SHAROES: A Data Sharing Platform for Outsourced Enterprise Storage Environments

Aameek Singh [#1], Ling Liu [*2]

[#] *Storage Systems, IBM Almaden Research Center*
San Jose, CA, USA
[1] aameek.singh@us.ibm.com

[*] *College of Computing, Georgia Tech*
Altanta, GA, USA
[2] lingliu@cc.gatech.edu

***Abstract*— With fast paced growth of digital data and exploding storage management costs, enterprises are looking for new ways to effectively manage their data. One such cost-effective paradigm is the Storage-as-a-Service model, in which enterprises *outsource* their storage to a storage service provider (SSP) by storing data at a remote SSP-managed site and accessing it over a high speed network. Often for a variety of reasons, enterprises find it unacceptable to fully trust the SSP and prefer to store data in an encrypted form. This typically limits collaboration and data sharing among enterprise users due to complex key management and access control challenges.**

In this paper, we propose a platform called SHAROES that provides data sharing capability over such outsourced storage environments. SHAROES provide rich *nix-like data sharing semantics over SSP stored data, without trusting the SSP for data confidentiality or access control. SHAROES is unique in its ability in reducing user involvement during setup and operation through the use of in-band key management and allows a near-seamless transition of existing storage environments to the new model. It is also superior in performance by minimizing the use of expensive public-key cryptography in metadata management. We present the architecture and implementation of various SHAROES components and our experiments demonstrate performance superior to other proposals by over 40% on a number of benchmarks.

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 [1]. 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 and is estimated to be 75% of the total cost of ownership [2].

One paradigm finding success in alleviating storage management costs is that of *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 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. This frees the enterprise from expensive and expertise-intensive storage management while also providing on-demand storage that can grow or shrink according to their needs. SSPs also often provide better disaster recovery (DR) and content dissemination capabilities. Many SSPs are in market today, for example, Amazon S3 [3], SUN Grid [4][1].

It is important to distinguish this model from an Application Service Provider (ASP) model, where along with data, applications are also hosted at a service provider. That assumes complete trust of the service provider as it has access to plaintext data for running hosted applications. Many times this assumption of trust is unacceptable to enterprises who, for intellectual property and/or regulatory compliance reasons, are required to protect their data.

Figure 1 shows an example SSP-enabled infrastructure. The client enterprise and its users, distributed across multiple geographical locations, access data over the high-speed network from a remote SSP site[2]. Typically, existing data is transferred to the SSP site (the *transition* phase) and later updates and writes are handled through the network. The data is stored in an encrypted form at the SSP ensuring confidentiality. The SSP on its part provides a storage medium for the data and is responsible for managing that storage performing tasks like provisioning, Storage Area Network (SAN) design, zoning/LUNmasking, backups and DR.

However, storing data in an encrypted form often limits collaboration and data sharing among enterprise users. In the absence of a trusted access control enforcement engine in the data I/O path, enterprises are faced with the daunting task of key management to allow users access to only the data that they are authorized to access. In recent years, few

This work is partially sponsored by IBM PhD fellowship for the first author, and grants from NSF CSR, NSF CyberTrust, grant from AFOSR, and an IBM faculty award (2006-2007) for the second author

[1] It is worthy to point out that the storage-as-a-service model is currently in its second incarnation with a first wave during the dot-com era. Readers may refer to [5] for a discussion on the evolution of SSP model.

