Practical Share Renewal for Large Amounts of Data

Arun Subbiah and Douglas Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA

Technical Report: GIT-CERCS-05-04
February, 2005

Abstract

Threshold secret sharing schemes encode data into several shares such that a threshold number of shares can be used to recover the data. Such schemes provide confidentiality of stored data without using encryption, thus avoiding the problems associated with key management. To provide long-term confidentiality, proactive secret sharing techniques can be used, where shares are refreshed or renewed periodically so that an adversary who obtains fewer than the threshold shares in each time period does not learn any information on the encoded data.

Share renewal is an expensive process, in terms of the computation and network communication involved. In the proactive model, this share renewal process must complete as soon as possible so that an adversary who compromises servers in the present time period does not learn shares stored in the last time period. This paper proposes an algorithm where the shares of all the stored data are renewed by the share renewal of only one secret. The computation and network communication overheads are thus drastically reduced, allowing for the share renewal of all the stored data to complete quickly. These benefits are gained at the expense of some performance penalty during reads and writes, which is shown to be worthwhile.

Keywords: Distributed data storage, proactive secret sharing, confidentiality, share renewal, mobile adversary

1 Introduction

The problem of providing distributed data storage service has received much attention [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. The stored data can be kept confidential by using either encryption or secret sharing techniques. While symmetric key encryption algorithms are typically much faster than secret sharing techniques and suited for real-time access to data, they do have some drawbacks. Key-based algorithms suffer from the

problem of key management. A key management infrastructure is required, which should take care of changes in access control. If the confidential data is long-lived, then it cannot be guaranteed that encryption algorithms will remain secure for a long time. Changes in the key or the encryption algorithm will require a trusted authority to decrypt and re-encrypt all the stored data with a new key, which may not be practical. Hence, in the context of providing confidential data storage for data that is long-lived or archival in nature, encryption algorithms may not be suitable.

Secret sharing algorithms [16, 17], on the other hand, do not use cryptographic keys. The data is encoded into *shares* so that some qualified subsets of them can be used to recover the data. A (k, n) secret sharing scheme generates n shares of a secret S, such that the knowledge of any k shares is enough to reconstruct the secret S. Perfect secret sharing schemes, also known as threshold schemes, have the additional property that knowledge of fewer than k shares does not give any information on the encoded data. These properties are provided without requiring any encryption, thus avoiding the issues associated with key management. Though threshold schemes were traditionally used to store keys in a distributed set of servers, some works [18, 19, 12] have also considered storing generic data using threshold schemes.

If the shares are distributed to a group of servers, then the secret can be eventually recovered by an adversary by his compromising any k servers, assuming each server gets one share. So to provide long-term confidentiality, proactive secret sharing [20, 21, 22] techniques can be used. In proactive secret sharing, servers periodically engage in a distributed share renewal protocol where the shares of the secret are refreshed or renewed. The new shares of the secret are incompatible with the shares used to store the secret before the share renewal process. This prevents an adversary from gathering a consistent set of k shares of the secret over time.

Typical data storage systems store orders of magnitude of data objects. If each data object were to be stored using secret sharing, then share renewal of all the data objects will incur significant computation and network communication overheads. This severely limits the scalability of proactive secret sharing techniques for large volumes of data. Our proposed solution allows for the share renewal of all the secrets (or data objects) stored in the system by the share renewal of only one secret, thereby drastically reducing the computation and network communication overheads. The proposed algorithm is thus critical for keeping large volumes of data proactively confidential.

2 The Problem

Consider a system in which there are n servers, and a (potentially unlimited) number of clients which read and write secrets to the servers. To write a secret, a client generates n shares of the secret using a (k, n)-threshold scheme, and securely gives each server one share. To read a secret, a client contacts as many servers as necessary to retrieve k shares as a (k, n)-threshold scheme is being used for the secret sharing.

To keep the secrets proactively secure, the n servers have to periodically engage in a share renewal protocol to generate new shares of the secrets. When the number of secrets is

large, this process can take a long time, and cause heavy computation and communication overheads. How to reduce these overheads, which arise due to the share renewal of a *large* number of secrets, is the problem addressed in this paper.

The proposed solution involves maintaining a system secret, denoted by U, which is not known to any client. The servers generate a new sharing for U, and use these shares to renew the shares of all the stored secrets. Thus, only one share renewal protocol is run to renew the shares of all the clients' secrets. The communication overhead is thus due to *only one* run of the share renewal protocol, and the computation overhead on the servers is also reduced. Thus, the proposed solution can be used to achieve efficient proactive security.

