compared the following algorithms in our evaluation. All algorithms were implemented in C++ and run on a Windows XP Pro machine with Pentium (M) 1.8 GHz processor and 512 MB RAM. For experiments involving time, results were averaged over multiple runs.

- **Individual Greedy Placement (INDV-GR)**: The greedy algorithm that independently places application storage and cpus as shown in Algorithm-1 (ref. Section-5.3).

- **Pairwise Greedy Placement (PAIR-GR)**: The greedy algorithm that places applications into best available storage-cpu pairs – Algorithm-2 (ref. Section-5.3).

- **OPT-LP**: The optimal solution obtained by the LP formulation. We used CPLEX Student [48] for obtaining integer solutions (it worked only for the smallest problem size) and popular MINOS [130] solver (through NEOS [60] web service) for fractional solutions in the [0,1] range for other sizes. We could only test upto 300 application nodes (problem size = 1.1 M) as the number of variables grew past the limits of the solvers after that.

- SPARK: Our SPARK algorithm as described in Section-5.3. It used the 0/1 knapsack algorithm contained in [143] with source code available from [144].

- SPARK-**R1**: The solution obtained after only a single round of SPARK. This helps illustrate the iteratively improving nature of SPARK.

5.4.1.2 Design of Experiments

We conducted the following set of experiments:

– Scalability Tests: An important requirement for SPARK is to be able to handle large data center environments. For validating this, we measure its performance with increasing size of the storage area network. As mentioned earlier, size of the SAN is computed based on the total number of applications in the workload. We varied this number from 10 (problem size 140) upto 2500 (problem size of 575 M). Please note that the OPT-LP implementation could only solve till 300 nodes (problem size 1.07 M). We report these experiments in Section-5.4.2.

– Varying Mean: Any fitting algorithm would be influenced by the required "tightness" of the fit. We varied the mean of the normal distributions used to generate workload CPU and storage requirements. For CPU requirements, the mean (μ) is varied through a parameter $\alpha \in (0, 1]$ using the formula $\mu = \frac{\alpha * cpu_{cap}}{N}$, where cpu_{cap} is the capacity of the CPUs

and N is the ratio of number of applications to available CPU nodes (similarly for storage requirements). It is easy to see that $\alpha = 1$ implies the strictest packing arrangement where the average CPU requirement is equal to the cpu_{cap} divided by the number of applications per CPU. Please note that high α values might not have any feasible solution for any solver.

– **Varying Std-dev**: Along with fitting tightness, uniformity of the workloads will also play an important role in the performance of various fitting algorithms. To evaluate this, we varied the standard deviation of the normal distribution as a function of $\beta \in [0, 1]$ using the formula $\sigma = \frac{\beta * cpu_{cap}}{N}$ where parameters are defined as above.

– **Varying Distance Factor (DF)**: DF determines the distances between storage nodes and CPU nodes at various levels. A higher distance factor implies a greater relative distance between two levels of CPU nodes from the underlying storage nodes. For example, if a CPU is accessing storage through a wide area network, its distance from the storage is much higher than any CPU accessing through the SAN fabric. This set of experiments provides interesting insights into placement characteristics of various algorithms (ref. Section-5.4.5).

5.4.2 Scalability Test: Varying SAN Size

In this experiment, we increased the size of the SAN with increasing number of applications in the workload. The problem size varied from 140 to 575M representing a small 10 applications workload increasing upto 2500. The other parameters were α=0.55, β=0.55, DF=2. We measured the quality of the optimization and solution processing time for all implementations. Recall that quality is measured using the cost metric given by cumulative sum of the application I/O volume times its storage-CPU pair distance.

Figure-51 shows the quality of optimization for INDV-GR, PAIR-GR, SPARK, SPARK-R1. Since OPT-LP only works upto a problem size of 1.1 M (as the number of variables in the LP formulation go past the limits of the solvers [60]), we show a zoomed graph for small sizes in Figure-52 and give exact cost measures in Table-21.

First notice the separation between the greedy algorithms and SPARK in Figure-51. Of

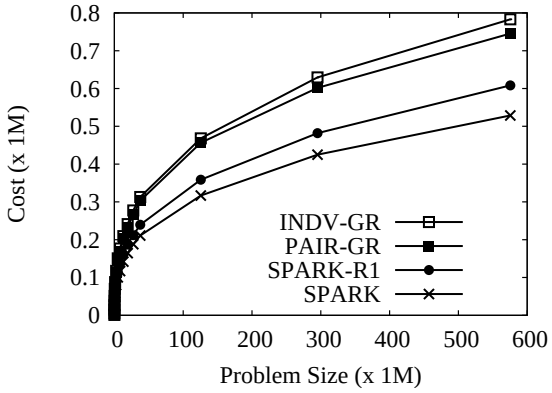


Figure 51: Quality with varying size

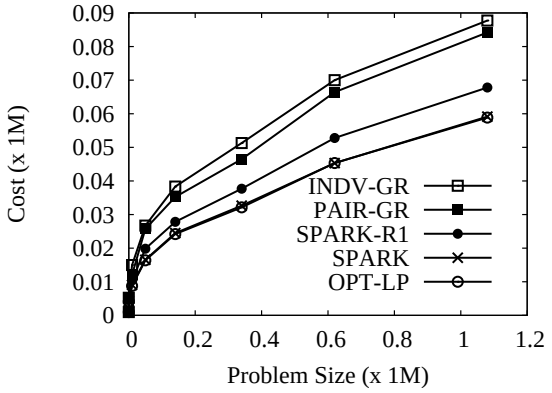


Figure 52: Varying Size: Comparison with OPT-LP

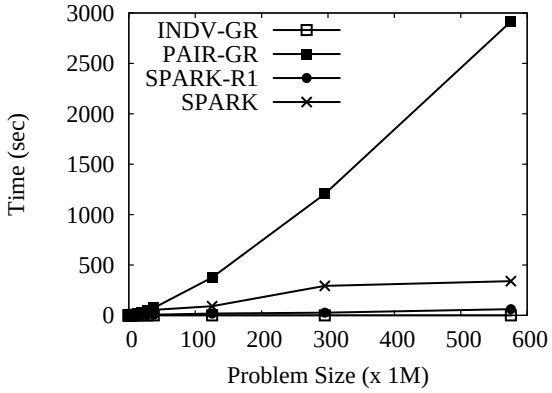


Figure 53: Time with varying size

the two greedy algorithms, PAIR-GR does better as it places applications based on storage-CPU pair distances, whereas INDV-GR's independent decisions do not capture that. It is interesting to see that SPARK-R1 performs better than both greedy algorithms. This is because of using knapsacks (thus picking the right application combinations to place on resources) and placing CPU based on the corresponding storage placement. With every subsequent round, SPARK iteratively improves to the best value shown.

Figure-52 shows the quality of all algorithms including OPT-LP for small problem sizes. While rest of the trends are similar, it is most interesting to see the closeness in curves of OPT-LP and SPARK. Table-21 shows the exact values of OPT-LP and SPARK optimization quality and SPARK is within 2% for all but one case where its within 4% of OPT-LP. This validates excellent optimization quality of SPARK.