[2] There are other possible architectures for example, using centralized gateways for data access or metadata management, but this decentralized model is preferred due to its scalability and cryptographic savings [6]

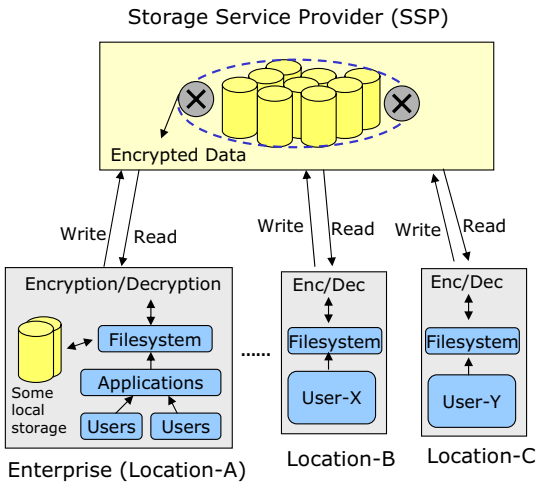

Fig. 1. Example SSP Infrastructure

solutions have been proposed like Plutus [7], Sirius [8] and SNAD [9]. However, these systems only provide a narrow subset of the data sharing semantics compared to those used in traditional systems, for example, the *nix data sharing model. This implies that the new systems would require extensive user involvement during the transition phase and also require re-education for using these systems. These complexities hinder data sharing and adversely impact the collaborative nature of an enterprise. Additionally, most system suffer performance issues due to greater use of expensive public key cryptography. We defer detailed discussion of related work to §VI.

In this paper, we describe a new system called SHAROES, that provides rich *nix-like data sharing semantics over SSP-stored data. To the best of our knowledge, SHAROES is a first such system. The SHAROES system is composed of three components: (1) migration tool – responsible for transitioning local storage to the outsourced model, (2) SSP data-serving tool – the SSP component for serving data from the remote site, and (3) a filesystem for local access to data served from the SSP. Using SHAROES, data stored in local storage can be seamlessly transitioned to the outsourced model with equivalant data sharing semantics without significant user involvement. The SHAROES filesystem manages key distribution completely in-band and users are required to only manage one public-private key pair. Further, by relying on symmetric key cryptography for metadata operations, SHAROES is much faster than other proposed systems. We describe the architecture of our prototype implementation and evaluate its performance on a number of micro and macro benchmarks. Our experiments demonstrate that the SHAROES filesystem consistently outperforms other proposed systems by 40-200%.

We start with a discussion of challenges involved in sharing data in the outsourced storage model.

A. Data Sharing Challenges

Providing sharing capabilities over SSP stored data can be very complex. Below, we discuss various challenges and give a glimpse of the features of our proposed approach:

- *Key Distribution*: Each enterprise user requires encryption keys for all files that he/she can access. *Distributing this large number of keys* to users in a decentralized manner is extremely challenging. Some related work in this area [7], [10] proposed using out-of-band channels like email for distribution. In contrast, SHAROES completely hides key management details from users.
- *Data Sharing Semantics*: Another important challenge is to provide *expressive data sharing semantics* like the ones used in typical local systems like the *nix model. Most of the related work [7], [8], [11], [12] provides only a restricted access control model with few permission settings. Typically, they only provide *read* and *write* permissions at a file level and hierarchical directory permissions are not supported, which studies suggest to be dominant in current local systems [13]. In contrast, the SHAROES system is able to provide rich data sharing semantics by carefully manipulating filesystem metadata and key distribution.
- *Performance*: In the storage-as-a-service model, each data and metadata access requires cryptographic operations. 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 [8], [11] relies on the expensive public key cryptography for *metadata* operations as it makes key distribution easier. In contrast, SHAROES predominantly uses symmetric key cryptography for metadata operations and outperforms comparable approaches by over 40% on a number of benchmarks.

II. BASIC CONCEPTS AND DATA STRUCTURES

Before we get into details of SHAROES, we describe some of the basic concepts and infrastructure requirements.

A. User and Group Keys

In SHAROES, 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 [14] 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). These encrypted group keys are stored at the SSP. When a user *alice* 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. Please refer to [6] for a detailed description of the group key distribution.