3 Preliminaries

Let the secrets belong to Abelian group AG_1, and the shares belong to Abelian group AG_2. The group AG_1 is Abelian with respect to the binary operator $\oplus$, while the group AG_2 is Abelian with respect to the binary operator $\otimes$. Given k shares, $s_{1,1}, ..., s_{1,\mathrm{k}}$, of the secret S_1 that were generated using a k-threshold scheme, the secret S_1 can be reconstructed by computing $s_{1,1} \otimes s_{1,2} \otimes ... \otimes s_{1,\mathrm{k}}$. In $(\oplus, \otimes)$-homomorphic (k, n)-threshold schemes [23], if

$$\text{shares of } S_1 \text{ are } (s_{1,1}, s_{1,2}, ..., s_{1,\mathrm{n}}), \text{ and}$$
$$\text{shares of } S_2 \text{ are } (s_{2,1}, s_{2,2}, ..., s_{2,\mathrm{n}}),$$
$$\text{then shares of } (S_1 \oplus S_2) \text{ are } ((s_{1,1} \otimes s_{2,1}), (s_{1,2} \otimes s_{2,2}), ..., (s_{1,\mathrm{n}} \otimes s_{2,\mathrm{n}})).$$

In this paper, we consider $(\oplus, \otimes)$-homomorphic (k, n)-threshold schemes. An example of a $(\oplus, \otimes)$-homomorphic (k, n)-threshold scheme is Shamir's scheme [16], which is a $(+, +)$-homomorphic (k, n)-threshold scheme.

Since AG_1 and AG_2 are Abelian groups, there exists an identity element, denoted by $0_1 \in AG_1$ and $0_2 \in AG_2$ respectively, for each of the two groups. That is, for an element $S_1 \in AG_1$, $S_1 \oplus 0_1 = S_1$, and for an element $s_{1,1} \in AG_2$, $s_{1,1} \otimes 0_2 = s_{1,1}$.

In each of the two Abelian groups, there also exist an inverse for every element in the group. For example, for an element $S_1 \in AG_1$, there exists a unique inverse denoted by $S_1^{-1} \in AG_1$ such that $S_1 \oplus S_1^{-1} = 0_1$. Likewise, for an element $s_{1,1} \in AG_2$, there exists a unique inverse denoted by $s_{1,1}^{-1} \in AG_2$ such that $s_{1,1} \otimes s_{1,1}^{-1} = 0_2$.

There are times in the text where a computation such as $s \otimes s \otimes ... \otimes s$ (R_S times) or $S \oplus S \oplus ... \oplus S$ (R_S times) may need to be performed. This is written abbreviated as $R_\mathrm{S}s$ or $R_\mathrm{S}S$ respectively.

4 Protocol Description

The protocol in the presence of passive adversaries, i.e., adversaries interested only in recovering data and not in corrupting them, is described in this section.

4.1 Initialization and Maintenance of the System Secret

When the system is first initialized, the servers come up with a secret they will maintain using a $(\oplus, \otimes)$-homomorphic (k, n) threshold scheme. The share renewal of this *system secret*, denoted by U, allows the renewal of shares of all the data objects stored at the servers. The system secret U must be generated such that no subset of $(k-1)$ servers can collectively find U. Also, the proposed algorithm requires generation and maintenance of the inverse of the system secret, denoted by U^{-1}.

At start, the identity element of AG_1, 0_1, is secret shared using an $(\oplus, \otimes)$-homomorphic (n, n) threshold scheme by a system administrator, and each server is given one share, i.e., server i receives share $0_{1,\mathrm{i}}$. This secret sharing need not be secret and can be done publicly. To set up the unknown system secret U, each server comes up with a random number $u_{\mathrm{i}} \in AG_2$ secretly. The n shares are now understood to represent the unknown system secret U using an $(\oplus, \otimes)$-homomorphic (n, n) threshold scheme. Each server i, from its share of U and 0_1, can find its share of U^{-1} by computing $u_{\mathrm{i}}^{(\mathrm{I})} = 0_{1,\mathrm{i}} \otimes (u_{\mathrm{i}})^{-1}$. The correctness of this sharing can be seen below:

$$
\begin{gathered}
U \oplus U^{-1} = 0_1 \\
\Rightarrow (u_1 \otimes u_1^{(\mathrm{I})}, u_2 \otimes u_2^{(\mathrm{I})}, ..., u_{\mathrm{n}} \otimes u_{\mathrm{n}}^{(\mathrm{I})}) = (0_{1,1}, 0_{1,2}, ..., 0_{1,\mathrm{n}}) \\
\text{i.e., } u_{\mathrm{i}} \otimes u_{\mathrm{i}}^{(\mathrm{I})} = 0_{1,\mathrm{i}},\ 1 \le i \le n \\
\Rightarrow u_{\mathrm{i}}^{(\mathrm{I})} = 0_{1,\mathrm{i}} \otimes u_{\mathrm{i}}^{-1},\ 1 \le i \le n
\end{gathered}
$$

Servers then redistribute the shares of U and U^{-1} from an (n, n)-threshold scheme to a (k, n)-threshold scheme using a protocol such as Desmedt and Jajodia's [24]. The shares of 0_1 can be deleted after this secret redistribution. Successful deletion of the shares of 0_1 is not required for the security of the system. However, reliable deletion of the shares of U and U^{-1} corresponding to the (n, n)-threshold scheme is required to maintain proactive security.