Table 21: Comparison with OPT-LP

Size	SPARK-**R1**	SPARK	**OPT-LP**	**Difference**
0.00 M	986	986	986	0%
0.01 M	10395	9079	8752	3.7%
0.14 M	27842	24474	24200	1.1%
0.34 M	37694	32648	32152	1.5%
0.62 M	52796	45316	45242	0.1%
1.08 M	67805	59141	58860	0.4%
2.51 M	91717	79933	–	–

Figure-53 shows the time taken by implementations in giving a solution. As expected, INDV-GR is extremely fast since it independently places application CPUs and storage, thus complexity of $|Apps| * (|STG| + |CPU|)$. On the other hand PAIR-GR first generates all storage-CPU pairs and places applications on pairs giving it a complexity of $|Apps| * |STG| * |CPU|$. Each SPARK round would place storage and CPU independently. SPARK-R1 curve shows the time for the first round which is very small. The total processing time in SPARK would be based on the total number of rounds and complexity of the knapsacks in each round. As shown in the graph, it is extremely competitive taking only 333 seconds even for the largest size of the problem (575 M). Because of its iterative nature and reasonable quality of SPARK-R1, SPARK can actually be prematurely terminated if a quicker decision is required and thus still provide a better solution than comparable algorithms.

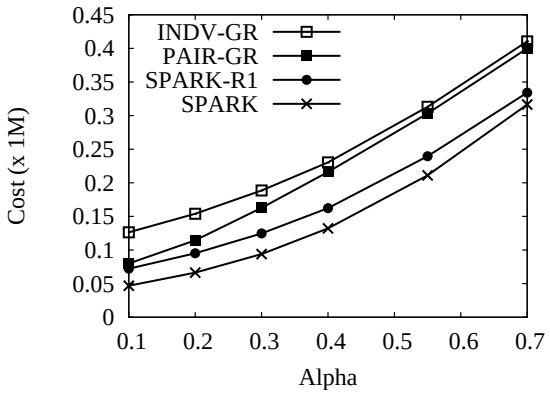


Figure 54: Quality with varying mean

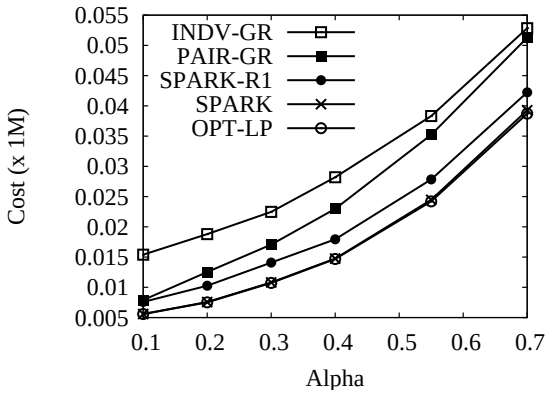


Figure 55: Varying Mean: Comparison with OPT-LP

5.4.3 Varying Mean

Our next experiments evaluate the algorithms with varying mean of the normal distribution used to generate CPU and storage requirements for the applications. As mentioned earlier, this mean is varied using the α parameter; higher α increases *tightness* of the fit by increasing the mean.

Figure-54 shows the results with α from 0.1 to 0.7 for a problem size of 37 M (1000 applications), $\beta = 0.55$ and $DF = 2$. The costs of all algorithms increase with increasing α. This is expected since tighter fittings will have fewer combinations that fit, resulting in overall increased cost.

Of the greedy algorithms, PAIR-GR worsens at a faster rate which is due to increasing impact of the *overlap* effect as mentioned in Section-5.3. Comparatively, SPARK continues

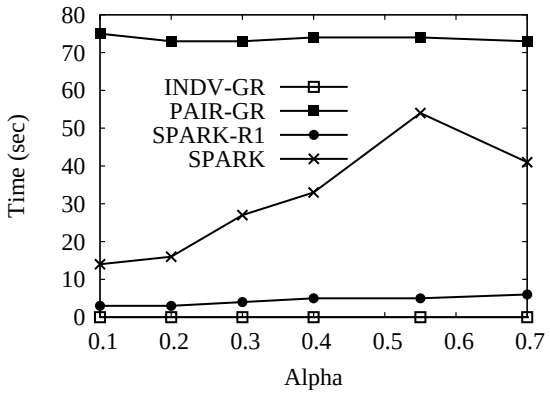


Figure 56: Time with varying mean

to outperform other algorithms for all α values. Also, as earlier, we plotted a similar graph for a smaller problem size of 0.14M (150 applications) to compare with OPT-LP in Figure-55. SPARK remains close to OPT-LP throughout, validating its power to find solutions with varying tightness of the fitting.

For processing time (Figure-56), the greedy algorithms remain constant with increasing α values as they only traverse storage and cpu lists per application. On the other hand, SPARK might need additional rounds of placements in order to converge to a local optimum value. As seen in the graph, while SPARK-R1 remains small, the total time increases overall due to increased number of rounds (varies between 3 and 7 in these experiments). Note that due to convergence to a *local* optimum, there would not be a consistent pattern with increasing α. The objective of the time experiments is to ensure that varying fitting-tightness does not significantly deteriorate SPARK in quality or processing time.

5.4.4 Varying Std-Dev

Similar to tightness of the fit, a relevant parameter is the uniformity of the workloads. For example, if all workloads have same CPU and storage requirements and same throughout rates, all solutions tend to have the same cost metric (as not much can be done with different combinations of applications during placements). As workloads become less uniform, there are different fittings that might be possible and different implementations will react differently.

Figure-57 shows the performance of the algorithms when β is varied from 0 to 0.55 for

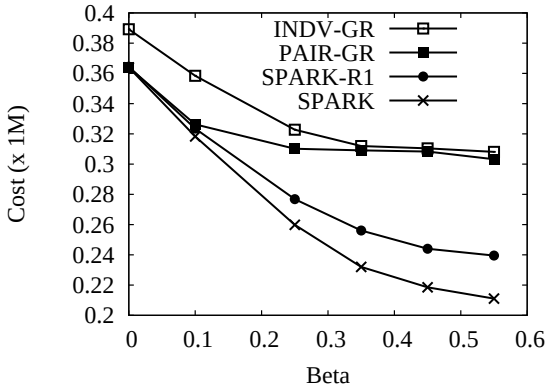


Figure 57: Quality with varying std-dev

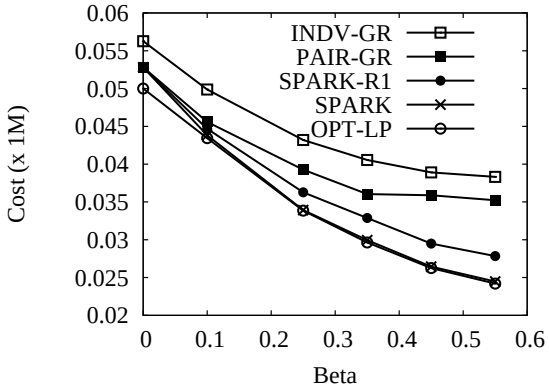


Figure 58: Varying std-dev: Comparison with OPT-LP

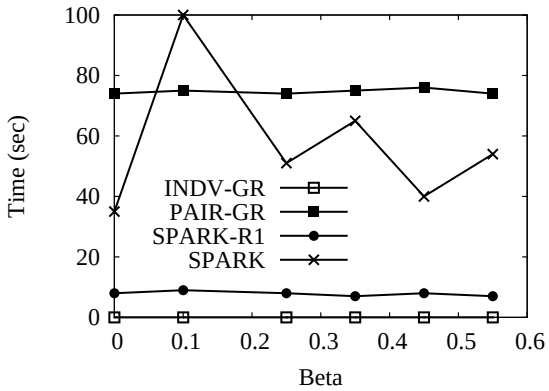


Figure 59: Time with varying std-dev

a problem size of 37 M (1000 applications workload), α=0.55 and DF=2. Notice that as β and thus non-uniformity of workloads increases, costs drop for all algorithms. This is because with availability of many applications with low resource requirements, it is possible to fit more applications at smaller distances. However, SPARK reacts the best to this and significantly drops in cost for higher β values. This is explained through the knapsacks component of SPARK as in case of greedy algorithms, an early sub-optimum decision can cause poor fitting (for example, choosing 6 before 5 & 3 were available for an 8 capacity resource).