B. Encrypting Data and Metadata

In our work, we only consider enterprise data stored in *filesystems* (it comprises around 85% of all data - the rest is in databases [15]). Each file or directory in the filesystem

has a *data* and a *metadata* component. In SHAROES, all file and directory **data** is encrypted using distinct *symmetric* keys. Also, larger files are divided into multiple blocks and each block is encrypted separately. This helps accommodate updates efficiently by avoiding re-encrypting entire files after a `write`. For ease of exposition, we describe SHAROES assuming that all file data fits into a single block. Also, note that the data block for a directory consists of a table with information about its contents (inode numbers and names for Ext2 [16]).

Each data block has two sets of keys associated with it. The first, *Data Encryption Key (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. For example, 128-bit AES key.

The second set of keys is *Data Signing (DSK) and Data Verification (DVK)* keys. The DSK and DVK are a pair of asymmetric keys such that any content signed with the DSK, can only be verified 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, 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[3].

1) Metadata: Unlike existing approaches [8], [11], [9], in SHAROES, we use symmetric key cryptography for **metadata** objects. Thus, we have a similar set of keys – a *Metadata Encryption Key (MEK)*, *Metadata Signing Key (MSK)* (distributed only to object owners) and a *Metadata Verification Key (MVK)*.

Next, we describe the internal data structures for SHAROES filesystem. This design is key for expressive data sharing semantics and superior performance characteristics.

C. Key Data Structures

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

1) Metadata: 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 SHAROES, 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, as shown in Figure 2. 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. Additionally, 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. A detailed discussion about metadata design is in [6].

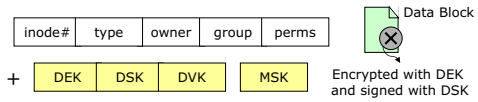

Fig. 2. SHAROES Metadata and Data

2) Directory Table: The data block for a directory contains information about the contents of the directory in a table. Our design extends Linux ext2 [16] directory table which consists of two columns containing (1) inode numbers and (2) names of the subfiles and subdirectories contained within that directory. For a user accessing a file in the directory, the inode number corresponding to its name is looked up and the metadata object for that inode number is obtained. From that metadata, the data block 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 3 shows the modified directory table structure.

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

Fig. 3. SHAROES Directory Table Structure

Note that fields containing keys in metadata and directory-table structures (new fields denoted by '+' in figures) are not always accessible to all users. In fact, their selective accessibility is what accomplishes access control and would eventually provide richness to the data sharing semantics. This is accomplished using a novel concept called Cryptographic Access control Primitive (CAP). In the next section, we describe CAPs for the *nix access control model.

III. CRYPTOGRAPHIC ACCESS CONTROL PRIMITIVES

A Cryptographic Access control Primitive (CAP) for a filesystem object tries to replicate an access control setting (for example, a read-only permission) in the outsourced storage

[3]Note that while public key schemes like RSA can be used for signing and verification, there are other techniques like ESIGN [17] that are over an order of magnitude faster [18]

model by manipulating accessibility of keys in metadata and directory-table structures and using cryptographic schemes when required. Below, we describe the design of CAPs for the *nix access control model. To support other models SHAROES only requires construction of similar CAPs [6].

A. *nix Directory CAPs

In the *nix access control model (which primarily follows the original UNIX model [19]), a directory can have three kinds of permissions – (a) `read`: allows listing the contents of the directory (that is, the command "`ls`"), (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 4 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.

For *nix, 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 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 *nix `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 *nix systems and our study at two large organizations showed that greater than 70% of users use `exec-only` permissions on directories [13]. To support this permission in SHAROES, the directory-table structure requires further manipulation using cryptographic primitives. We accomplish this as follows.

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 or SHA1 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 *nix permission not supported in SHAROES 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 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 *nix files.

B. *nix File CAPs

In the *nix 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 5 shows the design of CAPs for files. In case of files, data blocks are not used in the design and thus have been omitted from the figure.

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

[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.