Let the n shares of U^{-1} distributed using the (k, n)-threshold scheme be denoted by $(u_{-\mathrm{a},1}, u_{-\mathrm{a},2}, ..., u_{-\mathrm{a},\mathrm{n}})$. Successive share renewals of U^{-1} will produce shares denoted by $u_{-\mathrm{b},\mathrm{i}}$'s, $u_{-\mathrm{c},\mathrm{i}}$'s, and so on. Let the n shares of U distributed using the (k, n)-threshold scheme be denoted by $(u_{1,1}, u_{1,2}, ..., u_{1,\mathrm{n}})$. Successive share renewals of U will produce shares denoted by $u_{2,\mathrm{i}}$'s, $u_{3,\mathrm{i}}$'s, and so on. The shares of U and U^{-1} are renewed periodically to preserve the confidentiality of the system secret U. New sharings of U are also required in the proposed share renewal algorithm.

4.2 Data Write Protocol

This section describes the data write protocol followed by the servers and the users or clients when a secret is written. These secrets are called *user secrets*, as opposed to the *system secret* U.

To write a user secret S_1, a client generates n shares of S_1 using the $(\oplus, \otimes)$-homomorphic (k, n) threshold scheme, and gives each server securely its share. Let the shares of S_1 thus generated be denoted by $s_{1,1}$, $s_{1,2}$,...,$s_{1,\mathrm{n}}$. Upon receiving a write request, servers engage in a

share renewal protocol on U to generate a new sharing of U. For the write to secret S_1, let the sharing of U generated be denoted by $(s_{2,1}, s_{2,2}, ..., s_{2,\mathrm{n}})$. Each server "adds" ($\otimes$) its new share of U to its share of S_1, and permanently deletes the new share of U. Note that this share renewal of U is not for keeping U proactively secure. The new sharing of U is added to the received shares in the write request and the new sharing of U is permanently deleted. Thus, the secret S_1 is stored at the servers as $S_1 \oplus U$ after a write.

The share renewal of U for the purpose of writes need not be performed as and when writes are received. They can be run in anticipation of new writes, but all such sharings must be deleted before the servers transition into the next time period. In networks where there is no broadcast channel, an agreement protocol needs to be run amongst the servers during writes, and a share renewal on U could be accomplished as part of the agreement process.

For each data object or secret, a replicated variable is maintained by the servers. When the secret S_1 is written, servers create a variable R_{S_1} and initialize it to 1.

4.3 The Data Share Renewal Protocol

The algorithm works by altering the secrets when share renewal is done, but by a known quantity which is the system secret U.

At the time period boundaries, servers engage in a share renewal protocol on system secret U to generate a new set of shares of U that will be used to refresh the shares of *all* the user secrets stored in the system. Denote one such sharing of U generated for this purpose by $u_{2,1}, u_{2,2}, .., u_{2,\mathrm{n}}$.

Servers "add"($\otimes$) these shares of U to the shares of *all* the encoded data objects to renew the shares. That is, server i "adds" ($\otimes$) $u_{2\mathrm{i}}$ to the current share it has for *all* the data objects stored, and then reliably deletes $u_{2\mathrm{i}}$. Thus, the shares of user secret S_1 now stored at the servers correspond to the secret $S_1 \oplus U \oplus U = S_1 \oplus 2U$. The replicated variable R_{S_1} is incremented by one. In general, the shares of S_1 currently stored at the servers correspond to the secret $S_1 \oplus R_{\mathrm{S}_1}U$, where $R_{\mathrm{S}_1} - 1$ share renewals on secret S_1 have taken place since S_1 was last written.

4.4 Data Read Protocol

When a client requests user secret S_1, servers must return shares of S_1. To accomplish this, servers add R_{S_1} times u_{mi} ($u_{-\mathrm{a,i}} \otimes u_{-\mathrm{a,i}} \otimes ... \otimes u_{-\mathrm{a,i}}$ (R_{S_1} times)) to the share it has currently for S_{i} and returns the sum ($\otimes$). The shares received are thus that of secret S_1, because the resulting secret read will be

$$(S_1 \oplus R_{\mathrm{S}_1}U) \oplus (R_{\mathrm{S}_1}U^{-1}) = S_1 \oplus (R_{\mathrm{S}_1}U \oplus R_{\mathrm{S}_1}U^{-1}) = S_1 \oplus 0_1 = S_1$$

5 Security Analysis

We first present an informal analysis of the security of the share renewal algorithm.

We first show why U must be kept secret. Consider a client who has read and write access to user secret S_2, but has no such access to another user secret S_1. The client knows the system secret U and is interested in finding out S_1. When secret S_2 is written to the servers, the client is aware of the shares generated to represent S_2. He then compromises $k-1$ servers, and finds the $u_{-\mathrm{a,i}}$'s used to store U^{-1}. Since he knows U and U^{-1}, from these $k-1$ shares he can figure out the $u_{-\mathrm{a,i}}$'s stored at other servers. The client does a read on secret S_2, and since he knows the $u_{-\mathrm{a,i}}$'s, he can figure out the shares of $S_2 \oplus U$ stored at the servers. From this, the client can figure out the shares of U, say $u_{2,\mathrm{i}}$'s, used at the time S_2 was written. The client also learns of $k-1$ shares of secret $S_1 \oplus U$ stored at the $k-1$ compromised servers.