Figure-58 shows the graph with OPT-LP for smaller problem size of 0.14M (150 applications). Once again, SPARK is able to maintain its effectiveness in comparison to OPT-LP. This means that SPARK adjusts superbly with various workload characteristics. The times of different algorithms for the larger 37M problem with varying β are shown in Figure-59. As mentioned earlier, greedy algorithms tend to maintain constant time based on application and storage-cpu lists. In contrast, total time in SPARK is impacted by the number of rounds required to converge to the local optimal and does not have a consistent pattern. In this case, it used 3, 10, 6, 8, 5 and 7 rounds. However, it is important to note that the total time is still small and does not compromise on SPARK's application to addressing dynamic SAN events like workload surges.

5.4.5 Varying Distance Factor

The last set of experiments vary the distance factor (DF) that is used to obtain distance values between CPU and storage nodes. As mentioned earlier, we used uniform distance values across applications (that is, distance between a CPU and storage node is same for all applications) and the distance value was obtained using the formula DF^p where p is the physical inter-hop distance between nodes. A higher DF value implies that distance values increase much more rapidly with every hop, for example, due to increased latencies at switches, or going over a LAN or WAN for farther nodes.

The objective of this experiment is to better understand the characteristics of the algorithms as differences in fitting arrangements would be amplified at higher DF values

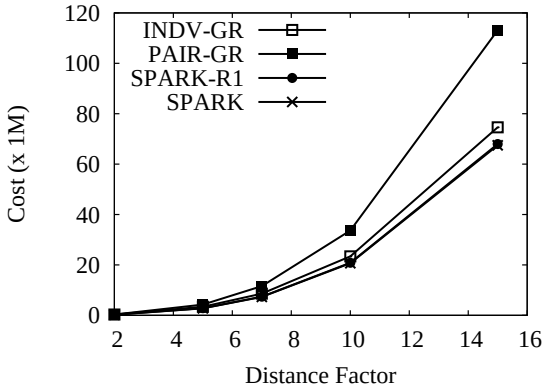


Figure 60: Quality with distance factor

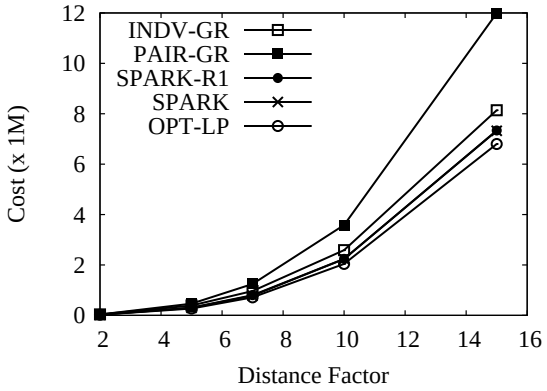


Figure 61: Comparison with OPT-LP

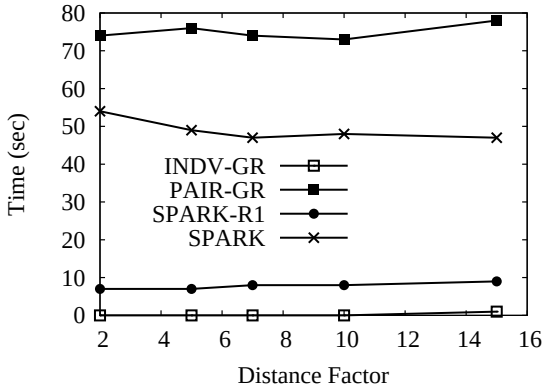


Figure 62: Time with distance factor

(fitting an application at distance i vs. fitting it at distance j has the cost difference of $IOVol * (DF^j - DF^i)$).

Figure-60 plots the graph as DF is varied from 2 to 15 for a problem size of 37 M (1000 applications) and α=β=0.55 (due to small difference between SPARK and SPARK-R1 relative to the scale of the graph, they appear as if a single line). The most interesting, and in fact surprising observation is the performance of INDV-GR. Not only does it outperform PAIR-GR at higher DFs but it comes reasonably close to SPARK. On close inspection, we found that INDV-GR was performing remarkably at placing applications at higher distances, that is, even though it fit fewer applications (and I/O volume) at lower distances (p=0 and p=1), it outperformed other algorithms by fitting more at p=3 and less at p=4. It requires further analysis to fully explain this characteristic[6], but we believe it can hint at improving SPARK performance for higher DFs as well. For a smaller problem size of 0.14 M, shown in Figure-61, the OPT-LP is able to separate itself from SPARK at higher DF values, though SPARK still remains within 8%. This is due to the amplification as well. Finally, Figure-62 shows that the processing time does not vary much with changing DF indicating it does not impact the fitting.

5.4.6 Summary of Results

Below, we summarize our key findings:

- SPARK is very **scalable** in optimization quality and processing time. With increasing size of the SAN and workloads, SPARK continues to perform (30-40%) better than other candidate algorithms and is **within 4% of LP based solutions** (tested for smaller problem sizes). It is also very fast to compute results taking 5.5 minutes for the largest instance with 2500 applications.

- The **iterative** nature of SPARK and its good performance even after a single round (SPARK-R1 in graphs) allows improving any initial configuration, and an attractive property of trading off quality with speed.

[6]It is impacted by how applications are fit at lower distances, as only the ones that do not fit there are candidates for fitting at higher distances

- SPARK is incredibly **robust** with changing workload characteristics like tightness of the fit and uniformity of workloads. For changing α, β and DF parameters, it maintains its superior performance over other algorithms and closeness to the LP solutions.

5.5 Summary

In this chapter we presented a novel algorithm, called SPARK that optimizes coupled placement of application computational and storage resources on nodes in a modern virtualized SAN environment. By effectively handling proximity and affinity relationships of CPU and storage nodes, SPARK produces placements that are within 4% of the optimal lower bounds obtained by LP formulations. Its fast running time even for very large instances of the problem make it especially suitable for storage service provider environments that deal with highly dynamic scenarios like workloads surges, scheduled downtime, growth and node failures. Among its other features, SPARK can iteratively improve any existing resource placement, accounting for migration costs and has inherent built-in flexibility to handle various placement constraints that impact the choice of resources where an application can be placed.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

As enterprises generate more and more data, they are faced with the complex task of managing storage systems that store this data. Management of large storage infrastructures can be very expensive, often accounting for nearly 75% of the total cost of ownership [25]. This has forced enterprises to look for more cost-effective data storage models. One paradigm finding success in alleviating these costs is the Storage-as-a-Service model, in which enterprises outsource their storage to an external storage service provider (SSP) by storing data at a remote SSP-managed site and accessing it over a high speed network. The storage-as-a-service model reduces storage management costs for enterprises and provides them with an on-demand storage infrastructure, growing or shrinking according to their needs. Additionally, storage service providers provide superior disaster recovery and content dissemination solutions. However, this model faces many unique technical challenges.