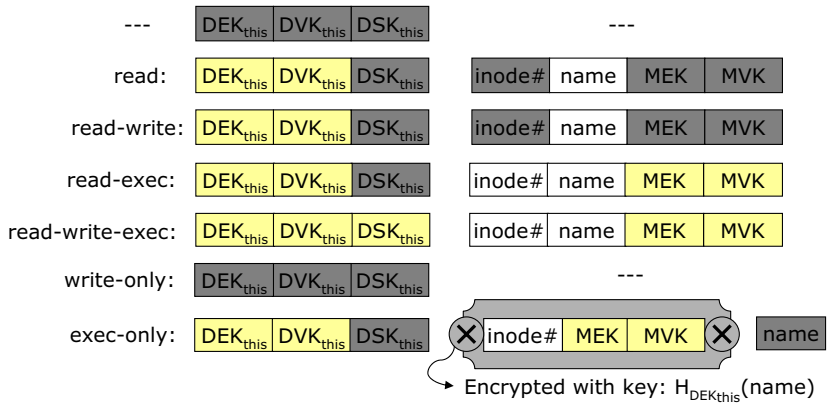

Fig. 4. *nix Directory CAPs

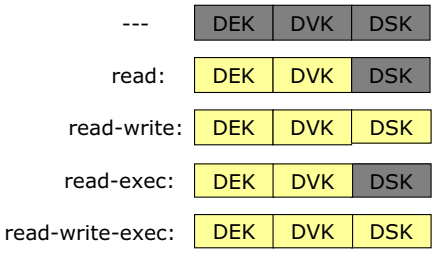

Fig. 5. *nix File CAPs

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 SSP-stored data. Please refer to Appendix-A for an integrated example of how file and directory CAPs interplay in a directory hierarchy.

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* [16]. We describe the superblock structure for SHAROES next.

C. 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 SHAROES, 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).

D. 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. We need to devise mechanisms such that each user gets access only to his/her CAP. 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.

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 [3] 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 [8], [18], [11] 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) Scheme-2: Sharing of CAPs: The second scheme avoids replicating metadata structures by observing that number of CAPs per object is typically much smaller than 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 and includes indirection from users' metadata and directory-table structures to point to the correct CAP for their individual permissions.

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 [20] when permissions for specific users or groups are added to the traditional *nix {*owner, group, others*} model. It is important to note that the total number of such splits is small, 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.

IV. SHAROES Architecture and Implementation

The SHAROES system is composed of three main components, as shown in Figure 6.

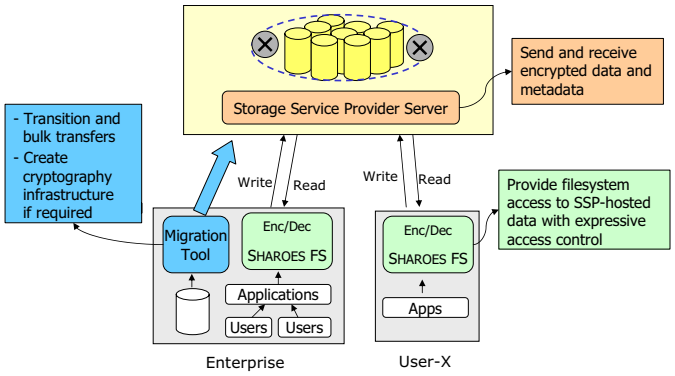

Fig. 6. SHAROES Architecture

- **Migration Tool**: This component is responsible for the initial setup and migration of data from local storage to the outsourced model. It can perform more efficient bulk data transfers (by using compression and other optimizations) and create the cryptographic infrastructure, if required (that is, generating user and group keys).
- **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).
- SHAROES **Filesystem**: The most important component in the architecture is the SHAROES 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 filesystem architecture next.

A. SHAROES *Filesystem*

The SHAROES filesystem provides filesystem-like access over remotely stored SSP data. It is this component that is responsible for navigating through the cryptographic CAPs based design and encryption/decryption of metadata and data blocks. Figure 7 shows the architecture of our prototype filesystem.