At the next time period boundary, these $k-1$ compromised servers have been corrected, and share renewal takes place. In the subsequent time period, the client compromises a different set of $k-1$ servers. From the new shares of U^{-1} stored at these servers, he can figure out all the new u_{mi}'s. He now does a read on S_2. Server i will return $s_{2,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes u_{3,\mathrm{i}} \otimes 2u_{-\mathrm{a,i}}$, where the $u_{3,\mathrm{i}}$'s are the shares of U "added" ($\otimes$) to the shares of *all* the user secrets stored in the system during the share renewal phase. The client is aware of all the quantities in the above expression except the $u_{3,\mathrm{i}}$'s, the shares of U added to the shares of all the user secrets at the time of share renewal. The client can hence find out the $u_{3,\mathrm{i}}$'s.

From one of the compromised servers, say the i^{th} server, the client can find out the new share for secret S_1 after the share renewal, which will be $s_{1,\mathrm{i}} \otimes u_{4,\mathrm{i}} \otimes u_{3,\mathrm{i}}$. Since he knows $u_{3,\mathrm{i}}$, he can find out the "old" share for S_1, which is $s_{1,\mathrm{i}} \otimes u_{4,\mathrm{i}}$. The client already has $k-1$ "old" shares of secret S_1, or rather that of $S_1 \oplus U$, and with this additional share he can recover $S_1 \oplus U$. Since the client knows U, he can find out S_1.

Hence, U must be kept secret.

We next explain why when a secret is written by a client, its shares are not stored as they are, and instead modified to store the secret $\oplus\, U$. Assume that when secret S_2 is written, its shares are written unmodified. So when a client does a read on S_2 before any share renewal has taken place, he gets the shares he originally generated for S_2. Secret S_1, that the client is not allowed to access, is also stored as the original shares generated by its owner. The client compromises $k-1$ servers, and finds out $k-1$ shares of secret S_1.

At the next time period boundary, share renewal takes place for all the user secrets as well as for the system secret U and U^{-1}. In the following time period, the client compromises another set of $k-1$ servers. The shares of secret S_2 stored at these servers are, say, $s_{2,\mathrm{i}} \otimes u_{2,\mathrm{i}}$'s. Since the client is aware of the $s_{2,\mathrm{i}}$'s, he can find out $k-1$ $u_{2,\mathrm{i}}$'s. The shares of secret S_1 stored at these compromised servers are $s_{1,\mathrm{i}} \otimes u_{2,\mathrm{i}}$. Since the client is aware of the $u_{2,\mathrm{i}}$'s stored at these compromised servers, he can find out an additional $k-1$ shares of secret S_1 (the $s_{1,\mathrm{i}}$'s). The client has already obtained upto $k-1$ shares of secret S_1 (that belong to the set of $s_{1,\mathrm{i}}$'s) from the earlier time period. Since he now has more than $k-1$ shares of S_1 (the

$s_{1,\mathrm{i}}$'s), he can find out secret S_1.

To avoid this scenario, at the time a user secret is written, the shares are modified to store the user secret $\oplus$ U. Also, the shares of U used at the time of a user-secret write is never used for other writes so that there exists no correlation between the stored shares of any two user secrets. Since the set of shares of U used at the time of writes to user secrets will never be used again, it must be deleted permanently from its local storage. This deletion must be done before the next immediate time period boundary.

To prove the security formally, we assume the existence of a client C who has read and write access to user secrets $S_2, S_3, ...$, and is interested in finding out user secret S_1 or the system secret U. The client C is hence also an adversary.

Theorem 1 *An adversary C, who has complete read and writes access over some user secrets, cannot recover the system secret U.*

Proof: Since a (k, n) threshold scheme is used throughout, an adversary can recover U if he is able to find at least k consistent shares of U, while $k-1$ shares of U will reveal no information about U.

The shares used to store U^{-1}, the $u_{-\mathrm{a,i}}$'s, are protected by periodic share renewal. Hence, an adversary cannot learn any information about U from these set of shares.

Sets of shares of U are created during a time period for data object writes, and are either used up during writes or are permanently deleted at the end of a time period. Hence, the same sets of shares of U are unavailable in plain text at the servers in more than one time period. Since only a maximum of $k-1$ servers can be in the compromised state in a time period, and the sets of shares of U are generated using a k-threshold scheme, an adversary cannot learn U from these sets of shares.