In this dissertation, we have contributed to addressing two core challenges faced by the storage-as-a-service model. The first and foremost challenge is that of security and access control. Enterprises find it tough to trust the SSP for preserving data confidentiality and enforcing access control over their stored data. To enforce access control in the absence of a trusted reference monitor, we propose using efficient cryptographic techniques to *embed* access control into the stored data, ensuring that users can decrypt only the data that they are authorized to access. Along with core filesystem data access, we also propose a similar embedding technique to support another access control aware application – multiuser filesystem search. Unlike traditional approaches that require a trusted access control monitor to filter query results and are prone to inference attacks, our access control embedding approach does not require a trusted engine at runtime and is more resilient to various attacks.

The second important challenge addressed in this dissertation is the management of

highly dynamic SSP storage data centers. To correctly integrate client-initiated changes, SSP administrators have to carefully analyze the impact of the proposed change on the rest of the system and also manually plan for resource allocation in response. This manual process is error prone, inefficient and slow. In this dissertation, we propose two autonomic techniques that proactively analyze the impact of a change on the storage area network and perform efficient allocation of resources to client workloads in response to their new requirements. This reduces administrator involvement in the process and aims to deliver a true on-demand storage infrastructure.

Specifically, this dissertation has made following new contributions to the area of the enterprise storage-as-a-service model:

- *Access Control with xACCESS*: We described xACCESS, an access control system for enterprise storage-as-as-service environments. xACCESS uses novel cryptographic access control primitives (CAPs) to *embed* access control into stored data, eliminating the requirement of a trusted access control enforcement engine. We showed how an expressive UNIX-like access control model can be supported using xACCESS with completely in-band key management that minimizes user involvement. Further, we demonstrated the greater efficiency of our filesystem achieved by using symmetric key cryptography for metadata operations. In our initial experiments, we outperformed related proposals by over 40% on a number of micro and macro benchmarks. We also analyzed the privacy aspects of our access control model and developed enhancements that provide more secure and convenient data sharing mechanisms.

- *Secure Multiuser Filesystem Search*: We also developed a new enterprise search architecture that provides access control aware search resilient to common inference attacks over SSP-stored data. Similar in concept to xACCESS, our search architecture *embeds* access control into search indices using a novel technique of access control barrels (ACB) and eliminates the requirement of a trusted monitor for filtering query results. We also showed how the distributed architecture helps in achieving greater indexing efficiency by using underutilized enterprise machines.

- *Autonomic Management of SSP Data Centers*: To assist SSP administrators in managing dynamic storage environments, we developed Zodiac, a proactive *what-if* analysis engine that can analyze the impact of a proposed change even before applying that change, thus identifying potential configuration and performance errors. We demonstrated how our three proposed optimizations allow Zodiac to scale well with increasing size of the storage area network and complexity of storage policies. We also developed SPARK, a unique resource allocation framework that efficiently re-allocates storage and CPU resources in response of a client workload surge, node failures, growth or planned downtime. Through our experiments, we also demonstrated its superior allocation quality, speed and robustness with varying system parameters.

6.1 Open Problems and Future Directions

This thesis has made a number of contributions in security and management of enterprise storage-as-a-service model. We believe that the techniques proposed in this dissertation lend themselves well for many interesting extensions in future. In this section, we describe some important directions for future work.

6.1.1 Access Control Models

The access control system proposed in this dissertation provided cryptographic access control primitives for the UNIX access control model. An important extension to this work is support of other access control models, most notably Windows. Windows offers a richer access control model with more permission settings. While the most frequently used permissions are similar in semantics to UNIX, there exist few unique permissions that can not be supported by easy extensions to UNIX based CAPs, for example, only creating or only deleting contents of a directory, taking ownership of a filesystem object or ability to change access privileges. It requires further exploration of cryptographic literature or development of new cryptographic schemes to provide equivalent semantics for such permissions. For example, append-only signatures [101] might be used for supporting creation-but-not-deletion of directory contents. Similarly, development of efficient schemes that provide asymmetric data encryption to support write-only permissions is an important direction of future work.

6.1.2 Access Control for Application Contexts

The proposed access control embedding approach provides techniques for enforcing access control over SSP-stored ***data*** without placing undue trust in the service provider. An interesting direction of future work is to use similar techniques for enforcing access control for application contexts. In other words, can we support a `setuid` form of permission that changes the effective user ID and thus, effective access privileges upon execution of an application command? The effective ID is applicable only for the execution of that command. In the storage-as-a-service model, this implies that a user can obtain access to certain encryption keys only during the execution of a particular command. We are investigating mechanisms in which virtualization based technologies can be used to enforce such access control. For example, the owner of the `setuid` program can create an application context inside a virtual machine (similar to a virtual appliance [186]) and embed encryption keys into the virtual machine. Then the virtual machine can be stored as a regular file at the SSP. Users can download the virtual machine and execute using the privileges (keys) of the owner user without obtaining plaintext access to the keys.

6.1.3 Other Security Issues

This thesis has contributed towards addressing unique data confidentiality and access control challenges in the decentralized storage-as-a-service model. Notwithstanding that these are two key components of an enterprise's security strategy, other important challenges like data integrity, intrusion detection and denial of service (DoS) attacks also need to be addressed for a comprehensively secure service environment. Data integrity solutions need to prevent against unintentional or malicious tampering with data stored at the SSP, either through an enterprise user or even the SSP [106, 67]. Intrusion detection may also require new methodologies as now the SSP is tasked with identifying unauthorized intrusion attempts which might require isolation of user requests across different client enterprises. Protection against DoS attacks is also crucial as a single SSP can impact various enterprise storage access services, and it is also possible that the attack could be perpetrated by the SSP itself. Investigation of such security solutions is a fertile area for future research.

6.1.4 Search Integrity

Many research efforts have proposed techniques to ensure integrity of data stored at the untrusted SSP [106, 67] by allowing users to detect unauthorized writes to data. Similar techniques may be required to ensure *integrity* of results in the multiuser search hosted at the SSP. A malicious SSP could return wrong files (that do not contain the query keywords) or not-return correct files (that contained the keywords) or just incorrectly rank the results (if using the $BDI - T$ approach). Search integrity techniques would provide a mechanism to detect such malicious behavior.

6.1.5 Autonomic Monitoring and Execution

In this dissertation, we proposed autonomic techniques that provide *Analysis* and *Planning* capabilities for efficient management of SSP data centers. Along with these, traditional autonomic computing consists of *Monitoring* and *Execution* capabilities as well (together, these four capabilities form a part of the MAPE loop [98]). While many existing products can be leveraged for monitoring and execution [88, 54, 181, 182], it still requires deeper analysis for integrating these capabilities, especially under the performance and response time requirements of the dynamic SSP data centers.

REFERENCES

[1] Adra, B., Blank, A., Gieparda, M., Haust, J., Stadler, O., and Szerdi, D., "Advanced POWER virtualization on IBM eserver P5 servers: Introduction and basic configuration," *IBM Redbook*, October 2004.

[2] Adya, A., Bolosky, W., Castro, M., Cermak, G., Chaiken, R., Douceur, J., Howell, J., Lorch, J., Theimer, M., and Wattenhofer, R., "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[3] Agrawal, D., Giles, J., Lee, K., Voruganti, K., and Adib, K., "Policy-Based Validation of SAN Configuration," in *Proceedings of the IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*, p. 77, 2004.

[4] Altiris Inc. *http://www.altiris.com/*, (Accessed February 8, 2007).

[5] Alvarez, G., Borowsky, E., Go, S., Romer, T., Szendy, R., Golding, R., Merchant, A., Spasojevic, M., Veitch, A., and Wilkes, J., "Minerva: An Automated Resource Provisioning tool for large-scale Storage Systems," *ACM Transactions on Computer Systems*, vol. 19, no. 4, 2001.