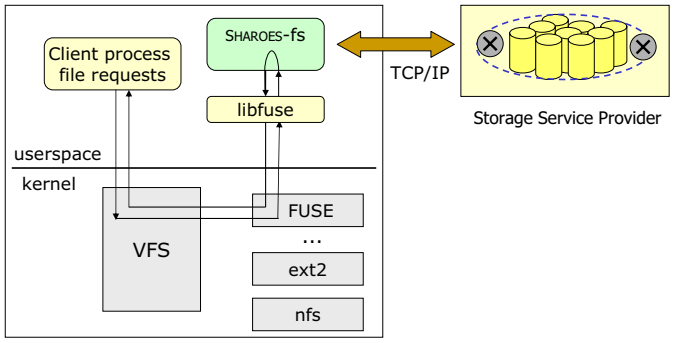

Fig. 7. Architecture of SHAROES Filesystem

We developed the SHAROES filesystem in userspace using the FUSE [21] 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 SHAROES filesystem is based. Also, we use TCP/IP sockets for the 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 commands below.

1) Filesystem Operations Implementation: All POSIX compliant filesystems implement a number of operations like `getattr`, `mkdir`, `read`, `close`. For the SHAROES filesystem, these functions are required to handle communication

with the SSP over the network and perform all encryption and decryption operations. Figure 8 shows how some of these functions are implemented. The last column also shows different components of cost involved.

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

Fig. 8. SHAROES Filesystem Operations

First, the **getattr** function is executed in response to a `stat` request. This function is required to return various attributes like owner, group and permissions for the filesystem object being `stat`'ed. In SHAROES, 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 SHAROES, 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 §III-D). The **mknod** function is similar except that a new *file* is created.

The **chmod** function is used to change permissions for a file or directory. For the SHAROES 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[5]. In related research on this topic, systems support one of the two revocation schemes – (a) immediate revocation [8], in which case a new key is created and file re-encrypted immediately during the `chmod` operation, or (b) lazy revocation [7], 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 SHAROES 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**.

[5]Revocation is also required when group memberships are changed.

V. EXPERIMENTAL EVALUATION

In this section, we perform a detailed evaluation of SHAROES comparing it to other related proposals. The core strengths of SHAROES design lie in its ability to provide an expressive *nix-like access control model, using symmetric key cryptography for metadata operations and complete in-band management of keys. We compared the SHAROES 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.
2) **NO-ENC-MD**: This implementation does not encrypt metadata but encrypts data with symmetric keys.
3) **PUBLIC**: This implementation encrypts data using symmetric keys and metadata objects with public key cryptography. This is representative of most other access control proposals for storage-as-a-service [8], [11], [9].
4) **PUB-OPT**: In PUB-OPT, the metadata object is encrypted with a symmetric key and then that key is encrypted with public key cryptography. This provides better performance than the PUBLIC implementation.

A. Setup

For SHAROES evaluation, we set up the SSP in Georgia Tech, Atlanta, GA, USA. The server is a shared SunOS server with four 1GHz processors and 8GB RAM. The client was set up in Birmingham, AL, USA, which is nearly 150 miles from the SSP. 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. For cryptographic operations, we used National Institute of Standards and Technology (NIST) approved standards used for protecting personal information of federal employees [22]. Specifically, we use 128-bit AES for symmetric key cryptography and 2048-bit RSA for public key cryptography. Finally, all experiments were repeated ten times and results were averaged.

1) Create-and-List Benchmark: To evaluate metadata performance, in our first benchmark, we measure the core costs of encryption and decryption of metadata objects in the outsourced storage model. Metadata is encrypted when new objects are created and decrypted when a `stat` is performed on a filesystem object (the `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 `stats` all files and directories. Figure 9 shows the results of this Create-And-List microbenchmark, with five implementations on the X-axis and time to create/list on Y-axis.