Let the client / adversary C have full read and write access to user secret S_2, and be interested in finding out system secret U. When user secret S_2 is written by client C, let the shares of S_2 generated using a (k, n)- threshold scheme be denoted by $s_{2,\mathrm{i}}$'s. To store S_2 as $S_2 \oplus U$, let the servers use the shares of U denoted by $u_{2,\mathrm{i}}$'s. The shares of S_2 are hence stored at the servers as $s_{2,\mathrm{i}} \otimes u_{2,\mathrm{i}}$'s. Let the current set of shares used to store U^{-1} at the servers be denoted by $u_{-\mathrm{a,i}}$'s. When C does a read on S_2 from all the servers, it obtains shares $s_{2,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes u_{-\mathrm{a,i}}$'s. Since C is aware of the $s_{2,\mathrm{i}}$'s, he can find out $u_{2,\mathrm{i}} \otimes u_{-\mathrm{a,i}}$ for $i = 1, 2, ..., n$. These shares correspond to the secret $U \oplus U^{-1} = 0_1$, the identity element in Abelian group AG_1, and as such are of no use in finding out U. Without loss of generality, assume that C compromises the first $k-1$ servers in this time period. C thus obtains the first $k-1$ shares of $u_{2,\mathrm{i}}$ and the first $k-1$ shares of $u_{-\mathrm{a,i}}$. The $u_{2,\mathrm{i}}$'s and the $u_{-\mathrm{a,i}}$'s are two different sets of shares. With only $k-1$ shares from each set, an adversary cannot recover U as a k-threshold scheme was used. To prove that even the knowledge of the n $(u_{2,\mathrm{i}} \otimes u_{-\mathrm{a,i}})$'s will not help in finding U, consider the following example. Fix a certain value for U, and from $u_{2,1}, ..., u_{2,\mathrm{k}-1}$ and $u_{-\mathrm{a},1}, ..., u_{-\mathrm{a,k}-1}$, the rest of the shares $u_{2,\mathrm{k}}, ..., u_{2,\mathrm{n}}$ and $u_{-\mathrm{a,k}}, ..., u_{-\mathrm{a,n}}$ can be fixed. The adversary C is aware of $u_{2,\mathrm{i}} \otimes u_{-\mathrm{a,i}}, 1 \leq i \leq n$, out of which the shares $u_{2,1}, ..., u_{2,\mathrm{k}-1}$ and $u_{-\mathrm{a},1}, ..., u_{-\mathrm{a,k}-1}$ were used to calculate the rest of the shares

$u_{2,\mathrm{i}}$'s and $u_{-\mathrm{a},\mathrm{i}}$'s. The secret that will be recovered from $(u_{2,\mathrm{i}} \otimes u_{-\mathrm{a},\mathrm{i}})$'s using the calculated $u_{2,\mathrm{i}}$'s and $u_{-\mathrm{a},\mathrm{i}}$'s will correspond to the secret $U \oplus U^{-1} = 0$, the identity element which is independent of U. Out of the read $(u_{2,\mathrm{i}} \otimes u_{-\mathrm{a},\mathrm{i}})$'s and the calculated $(u_{2,\mathrm{i}} \otimes u_{-\mathrm{a},\mathrm{i}})$'s, $k-1$ of the $(u_{2,\mathrm{i}} \otimes u_{-\mathrm{a},\mathrm{i}})$'s are identical and both correspond to the secret "0_1." Therefore, since a k-threshold scheme was used, the rest of the share sets (the read and the computed) must also be identical.

Hence, the knowledge of $k-1$ $u_{2,\mathrm{i}}$'s and $k-1$ $u_{-\mathrm{a},\mathrm{i}}$'s, and all the $(u_{2,\mathrm{i}} \otimes u_{-\mathrm{a},\mathrm{i}})$'s, will not reveal any information on the system secret U.

In the subsequent time period, assume the set of shares of U denoted by $u_{2,\mathrm{i}}$'s were generated and "added" ($\otimes$) to the shares of all the user secrets in order to renew their shares. The shares of S_2 stored at the servers are hence $s_{2,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes u_{3,\mathrm{i}}$. Also, the shares used to store U, the $u_{-\mathrm{a},\mathrm{i}}$'s, were replaced with $u_{-\mathrm{b},\mathrm{i}}$'s.

When client C does a read on S_2 from all the servers, he gets $(s_{2,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes u_{3,\mathrm{i}} \otimes 2u_{-\mathrm{b},\mathrm{i}})$'s, $i = 1, 2, ..., n$. C can thus obtain $(u_{2,\mathrm{i}} \otimes u_{3,\mathrm{i}} \otimes 2u_{-\mathrm{b},\mathrm{i}})$'s, $i = 1, 2, ..., n$. These shares correspond again to the secret *zero* and as such are of no use in finding U. Even the knowledge of $(u_{2,\mathrm{i}} \otimes u_{-\mathrm{a},\mathrm{i}})$, $i = 1, 2, ..., n$, from the last time period does not help as the $u_{-\mathrm{a},\mathrm{i}}$'s have been replaced with $u_{-\mathrm{b},\mathrm{i}}$'s.

By compromising $k-1$ servers in this time period, C can obtain $k-1$ $(u_{2,\mathrm{i}} \otimes u_{3,\mathrm{i}})$'s and $k-1$ $u_{-\mathrm{b},\mathrm{i}}$'s. C is already aware of $k-1$ shares of $u_{2,\mathrm{i}}$ and $u_{-\mathrm{a},\mathrm{i}}$ from the last time period. Since the $u_{-\mathrm{a},\mathrm{i}}$'s have been replaced by $u_{-\mathrm{b},\mathrm{i}}$'s in the new time period, the adversary can find out U only by recovering additional shares of $u_{2,\mathrm{i}}$. For this, C needs to be aware of some of the $u_{3,\mathrm{i}}$'s.