[6] Amazon Simple Storage Service. *http://aws.amazon.com/s3*, (Accessed February 8, 2007).

[7] Anderson, E., "Simple table-based modeling of storage devices," *HP Labs Tech Report HPL-SSP-2001-4*, 2001.

[8] Anderson, E., Hall, J., Hartline, J., Hobbs, M., Karlin, A., Saia, J., Swaminathan, R., and Wilkes, J., "An experimental study of data migration algorithms," in *Proceedings of the International Workshop on Algorithm Engineering*, pp. 28–31, 2001.

[9] Anderson, E., Hobbs, M., Keeton, K., Spence, S., Uysal, M., and Veitch, A., "Hippodrome: Running Circles Around Storage Administration," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[10] Anderson, E., Kallahalla, M., Spence, S., Swaminathan, R., and Wang, Q., "Ergastulum: Quickly finding near-optimal Storage System Designs," *HP Labs Tech Report HPL-SSP-2001-5*, 2001.

[11] Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Kalantar, M., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B., "Oceano - SLA-based management of a computing utility," in *Proceedings of the IFIP/IEEE Symposium on Integrated Network Management*, pp. 855–868, 2001.

[12] ARSENAL DIGITAL. *http://www.arsenaldigital.com*, (Accessed February 8, 2007).

[13] AUREMA, "Application performance and server consolidation solutions," *http://www.aurema.com/*, (Accessed February 8, 2007).

[14] BANDARA, A., LUPU, E., and RUSSO, A., "Using Event Calculus to Formalise Policy Specification and Analysis," in *Proceedings of the IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*, 2003.

[15] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., and WARFIELD, A., "Xen and the art of virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pp. 164–177, 2003.

[16] BAWA, M., BAYARDO, R., and AGARWAL, R., "Privacy-preserving indexing of documents on the network," in *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2003.

[17] BERENBRINK, P., BRINKMANN, A., and SCHEIDELER, C., "SimLab - A Simulation Environment for Storage Area Networks," in *Proceedings of the Workshop on Parallel and Distributed Processing (PDP)*, 2001.

[18] BISHOP, M., *Computer Security: Art and Science*. Addison-Wesley, 2002.

[19] BLAZE, M., "A Cryptographic File System for Unix," in *Proceedings of the ACM Conference on Computer and Communications Security*, 1993.

[20] BLOM, R., "An optimal class of symmetric key generation systems," in *Proceedings of the EUROCRYPT Workshop on Advances in Cryptology: Theory and Application of Cryptographic Techniques*, pp. 335–338, 1985.

[21] BONEH, D., CRESCENZO, G., OSTROVSKY, R., and PERSIANO, G., "Public key encryption with keyword search," in *Proceedings of the International Cryptology Conference (EUROCRYPT)*, 2004.

[22] BONEH, D. and FRANKLIN, M., "Identity based encryption from the Weil pairing," *SIAM Journal of Computing*, vol. 32, no. 3, 2003.

[23] BONNIE BENCHMARK *http://www.textuality.com/bonnie*, (Accessed February 8, 2007).

[24] BRESSOUD, T., KOZUCH, M., and NATH, P., "Pushing the virtual envelope: Internet suspend/resume over HTTP(S)," in *Proceedings of the International Conference on Web Technologies, Applications, and Services*, pp. 69–76, 2005.

[25] BRICK, F., "Are you ready to outsource your storage?," *Computer Technology Review*, June 2003.

[26] BUCY, J., GANGER, G., GRIFFIN, J., SCHINDLER, J., SCHLOSSER, S., WORTHINGTON, B., and PATT, Y., "The DiskSim Simulation Environment," *CMU Technical Report, CMU-CS-03-102*, 2003.

[27] BUTLER GROUP, "Unlocking value from text-based information," *Review Journal Article*, March 2003.

[28] BÜTTCHER, S. and CLARKE, C., "A security model for full-text file system search in multi-user environments," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[29] CAPPELLI, D. and KEENEY, M., "Insider threat: Real data on a real problem," *CERT Technical Report* *http://www.cert.org/nav/present.html*, (Accessed February 8, 2007).

[30] CARD, R., TS'O, T., and TWEEDIE, S., "Design and Implementation of the Second Extended Filesystem," in *Proceedings of the First Dutch International Symposium on Linux*, 1995.

[31] CARNEGIE MELLON SOFTWARE ENGINEERING INSTITUTE (CERT) *http://www.cert.org*, (Accessed February 8, 2007).

[32] CERI, S., MARTELLA, G., and PELAGATTI, G., "Optimal file allocation in a computer network: A solution method based on the knapsack problem," *Computer Networks*, vol. 6, pp. 345–357, November 1982.

[33] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., and LAM, M., "The Collective: A cache-based system management architecture," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[34] CHANG, Y. and MITZENMACHER, M., "Privacy preserving keyword searches on remote encrypted data," in *Proceedings of the Applied Cryptography and Network Security*, 2005.

[35] CHASE, J., ANDERSON, D., THAKAR, P., VAHDAT, A., and DOYLE, R., "Managing energy and server resources in hosting centers," in *Proceedings of ACM Symposium on Operating System Principles*, pp. 103–116, 2001.

[36] CHAUDHURI, S. and NARASAYYA, V., "AutoAdmin 'What-if' Index Analysis Utility," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998.

[37] CHEKURI, C. and KHANNA, S., "A PTAS for the multiple knapsack problem," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 213–222, 2000.

[38] CHEKURI, C., KHANNA, S., and SHEPHERD, F., "The all-or-nothing multicommodity flow problem," in *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pp. 156–165, 2004.

[39] CHOR, B., GOLDREICH, O., KUSHILEVITZ, E., and SUDAN, M., "Private information retrieval," in *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1995.

[40] CISCO MDS 9000, "Advanced Services Module," *http://www.cisco.com/en/US/products/ps5991*, (Accessed February 8, 2007).

[41] Cisco MDS 9000 Family SANTap Service, "Enabling Intelligent Fabric Applications," *CISCO Whitepaper*, 2005.

[42] Clark, C., Fraser, K., Hand, S., Hansen, J., Jul, E., Limpach, C., Pratt, I., and Warfield, A., "Live migration of virtual machines," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[43] Clark, T., *Designing Storage Area Networks.* Addison-Wesley, 1999.

[44] Cohen, I., Chase, J., Goldszmidt, M., Kelly, T., and Symons, J., "Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[45] Computer Associates BrightStor *http://www.ca.com*, (Accessed February 8, 2007).

[46] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to Algorithms.* McGraw-Hill Book Company, 2001.

[47] Coveo Enterprise Search *http://www.coveo.com*, (Accessed February 8, 2007).

[48] CPLEX 8.0 Student Edition for AMPL *http://www.ampl.com/DOWNLOADS/cplex80.html*, (Accessed February 8, 2007).

[49] Damianou, N., Dulay, N., Lupu, E., and Sloman, M., "The Ponder Policy Specification Language," in *Proceedings of the IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*, 2001.

[50] DMTF Common Information Model *http://www.dmtf.org*, (Accessed February 8, 2007).

[51] Dowdy, L. W. and Foster, D. V., "Comparative models of the file assignment problem," *ACM Surveys*, vol. 14, pp. 287–313, 1982.

[52] Doyle, R., Chase, J., Asad, O., Jin, W., and Vahdat, A., "Model-based resource provisioning in a web service utility," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 2003.