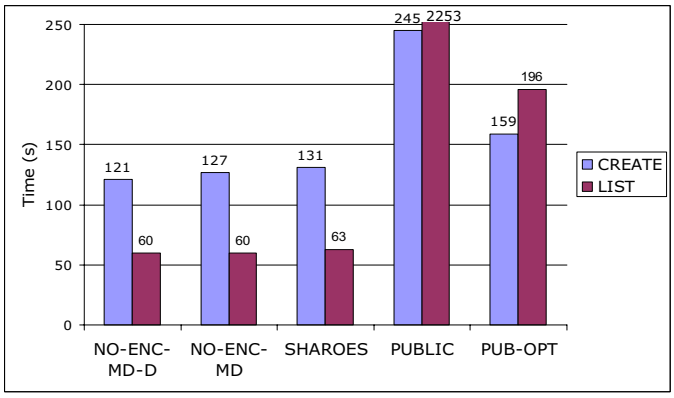

Fig. 9. Create-And-List Benchmark

First, we would like to point out the extremely poor performance of the PUBLIC implementation proposed in [8], [9], [11]. For creating 500 files, it took 245 seconds, almost twice as much as the NO-ENC and SHAROES 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. Operations with the private key are much more expensive and 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 9, 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, SHAROES 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 SHAROES 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.

B. Postmark Benchmark

Our second benchmark is the popular filesystem benchmark, called Postmark [23]. 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 10 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.

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 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 SHAROES. In contrast, SHAROES is always within 15% of the NO-ENC-MD-D approaches. This demonstrates superior performance of SHAROES for a metadata intensive workload.

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

C. Andrew Benchmark

The widely used Andrew Benchmark [24] 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) 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 11 plots the results for each individual phases and Figure 12 lists the cumulative performance for all five phases.

From Figure 11, Phase-2 and Phase-4 results show that I/O overheads for SHAROES 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

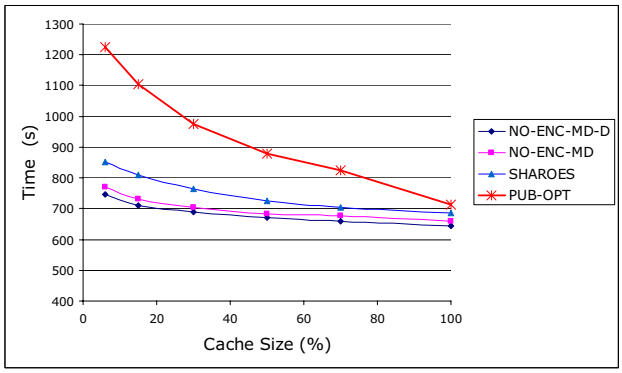

Fig. 10. Postmark Benchmark

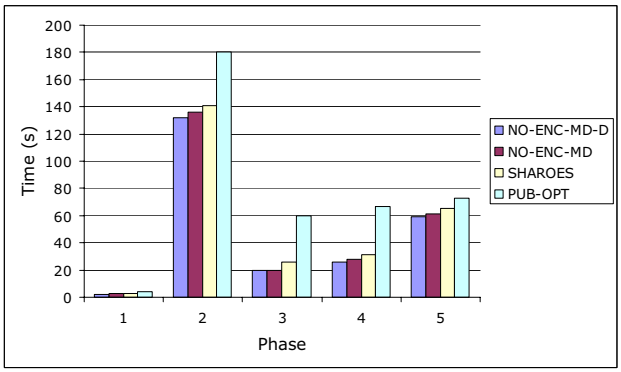

Fig. 11. SHAROES Andrew Benchmark Results

Scheme	Time (s)	Overheads
NO-ENC-MD-D	239	–
NO-ENC-MD	248	3.7%
SHAROES	266	11%
PUB-OPT	384	60%