The $u_{3,\mathrm{i}}$'s are however unavailable in plain text and have been "added" to the shares of all the data objects. The $u_{3,\mathrm{i}}$'s can be recovered only by compromising servers while the share renewal process was in progress. Servers compromised during the share renewal phase, however, are considered as compromised in the current and the last time periods. Hence, an adversary who has compromised a server during the share renewal phase will anyway be aware of the $u_{2,\mathrm{i}}$ stored at the compromised server. So new $u_{2,\mathrm{i}}$'s can be found by the adversary.

Also, it can be seen that not more than $k-1$ $u_{3,\mathrm{i}}$'s can be found by the adversary. If the adversary was able to find a certain $u_{3,\mathrm{i}}$, that would have been possible only because the adversary was aware of the $u_{2,\mathrm{i}}$ stored at that server. The adversary could not recover more than $k-1$ $u_{2,\mathrm{i}}$'s used in the earlier time period. By induction, it can be seen that even after subsequent time periods and share renewals, k consistent shares of U can never be found. It can also be seen that the knowledge of user secrets in addition to user secret S_2 does not help the adversary C in any way. The adversary is hence unable to find out the system secret U. ∎

Theorem 2 *An adversary C, who has read and write access to some user secrets, cannot recover a user secret to which he has no read access.*

Proof: Assume a client / adversary C knows user secrets $S_2, S_3, ...$ and is interested in finding user secret S_1. To find secret S_1, the adversary must obtain at least k consistent

shares of S_1 by compromising as many servers. Since a random set of shares of system secret U is used to store S_1 when it is written, and this set of shares of U is never used again to store other secrets, there is no correlation between shares of S_1 and any other secret. Hence, the knowledge of other secrets is of no use to the adversary interested in finding the secret S_1.

Assume the set of shares of U used to encode S_1 as $S_1 \oplus U$ when S_1 was written are denoted by $u_{2,\mathrm{i}}$'s. By the compromise of $k-1$ servers at the time the secret S_1 was written, the adversary C may be able to obtain $k-1$ shares of S_1 and $k-1$ shares of U used to store S_1 as $S_1 \oplus U$. In the next time period and following the share renewal, the adversary can obtain $k-1$ $(s_{1,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes u_{3,\mathrm{i}})$'s, where the $u_{3,\mathrm{i}}$'s are the shares of U used to renew the shares of *all* the user secrets during the share renewal. The adversary has to now compromise a server other than the servers compromised in the earlier time period in order to recover new shares of S_1. Since at the currently compromised servers, the $(u_{2,\mathrm{i}} \otimes u_{3,\mathrm{i}})$'s are not known to the adversary (from Theorem 1, the system secret U cannot be found by the adversary), the adversary will be unable to recover k consistent $s_{1,\mathrm{i}}$'s. It can be seen that this result holds during subsequent share renewals also.

The secret S_1 can also be recovered from k consistent shares of $S_1 \oplus U \oplus U^{-1}$, or from k consistent shares of $S_1 \oplus 2U \oplus 2U^{-1}$. That is, S_1 can be found from k consistent $(s_{1,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes u_{-\mathrm{a,i}})$'s, or k consistent $(s_{1,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes u_{3,\mathrm{i}} \otimes 2u_{-\mathrm{b,i}})$'s. Since the shares used to store U^{-1}, the $u_{-\mathrm{a,i}}$'s, are changed (renewed) at every time period boundary (in this case, to $u_{-\mathrm{b,i}}$'s), it is not possible to obtain k consistent $u_{-\mathrm{a,i}}$'s or $u_{-\mathrm{b,i}}$'s. Hence, k consistent $(s_{1,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes u_{-\mathrm{a,i}})$'s or k consistent $(s_{1,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes u_{3,\mathrm{i}} \otimes u_{-\mathrm{b,i}})$'s cannot be recovered by the adversary.

Hence, even with the knowledge of some user secrets and with the ability to compromise at most $k-1$ servers in every time period, the adversary will be unable to find out other user secrets. ∎

From a practical standpoint, it may be advisable to change the system secret every now and then because the security of the entire system rests on maintaining the secrecy of the system secret. Even after the system secret is changed, the past system secrets should never be discovered. This is because an adversary could have collected shares over several time periods, and with the knowledge of the system secret used then, he will be able to decipher user secrets stored in those time periods. The system secret must therefore be changed as a safety precaution.

To change the system secret from U to V, the procedure used to setup shares of U and U^{-1} are followed again to obtain shares of V and V^{-1}. Note that the system secrets are not known even to the system administrator as each server generates its own random share. For a secret, say S_1, stored in the system, server i computes its new share of S_1 with respect to system secret V by computing $s_{1,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes ... \otimes u_{3,\mathrm{i}} \otimes R_{\mathrm{S_1}} u_{-\mathrm{a,i}} \otimes v_{2,\mathrm{i}}$, where the $u_{-\mathrm{a,i}}$'s are the current shares used to store U^{-1} and the $v_{2,\mathrm{i}}$'s are the shares of V "added" ($\otimes$) to the shares of *all* the user secrets. $R_{\mathrm{S_1}}$ is initialized back to 1. When S_1 is read, the shares returned will be of the form $s_{1,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes ... \otimes u_{3,\mathrm{i}} \otimes R_{\mathrm{S_1}} u_{-\mathrm{a,i}} \otimes v_{2,\mathrm{i}} \otimes v_{-\mathrm{a,i}}$, where the $v_{-\mathrm{a,i}}$'s are the shares currently being used to store V^{-1}.