[53] Eastlake, E., "US secure hash algorithm I," *http://www.ietf.org/rfc/rfc3174.txt*, 2001.

[54] EMC Control Center *http://www.emc.com*, (Accessed February 8, 2007).

[55] eVault. *http://www.evault.com*, (Accessed February 8, 2007).

[56] Ferraiolo, D. and Kuhn, D., "Role Based Access Control," in *Proceedings of the 15th National Computer Security Conference*, 1992.

[57] Fischer-Hubner, S., "Towards a Privacy-Friendly Design and Usage of IT-Security Mechanisms," in *Proceedings of the 17th National Computer Security Conference*, 1994.

[58] FISCHER-HUBNER, S., "Considering Privacy as a Security-Aspect: A Formal Privacy-Model," *DASY Papers No. 5/95, Computer and System Sciences, Copenhagen Business School*, 1995.

[59] FISCHER-HUBNER, S. and OTT, A., "From a Formal Privacy Model to its Implementation," in *Proceedings of the 21st National Information Systems Security Conference*, 1998.

[60] FOR OPTIMIZATION, N. S. *http://www-neos.mcs.anl.gov*, (Accessed February 8, 2007).

[61] FU, Z., WU, S., HUANG, H., LOH, K., GONG, F., BALDINE, I., and XU, C., "IPSec/VPN Security Policy: Correctness, Conflict Detection, and Resolution," in *Proceedings of the Workshop on Policies for Distributed Systems and Networks*, 2001.

[62] GALE, D. and SHAPLEY, L. S., "College Admissions and Stability of Marriage," *American Mathematical Monthly*, pp. 9–14, 1962.

[63] GANGER, G., STRUNK, J., and KLOSTERMAN, A., "Self-* Storage: Brick-based Storage with Automated Administration," *CMU Tech Report CMU-CS-03-178*, 2003.

[64] GANGER, G. R., STRUNK, J. D., and KLOSTERMAN, A. J., "Self-* Storage: Brick-based Storage with Automated Administration," *Carnegie Mellon University Technical Report, CMU-CS-03-178*, August 2003.

[65] GARTNER GROUP. *http://www.gartner.com*, (Accessed February 8, 2007).

[66] GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., and ZELENKA, J., "A Cost-effective, High-bandwidth Storage Architecture," in *Proceedings of the International conference on Architectural support for programming languages and operating systems (ASPLOS)*, pp. 92–103, 1998.

[67] GOH, E., SHACHAM, H., MODADUGU, N., and BONEH, D., "SiRiUS: securing remote untrusted storage," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2003.

[68] GOLDBERG, R., "Survey of virtual machine research," *IEEE Computer*, vol. 7, pp. 34–45, June 1974.

[69] GOOGLE DESKTOP *http://desktop.google.com*, (Accessed February 8, 2007).

[70] GOOGLE ENTERPRISE SEARCH *http://www.google.com/enterprise*, (Accessed February 8, 2007).

[71] GOOGLE MAIL *http://www.gmail.com*, (Accessed February 8, 2007).

[72] GRUNBACHER, A. and NUREMBERG, A., "POSIX Access Control Lists on Linux," *http://www.suse.de/~agruen/acl/linux-acls/online*, (Accessed February 8, 2007).

[73] HACIGUMUS, H., IYER, B., LI, C., and MEHROTRA, S., "Executing SQL over encrypted data in the database service provider model," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.

[74] HACIGUMUS, H., MEHROTRA, S., and IYER, B., "Providing Database as a Service," in *Proceedings of the International Conference on Data Engineering (ICDE)*, p. 29, 2002.

[75] HALL, J., HARTLINE, J., KARLIN, A., SAIA, J., and WILKES, J., "On algorithms for efficient data migration," in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pp. 620–629, 2001.

[76] HARRINGTON, A. and JENSEN, C., "Cryptographic access control in a distributed file system," in *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT)*, pp. 158–165, 2003.

[77] HASAN, R., YURCIK, W., and MYAGMAR, S., "The evolution of storage service providers: techniques and challenges to outsourcing storage," in *Proceedings of the ACM workshop on Storage security and survivability (StorageSS)*, pp. 1–8, 2005.

[78] HE, D., "Cleaned W3C Subcollections," *http://www.sis.pitt.edu/~daqing/w3c-cleaned.html*, (Accessed February 8, 2007).

[79] HEWLETT PACKARD. *http://www.hp.com*, (Accessed February 8, 2007).

[80] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., and WEST, M. J., "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51–81, 1988.

[81] HP STORAGEWORKS SAN 2005.

[82] HUSTON, L., SUKHTANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G., RIEDEL, E., and AILAMAKI, A., "Diamond: A storage architecture for early discard in interactive search," in *Proceedings of the USENIX Conference on File and Storage Technologies*, pp. 73–86, 2004.

[83] IBARRA, O. and KIM, C., "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems," *Journal of the ACM*, vol. 22, no. 4, pp. 463–468, 1975.

[84] IBM, "Storage virtualization: Value to you," *IBM Whitepaper*, May 2006.

[85] IBM, "SAN volume controller for Cisco MDS 9000," *http://www-03.ibm.com/servers/storage/software/virtualization/svc_cisco%/index.html*, (Accessed February 8, 2007).

[86] IBM, "System Storage DS8000," *http://www-03.ibm.com/servers/storage/disk/ds8000/*, (Accessed February 8, 2007).

[87] IBM TIBURON PROJECT *http://www.almaden.ibm.com/StorageSystems/Advanced_Storage_Systems/Tibu%ron*, (Accessed February 8, 2007).

[88] IBM TOTALSTORAGE PRODUCTIVITY CENTER 2005.

[89] IBM WEBSPHERE INFORMATION INTEGRATOR *http://www-306.ibm.com/software/data/integration/db2ii*, (Accessed February 8, 2007).

[90] INDEX ENGINES ENTERPRISE SEARCH *http://www.indexengines.com/product_enterprise_search_appliance.htm*, (Accessed February 8, 2007).

[91] INTELLIMAGIC DISC MAGIC *http://www.intellimagic.nl*, (Accessed February 8, 2007).

[92] INTERNATIONAL BUSINESS MACHINES. *http://www.ibm.com*, (Accessed February 8, 2007).

[93] IRON MOUNTAIN. *http://www.ironmountain.com*, (Accessed February 8, 2007).

[94] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., and FU, K., "Plutus: Scalable secure file sharing on untrusted storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[95] KATCHER, J., "PostMark: A New File System Benchmark," *Network Appliance Technical Report TR3022*, 1997.

[96] KEETON, K., "Designing for disasters," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2004.

[97] KEETON, K. and MERCHANT, A., "A Framework for Evaluating Storage System Dependability," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN'04)*, 2004.

[98] KEPHART, J. O. and CHESS, D. M., "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[99] KERNS, R., "Storage service provider revival?," *Computerworld*, March 2005.

[100] KHULLER, S., KIM, Y., and WAN, Y., "Algorithms for data migration with cloning," in *Proceedings of ACM Symposium on Principles of Database Systems*, pp. 27–36, 2003.

[101] KILTZ, E., MITYAGIN, A., PANJWANI, S., and RAGHAVAN, B., "Append-Only Signatures," in *International Colloquium on Automata, Languages and Programming (ICALP)*, pp. 434–445, 2005.

[102] KLEIN, P., AGRAWAL, A., RAVI, R., and RAO, S., "Approximation through multicommodity flow," in *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 726–737, 1990.