Fig. 12. Cumulative performance for Andrew Benchmark

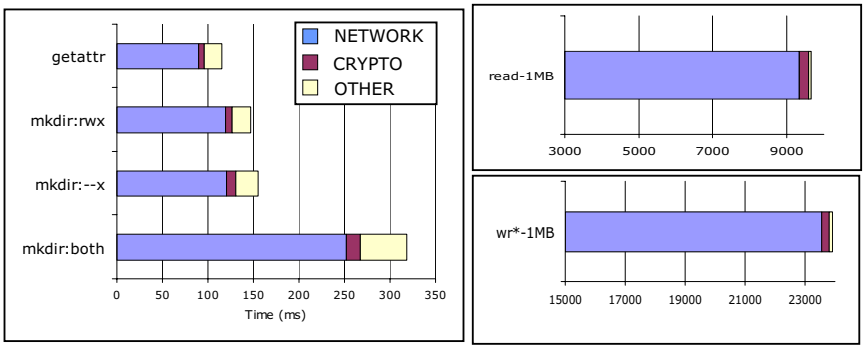

Fig. 13. Filesystem Operation Costs

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

D. Filesystem Operation Costs

We also analyzed the micro costs of various filesystem operations in SHAROES. We broke the costs into three components – (a) NETWORK: the network costs, (b) CRYPTO: cryptographic costs and (c) OTHER: all other costs. Figure 13 shows these costs for the `getattr`, `mkdir` (for different CAPs), and large file I/O (`read` and `write+close` of 1 MB files).

The `getattr` function obtains encrypted metadata from the SSP and decrypts it to access metadata attributes (see Figure 8). 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 (§III-A). 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 *nix directories).

We also evaluate SHAROES performance for large file I/O. As part of the experiment we read and wrote (+closed) 1 MB files. As the graph shows, SHAROES 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. Additional experimental analysis of SHAROES with varying network characteristics can be found in [6].

VI. Related Work

In recent years, a number of research efforts have proposed techniques for data sharing in the untrusted storage model. Plutus [7] describes a filesystem 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 [10]. In contrast, SHAROES provides an in-band key management technique that provides rich *nix-like data sharing semantics. It provides full support for hierarchical permissions and can seamlessly transition local storage to the outsourced model.

Another effort, Sirius [8] 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. This requires expensive private key based computation for every metadata read. Also, it does not support any directory-level permissions. Similar public key cryptography technique was used in Farsite [11] which provides a file system over untrusted P2P storage using Byzantine fault tolerance and access control authorization. In contrast, SHAROES predominantly uses symmetric key cryptography and provides complete *nix-like access control semantics. SNAD [9] also used a public keys for metadata and also trusts the SSP to perform certain verification operations.

Naor et al [12] propose a cryptographic primitive based on the Leighton-Micali [25] or Blom [26] schemes that can reduce public key cryptography 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 is that of SUNDR [18]. It describes the levels of consistency that can be supported in the untrusted storage-as-a-service model (*fork-consistency*). Their work is a complimentary contribution and we are currently integrating their consistency mechanisms with the SHAROES prototype.

VII. Discussion

The premise of the outsourced storage model is that a SSP is trusted to faithfully store/retrieve data for the client, but it is not trusted with either data confidentiality or access control. While it is possible that a malicious SSP can perform drastic attacks like erasing all data or Denial-of-Service, such attacks can be deterred through the use of Service Level Agreements (SLAs), in which various measurable parameters are negotiated (like throughput, data read and write success rates, latency) and penalties agreed upon in case SSP does not meet its objectives. We believe our work is important since it ensures that the confidentiality responsibilities lie primarily with enterprise users (keys never leave the enterprise domain in plaintext), similar to existing local systems. Also, any malicious attacks can be detected (through in-built verification processes and integrity techniques [18]) and thus penalized through a SLA.