The following theorem states that when the system secret is changed, the confidentiality

of the user and system secrets will be maintained.

Theorem 3 *Let the secret U be changed to V. An adversary who has read and write access to some user secrets will not be able to recover V or any other user secret in the process.*

Proof: When U is changed to V, a fresh set of shares, $v_{-\mathrm{a,i}}$'s, are used to store V^{-1}, and another set of shares, $v_{2,\mathrm{i}}$'s, are used to refresh the shares of *all* the secrets stored in the system.

From Theorem 1, where the knowledge of some secret is assumed in order to find U, it is clear that the present system secret V cannot be learnt.

In order to prove that the user secrets cannot be found when the current system secret U is changed to V, consider the time periods before and after the share renewal phase when U was changed to V. Denote these time periods as T_1 and T_2 respectively.

Going back to the proof of Theorem 2, at the time of the first share renewal of a secret, the same set of shares of U was added to the shares of all the secrets stored in the system. The adversary knew a maximum of $k-1$ shares of $s_{1,\mathrm{i}} \otimes u_{2,\mathrm{i}}$ or $s_{1,\mathrm{i}} \otimes u_{2,\mathrm{i}} \otimes u_{3,\mathrm{i}}$, and it was shown that the adversary would not be able to find out secret S_1.

Likewise, in this case, a maximum of $k-1$ shares of secret S_1 as stored in the system at time period T_1 are known to the adversary. From Theorem 2, more than $k-1$ shares cannot be known to the adversary. Instead of adding a new set of shares of U and refreshing the shares of U^{-1} during share renewal, here a new set of shares of a new system secret V are being added to the shares of all the secrets stored in the system, and a new set of shares are used to store V^{-1}. Noting that the $v_{-\mathrm{a,i}}$'s used to store V^{-1} are share-renewed at the time period boundaries, the analysis to prove that secret S_1 cannot be found by the adversary is similar to Theorem 2. ∎

6 Performance Analysis

In this section, we show the benefits gained by the proposed algorithm over traditional methods where each secret (or data object) is share-renewed independently.

Shamir's secret sharing scheme [16] along with the share renewal scheme proposed in Herzberg et al. [21] is used for the analysis. In Shamir's scheme, the Abelian groups AG_1 and AG_2 are both Z_{p}, where p is a prime number. As mentioned earlier, Shamir's scheme is a $(+,+)$-homomorphic (k,n)-threshold secret sharing scheme.

A brief overview of the share renewal algorithm for Shamir's scheme proposed in Herzberg et al. [21] is as follows: To renew a secret, each server uses a (k,n)-threshold Shamir's scheme to generate n shares of $0_1 = 0$, the zero element in AG_1. This step involves computing $(k-1)$ modular multiplications and $(k-2)$ modular additions. These n shares are given to the respective servers securely. Each server thus receives n shares of 0, and it adds these shares to its share of the secret which is being share-renewed. This step therefore involves n modular additions. The network communication cost is $n^2 l$ bits, where l is the length in bits of a share.

	Proposed Algorithm	**Traditional Approach**
Computation Overhead during writes	$(k-1)$ mod mults $(n+k-2)$ mod adds	0 mults 0 adds
Communication Overhead during writes	n^2l bits	0 bits
Computation Overhead during share renewal	$3(k-1)$ mod mults $3(n+k-2)+N_{\mathrm{D}}$ mod adds	$N_{\mathrm{D}}(k-1)$ mod mults $N_{\mathrm{D}}(n+k-2)$ mod adds
Communication Overhead during share renewal	$3n^2l$ bits	$N_{\mathrm{D}}n^2l$ bits
Computation Overhead during reads	1 mod mult 1 mod adds	0 mod mults 0 mod adds

Table 1: **Comparison between the proposed algorithm against independent share renewal of each secret**

In our proposed algorithm, there are computation and communication overheads during reads and writes, which straightforward share renewal of each secret does not incur any overheads during reads and writes. Note that we are restricting our study to passive adversaries, where only the confidentiality of the data is at stake and there is no malicious behavior of servers. Table 1 compares our proposed algorithm against the straightforward case where each secret is managed independently (called the "traditional approach").

The efficiency of the share renewal of all the stored data is clear from the table. Multiplication operations incur a higher overhead than addition operations, and the proposed algorithm performs only a constant number of multiplications which is independent of the number of user secrets N_{D}. The number of additions that need to be performed are also reduced by a factor of $(n+k-2)$. In the proposed algorithm, three runs of the share renewal protocol must be done: one for preserving the confidentiality of U, another for U^{-1}, and finally to generate a new sharing to renew all the user secrets. This contrasts with the traditional approach where each secret is independently share-renewal, thus leading to N_{D} runs of the share renewal protocol. Thus, share renewal using our proposed algorithm can complete sooner. This is essential in the proactive security model, which assumes that no more than $k-1$ servers are compromised in each time period. By quick share renewal of all the user secrets, chances of compromising a server in the current time period and obtaining shares in the earlier time period is very much reduced.