[103] KRAWCZYK, H., BELLARE, M., and CANETTI, R., "HMAC: Keyed-hashing for message authentication," *http://www.faqs.org/rfcs/rfc2104.html*, (Accessed February 8, 2007).

[104] KRETSER, O., MOFFAT, A., SHIMMIN, T., and ZOBEL, J., "Methodologies for distributed information retrieval," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 1998.

[105] LEIGHTON, T. and MICALI, S., "Secret-key agreement without public-key," in *Proceedings of the International Conference on Advances in Cryptology (CRYPTO)*, pp. 456–479, 1994.

[106] Li, J., Krohn, M., and Mazieres, D., "Secure untrusted data repository SUNDR," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[107] Linux Manual Pages. *man command-name.*

[108] Linux Manual Pages (5) *man 5 attr.*

[109] Liu, J., Huang, W., Abali, B., and Panda, D., "High performance vmm-bypass i/o in virtual machines," in *Proceedings of the USENIX Annual Technical Conference*, 2006.

[110] Lu, C., Alvarez, G., and Wilkes, J., "Aqueduct: Online Data Migration with Performance Guarantees," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[111] Lymberopoulos, L., Lupu, E., and Sloman, M., "An Adaptive Policy-based Framework for Network Services Management," *Journal of Networks and System Management*, vol. 11, no. 3, 2003.

[112] McCallum, A., "Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering," *http://www.cs.cmu.edu/~mccallum/bow*, (Accessed February 8, 2007).

[113] McVitie, D. G. and Wilson, L. B., "The stable marriage problem," *Communications of the ACM*, vol. 14, pp. 486–490, July 1971.

[114] Melancon, D., "Keep control of your system," *http://www.computeruser.com/articles/2310,5,88,1,1001,04.html*, 2004.

[115] Menezes, A., van Oorschot, P. C., and Vanstone, S., "Handbook of applied cryptography," *CRC Press*, 1996.

[116] Menon, A., Cox, A., and Zwaenepoel, W., "Optimizing network virtualization in xen," in *Proceedings of the USENIX Annual Technical Conference*, 2006.

[117] Menon, J., Pease, D. A., Rees, R., Duyanovich, L., and Hillsberg, B., "IBM Storage Tank– A heterogeneous scalable SAN file system," *IBM Systems Journal*, vol. 42, no. 2, pp. 250–267, 2003.

[118] Microsoft Hotmail *http://www.hotmail.com*, (Accessed February 8, 2007).

[119] Miller, E., Long, D., Freeman, W., and Reed, B., "Strong Security for Distributed File Systems," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[120] Mitnick, K., Simon, W., and Wozniak, S., *The Art of Deception: Controlling the Human Element of Security.* John Wiley and Sons, 2002.

[121] Molero, X., Silla, F., Santonja, V., and Duato, J., "Modeling and Simulation of Storage Area Networks," in *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2000.

[122] Mozilla *http://www.mozilla.org*, (Accessed February 8, 2007).

[123] Mozilla Bug Report, "Bug 59557," *https://bugzilla.mozilla.org/show_bug.cgi?id=59557*, (Accessed February 8, 2007).

[124] Mozilla Contents *http://www.holgermetzger.de/pdl.html*, (Accessed February 8, 2007).

[125] MSN Toolbar *http://toolbar.msn.com*, (Accessed February 8, 2007).

[126] Naor, D., Shenhav, A., and Wool, A., "Toward securing untrusted storage without public-key operations," in *Proceedings of the ACM workshop on Storage security and survivability (StorageSS)*, pp. 51–56, 2005.

[127] National Insitute of Standards and Technology (NIST) *http://www.nist.gov*, (Accessed February 8, 2007).

[128] National Security Agency *http://www.nsa.gov*, (Accessed February 8, 2007).

[129] Nelson, M., Lim, B., and Hutchins, G., "Fast Transparent Migration for Virtual Machines," in *Proceedings of the USENIX Annual Technical Conference*, pp. 383–386, 2005.

[130] NEOS Server: MINOS Solver *http://neos.mcs.anl.gov/neos/solvers/nco:MINOS/AMPL.html*, (Accessed February 8, 2007).

[131] Ng, W. T., Sun, H., Hillyer, B., Shriver, E., Gabber, E., and Ozden, B., "Obtaining high performance for storage outsourcing," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, p. 11, 2002.

[132] NIST, "Cryptographic Algorithms and Key Sizes for Personal Identity Verification," *NIST Special Publication 800-78*, 2005.

[133] NIST, "AES: Advanced Encryption Standard," *http://csrc.nist.gov/CryptoToolkit/aes*, (Accessed February 8, 2007).

[134] NIST, "DES: Data Encryption Standard," *http://www.itl.nist.gov/fipspubs/fip46-2.htm*, (Accessed February 8, 2007).

[135] Okamoto, T., Fujisaki, E., and Morita, H., "TSH-ESIGN: Efficient Digital Signature Scheme Using Trisection Size Hash," *IEEE P1363 Research Contributions*, 1998.

[136] Okamoto, T. and Stern, J., "Almost uniform density of power residues and the provable security of ESIGN," in *Proceedings of the International Conference on Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pp. 287–301, 2003.

[137] Opera *http://www.opera.com*, (Accessed February 8, 2007).

[138] Ott, A. and Fischer-Hubner, S., "The Rule Set Based Access Control (RSBAC) Framework for Linux," *Karlstad University Studies*, vol. 28, 2001.

[139] Packeteer. *http://www.packeteer.com*, (Accessed February 8, 2007).

[140] PanFS Parallel File System. *http://www.panasas.com/panfs.html*, (Accessed February 8, 2007).

[141] Parallel NFS. *http://www.pdl.cmu.edu/pNFS*, (Accessed February 8, 2007).

[142] Pine *http://www.washington.edu/pine/*, (Accessed February 8, 2007).

[143] Pisinger, D., "A minimal algorithm for the 0-1 knapsack problem," *Journal of Operations Research*, vol. 45, pp. 758–767, 1997.

[144] Pisinger, D., "Minknap Source Code," *http://www.diku.dk/~pisinger/codes.html*, (Accessed February 8, 2007).

[145] Planet Lab *http://www.planet-lab.org*, (Accessed April 2, 2007).

[146] Platespin, "Server consolidation and disaster recovery with virtual machines," *http://www.platespin.com/*, (Accessed February 8, 2007).

[147] Reiser4 Aliasing Debate *http://kerneltrap.org/node/3749*, (Accessed February 8, 2007).

[148] Ritchie, D. and Thompson, K., "The UNIX Time-Sharing System," *Communications of the ACM*, vol. 17, no. 7, 1974.

[149] Rivest, R., "The MD5 message-digest algorithm," *http://www.ietf.org/rfc/rfc1321.txt*, 1992.

[150] Rivest, R., Shamir, A., and Adleman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[151] Robertson, S., Walker, S., and Beaulieu, M., "Okapi at trec-7: Automatic ad hoc, filtering, vlc and interactive.," in *Proceedings of the Text Retrieval Conference (TREC)*, 1998.

[152] Rosenblum, M. and Garfinkel, T., "Virtual machine monitors: Current technology and future trends," *IEEE Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[153] Ruemmler, C. and Wilkes, J., "An Introduction to Disk Drive Modeling," *IEEE Computer*, vol. 27, no. 3, 1994.

[154] Salesforce.com . *http://www.salesforce.com*, (Accessed February 8, 2007).

[155] SAN Advisor, E. *http://www.emc.com*, (Accessed February 8, 2007).