VIII. Conclusions and Future Work

We have described SHAROES, a platform for data sharing in the storage-as-a-service model. SHAROES uses novel cryptographic access control primitives (CAPs) to support rich data sharing semantics without trusting the SSP for enforcement of security policies. We showed how SHAROES is able to support an expressive access control model, which in conjunction with its in-band key management technology provides a seamless transition ability from local storage to the outsourced model with minimal user involvement. Additionally, by primarily using symmetric key cryptography, SHAROES outperforms other systems by 40-200% on many benchmarks.

Our research on SHAROES continues along several dimensions. First, we plan to implement integrity mechanisms for SHAROES, leveraging some of the related work. Second, we are investigating the use of virtual machines to securely transmit execution contexts to support `setuid` operations.

References

[1] Gartner Group., *http://www.gartner.com*.

[2] F. Brick, "Are you ready to outsource your storage?" *Computer Technology Review*, June 2003.

[3] Amazon Storage Service., *http://aws.amazon.com/s3*.

[4] SUN Grid., *http://www.sun.com/service/sungrid/index.jsp*.

[5] R. Hasan, W. Yurcik, and S. Myagmar, "The evolution of storage service providers: techniques and challenges to outsourcing storage," in *StorageSS*, 2005, pp. 1–8.

[6] A. Singh, "Secure Management of Networked Storage Services: Models & Techniques," *PhD Thesis, Georgia Tech http://www.aameeksingh.com/thesis.pdf*, 2007.

[7] M. Kallahalla, E. Riedel, and et al, "Plutus: Scalable secure file sharing on untrusted storage," in *FAST*, 2003.

[8] E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: securing remote untrusted storage," in *NDSS*, 2003.

[9] E. Miller, D. Long, W. Freeman, and B. Reed, "Strong Security for Distributed File Systems," in *FAST*, 2002.

[10] A. Harrington and C. Jensen, "Cryptographic access control in a distributed file system," in *SACMAT*, 2003.

[11] A. Adya, W. Bolosky, and et al, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," in *OSDI*, 2002.

[12] D. Naor, A. Shenhav, and A. Wool, "Toward securing untrusted storage without public-key operations," in *StorageSS*, 2005.

[13] A. Singh, L. Liu, and M. Ahamad, "Privacy analysis for data sharing in *nix systems," in *USENIX Annual*, 2006.

[14] D. Boneh and M. Franklin, "Identity based encryption from the Weil pairing," *SIAM Journal of Computing*, vol. 32, no. 3, 2003.

[15] Bulter Group, "Unlocking value from text-based information," *Review Journal Article*, 2003.

[16] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," in *First Dutch International Symposium on Linux*, 1995.

[17] T. Okamoto, E. Fujisaki, and H. Morita, "TSH-ESIGN: Efficient Digital Signature Scheme Using Trisection Size Hash," *IEEE P1363 Research Contributions*, 1998.

[18] J. Li, M. Krohn, and D. Mazieres, "Secure untrusted data repository SUNDR," in *OSDI*, 2004.

[19] D. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, vol. 17, no. 7, 1974.

[20] A. Grunbacher and A. Nuremberg, "POSIX Access Control Lists on Linux," *http://www.suse.de/ agruen/acl/linux-acls/online*.

[21] M. Szeredi, "FUSE: Filesystem in Userspace."

[22] NIST, "Cryptographic Algorithms and Key Sizes for Personal Identity Verification," *NIST Special Publication 800-78*, 2005.

[23] J. Katcher, "PostMark: A New File System Benchmark," *Network Appliance Tech Report TR3022*, 1997.

[24] J. H. Howard, M. L. Kazar, and et al, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51–81, 1988.

[25] T. Leighton and S. Micali, "Secret-key agreement without public-key," in *CRYPTO*, 1994, pp. 456–479.

[26] R. Blom, "An optimal class of symmetric key generation systems," in *EUROCRYPT*, 1985.