These benefits come at the cost of overheads during writes and reads. There is one invocation of the share renewal protocol during writes in the proposed algorithm. For data that is archival in nature, it can be expected that there are very few writes in a time period while the number of stored user secrets could be very large. So this tradeoff may be worthwhile. Moreover, when active adversaries or faulty clients are considered, servers must engage in an agreement protocol during writes to ensure that the received shares are consistent (any k shares give the same secret). A share renewal protocol can be piggy-backed

onto such an agreement protocol. Due to space constraints, we do not give details on such methods.

The performance penalty incurred during a read is negligible. For each read, a server has to perform only one modular multiplication and one modular addition, but traditional schemes do not incur any overhead. Read requests arrive at different times during a time period, while share renewal of all user secrets is done at the time period boundaries. This share renewal must complete as quickly as possible, and this is paramount. The proposed algorithm achieves this while incurring some overheads during reads, which is a very worthwhile tradeoff.

7 Conclusions

An algorithm for efficient share renewal of a large number of secrets is given. The algorithm achieves efficiency by refreshing shares of all the secrets by the share renewal of only one secret, called the "system secret." The efficiency is gained at the expense of some performance penalties during reads and writes, but there is no loss of security. If the underlying algorithms provide information-theoretic secrecy, then the proposed algorithm maintains that level of security.

The proposed algorithm is well suited in data storage applications where data is archival in nature, and long-term confidentiality of the data must be provided. Since secret sharing techniques form the basis of the proposed work, applications where key management is a problem will also find the proposed algorithm attractive.

References

[1] M. Herlihy and J. D. Tygar, "How to make replicated data secure," in *Crypto*, 1987.

[2] G. Agrawal and P. Jalote, "Coding based replication schemes for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 3, pp. 240–251, 1995.

[3] A. Iyengar, R. Cahn, C. Jutla, and J. Garay, "Design and implementation of a secure distributed data repository," in *Proceedings of the* 14$^{\text{th}}$ *IFIP International Information Security Conference*, 1998.

[4] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos, "A prototype implementation of archival intermemory," in *Proceedings of the* 4$^{\text{th}}$ *ACM International Conference on Digital Libraries*, 1999.

[5] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of the* 9$^{\text{th}}$ *ASPLOS*, 2000.

[6] R. J. Anderson, "The eternity service," in *Proceedings of the* 1$^{\text{st}}$ *International Conference on Theory and Application of Cryptography (Pragocrypt)*, 1996.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[8] M. Waldman, A. D. Rubin, and L. F. Cranor, "Publius: A robust, tamper-evident, censorship-resistant web publishing system," in *Proceedings of the* 9th *Usenix Security Symposium*, 2000.

[9] R. Dingledine, M. J. Freedman, and D. Molnar, "The free haven project: Distributed anonymous storage service," in *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[10] S. Lakshmanan, M. Ahamad, and H. Venkateswaran, "A secure and highly available distributed store for meeting diverse systems and networks," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2001.

[11] L. Kong, A. Subbiah, M. Ahamad, and D. Blough, "A reconfigurable Byzantine quorum approach for the agile store," in *Proceedings of the International Symposium on Reliable Distributed Systems*, 2003.

[12] A. Subbiah, M. Ahamad, and D. M. Blough, "Using Byzantine quorum systems to manage confidential data," Tech. Rep. GIT-CERCS-04-13, Georgia Tech Center for Experimental Research on Computer Systems, 2004.

[13] "The agile store project." http://www.ece.gatech.edu/research/labs/agile_store.

[14] "Pasis." http://www.pdl.cmu.edu/Pasis.

[15] "Mojonation." http://www.mojonation.net.

[16] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[17] G. R. Blakley, "Safeguarding cryptographic keys," in *Proceedings of the National Computer Conference*, 1979.

[18] T. M. Wong, C. Wang, and J. M. Wing, "Verifiable secret redistribution for archive systems," in *Proceedings of the* 1st *International IEEE Security in Storage Workshop*, 2002.

[19] S. Lakshmanan, M. Ahamad, and H. Venkateswaran, "Responsive security for stored data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 9, pp. 818–828, 2003.

[20] R. Ostrovsky and M. Yung, "How to withstand mobile virus attacks," in *Proceedings of the* 10th *Symposium on the Principles of Distributed Computing*, 1991.

[21] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *Crypto*, 1995.

[22] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor, "Proactive security: Long term protection against breakins," *RSA Laboratories' Cryptobytes*, vol. 3, no. 1, 1997.

[23] J. Benaloh, "Secret sharing homomorphisms: Keeping shares of a secret secret," in *Crypto*, 1986.

[24] Y. Desmedt and S. Jajodia, "Redistributing secret shares to new access structures and its applications," Tech. Rep. ISSE TR-97-01, George Mason University, 1997.