[156] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C., "Role-based Access Control Models," *IEEE Computers*, vol. 29, no. 2, pp. 38–47, 1996.

[157] SANscreen, O. *http://www.onaro.com*, (Accessed February 8, 2007).

[158] Savelsbergh, M. W. P., "A branch-and-price algorithm for the generalized assignment problem," *Journal of Operations Research*, vol. 45, no. 6, pp. 831–841, 1997.

[159] Security-Enhanced Linux *http://www.nsa.gov/selinux*, (Accessed February 8, 2007).

[160] Sexton, M., *Broadband Networking: ATM, SDH and SONET.* Artech House Publishing, 2007.

[161] Shamir, A. and Tromer, E., "Factoring Large Numbers with the TWIRL Device," in *Proceedings of the International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pp. 1–26, 2003.

[162] Sheth, A. and Larson, A., "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys*, vol. 22, no. 3, pp. 180–236, 1990.

[163] Shmoys, D. and Tardos, É., "An approximation algorithm for the generalized assignment problem," *Journal of Mathematical Programming*, vol. 62, no. 1, pp. 461–474, 1993.

[164] Singh, A., Liu, L., and Ahamad, M., "Privacy analysis for data sharing in *nix systems," in *Proceedings of the USENIX Annual Technical Conference*, 2006.

[165] Singh, A., Srivatsa, M., and Liu, L., "Efficient and Secure Search of Enterprise File Sytems," *Georgia Tech CERCS Technical Report GIT-CERCS-07-07*, 2007.

[166] Sit, E. and Fu, K., "Web Cookies: Not Just a Privacy Risk," *Communications of the ACM*, vol. 44, no. 9, 2001.

[167] Slashdot, "Kevin Mitnick Interview," *http://interviews.slashdot.org/article.pl?sid=03/02/04/2233250&mode=nocomment&tid=103&tid=123&tid=172*, (Accessed February 8, 2007).

[168] SNIA Storage Management Initiative *http://www.snia.org*, (Accessed February 8, 2007).

[169] Song, D., Wagner, D., and Perrig, A., "Practical techniques for searches over encrypted data," in *Proceedings of the IEEE Security and Privacy Symposium*, 2000.

[170] Sugerman, J., Venkitachalam, G., and Lim, B., "Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor," in *Proceedings of the USENIX Annual Technical Conference*, pp. 1–14, 2001.

[171] SUN Grid. *http://www.sun.com/service/sungrid/index.jsp*, (Accessed February 8, 2007).

[172] Symantec Security Manager *http://enterprisesecurity.symantec.com/products/products.cfm?productid=%45*, (Accessed February 8, 2007).

[173] Szeredi, M., "FUSE: Filesystem in Userspace," *http://fuse.sourceforge.net*, (Accessed February 8, 2007).

[174] Thereska, E., Narayanan, D., and Ganger, G., "Towards self-predicting systems: What if you could ask "what-if"?," in *Proceedings of the Workshop on Self-adaptive and Autonomic Comp. Systems*, 2005.

[175] Tomlinson, W. J. and Lin, C., "Optical wavelength-division multiplexer for the 1-1.4-micron spectral region," *Electronic Letters*, vol. 14, p. 345, 1978.

[176] TREC Enterprise Track *http://www.ins.cwi.nl/projects/trec-ent*, (Accessed February 8, 2007).

[177] United States Secret Service and CERT Coordination Center, "2004 E-Crime Watch Survey," *http://www.cert.org/nav/allpubs.html*, (Accessed February 8, 2007).

[178] Varki, E., Merchant, A., Xu, J., and Qiu, X., "Issues and Challegenges in the Performance Analysis of Real Disk Arrays," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, 2004.

[179] Veritas, "Storage foundation management server," *http://www.symantec.com/enterprise/sfms/index.jsp*, (Accessed February 8, 2007).

[180] VIEW-OS, "View-OS: A Process with a View," *http://savannah.nongnu.org/projects/view-os*, (Accessed February 8, 2007).

[181] VMware, "Resource management with VMware DRS," *VMware Whitepaper*, 2006.

[182] VMware, "VMware Infrastructure 3 architecture," *Whitepaper*, 2006.

[183] VMware, "Open Virtualization Standards," *http://www.vmware.com/news/releases/community_source.html*, (Accessed February 8, 2007).

[184] VMware Inc. *http://www.vmware.com*, (Accessed February 8, 2007).

[185] VMware Server *www.vmware.com/products/server/*, (Accessed February 8, 2007).

[186] VMware Virtual Appliance *http://www.vmware.com/vmtn/appliances*, (Accessed February 8, 2007).

[187] Waldspurger, C., "Memory resource management in vmware esx server," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[188] Wang, C. and Spooner, D., "Access Control in a Heterogeneous Distributed Database Management System," in *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, 1987.

[189] Wang, M., Au, K., Ailamaki, A., Brockwell, A., Faloutsos, C., and Ganger, G., "Storage device performance prediction with CART models," *SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, 2004.

[190] Ward, J., Sullivan, M., Shahoumian, T., and Wilkes, J., "Appia: Automatic Storage Area Network Fabric Design," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[191] Ward, J., Sullivan, M., Shahoumian, T., Wilkes, J., Wu, R., and Beyer, D., "Appia and the HP SAN Designer: Automatic Storage Area Network Fabric Design," in *Proceedings of the HP Technical Conference*, 2003.

[192] Whitaker, A., Shaw, M., and Gribble, S., "Denali: Lightweight virtual machines for distributed and networked applications," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pp. 195–209, 2002.

[193] Wijnen, B., Presuhn, R., and McCloghrie, K., "View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)," *RFC 2275*, 1998.

[194] Wilkes, J., "The Pantheon Storage-System Simulator," *HP Labs Tech Report HPL-SSP-95-14*, 1995.

[195] Windows Desktop Search for Enterprise *http://www.microsoft.com/windows/desktopsearch*, (Accessed February 8, 2007).

[196] Witten, I., Moffat, A., and Bell, T. C., *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[197] Wright, C., Martino, M., and Zadok, E., "NCryptfs: A Secure and Convenient Cryptographic File System," in *Proceedings of the USENIX Annual Technical Conference*, 2003.

[198] XenEnterprise: Virtual data center *http://www.xensource.com/solutions/virtual_data_center.html*, (Accessed February 8, 2007).

[199] Yahoo Desktop *http://desktop.yahoo.com*, (Accessed February 8, 2007).

[200] Zadok, E., Badulescu, I., and Shender, A., "Cryptfs: A stackable vnode level encryption file system," *Tech Report CUCS-021-98, Computer Science, Columbia University*, 1998.

VITA

Aameek Singh was born and raised in the northern city of Amritsar in Punjab, India. He graduated with a Bachelors of Technology degree in Computer Science and Engineering from Indian Institute of Technology (IIT) Bombay in 2002. Upon graduation, Aameek joined the PhD program in the College of Computing at Georgia Institute of Technology, where he is affiliated with the Distributed Data Intensive Systems Laboratory (DISL) and the CERCS systems group. He has also collaborated with the Storage Systems group at IBM Almaden Research Center. Aameek's research interests are in the area of storage systems and distributed systems. His dissertation research has proposed new access control and storage management techniques for the enterprise storage-as-a-service paradigm. During his PhD, Aameek has also led various other projects in Peer-to-Peer systems, web caching and SIP/Voice-over-IP. Aameek has over twenty submitted and refereed publications in international journals and conferences and is a co-inventor on six patent applications. He is also a recipient of the IBM PhD fellowship for 2005-06 and 2006-07.