# Self Destructing Data System Based On Session Keys

N.S.Jeyakarthikka, S.Bhaggiaraj, A.Abuthaheer,

**Abstract:** Personal data stored in the cloud may contain account numbers, secret codes and other necessary details that could be used and misused. These data can be cashed, copied by cloud service provider without user's control. We propose a self destructing data which mainly aims at protecting the user data's privacy by making the sensitive data automatically destructed after a period of time. First encrypt the data into cipher text and then distribute both the decryption key and the cipher text into a distributed hash table. To recover the plain text, both the decryption key and the cipher text should be obtained from the DHT before the pre-configured period of time.

**Keywords:** Cloud computing, Data privacy, self-destructing data, Distributed hash table (DHT).

## 1. Introduction

With the development of cloud computing and popularization of mobile web, cloud online services are becoming more useful for people's life. People are requested to submit their personal information to the cloud by the internet. When people do this, they hope the service provider will provide security to protect their data from leaking, so other people will not invade their privacy. With more and more services and applications are emerging in the internet, it becomes easier to expose the user's sensitive data in the internet. Without much attention to this problem, it will result in the break out of leaking messages. For e.g. Emails can be leaked via cloud service provider, negligence or hackers. One of the most important reasons for exposing sensitive user data is that the user data will be stored in the cloud environment for a long time, which may not be controlled by the user himself. These are the major challenges to protect people's privacy. Self-destructing data systems are designed to address these concerns. Their goal is to destroy data after a pre-specified timeout. Self destruction is implemented by encrypting data with a key and then escrowing the information needed to reconstruct the decryption key with one or more third parties. Assuming that the key reconstruction information disappears from the escrowing third parties at the intended time, encrypted data will become permanently unreadable: (1) even if an attacker obtains a copy of the encrypted data and the user's cryptographic keys and passphrases after the timeout, (2) without the user or user's agent taking any explicit action to destroy it, (3) without need to change any stored or archived copies of that data, and (4) without the user relying on secure hardware. Once the key-renovation details disappears, data owners can be confident that their data will remain inaccessible to powerful attacks, whether from hackers who obtain copies of backup archives and passphrases or through legal means.


- *N.S.Jeyakarthikka, PG-Scholar, Sri Ramakrishna Eng Coll, Coimbatore.*
- *S.Bhaggiaraj, Asst-Professor, Sri Ramakrishna Eng Coll, Coimbatore.*
- *A.Abuthaheer, PG-Scholar, Sri Ramakrishna Eng Coll,Coimbatore*


## 2. Related Works

Vanish [1] is a system for creating messages that automatically self-destruct after a period of time. Vanish encapsulates data objects so that they "self-destruct" after a specified time, becoming permanently unreadable. It encrypts the data using a randomly generated key and then uses Shamir secret sharing [2] to break the key into n shares where k of them are needed to reconstruct the key. Vanish stores these shares in random indices in a large, pre-existing distributed hash table. After they expire, the key is permanently unavailable, and the encrypted content is permanently unreadable. However, Sybil attacks [3] may compromise the system by continuously crawling the DHT and saving each stored value before it ages out. They can efficiently recover keys for more than 99% of Vanish messages. Wolchok *et al.* [3] concludes that public DHTs like Vuze DHT [7] probably cannot provide strong enough security for Vanish. So, Geambasu *et al.* [6] proposes two main countermeasures. Although using both OpenDHT [8] and VuzeDHT can provide the maximum security that is derived from either system: if both DHTs are insecure, then the hybrid will also be insecure. Vanish is an interesting approach, it provide security to people's confidential data but, in its current form, it is insecure [3]. To address the problem of Vanish discussed above, in our previous work [4], we proposed a new scheme, called *Safe Vanish*, to prevent *hopping attack*, which is one kind of the *Sybil attacks* [9], [10], by extending the length range of the key shares to increase the attack cost, and did some improvement on the Shamir Secret Sharing algorithm and is implemented in the Vanish system. Our contributions are summarized as follows.

- We focus on the related key distribution algorithm, Shamir's algorithm [2], which is used as the core algorithm to implement client (users) distributing keys in the storage system. We can use this algorithm to implement a safety destruct with equally divided keys (Shamir Secret Shares [2]).
- Based on active storage framework, object-based storage interface is used to store and manage the equally divided key.
- Through functionality and security properties evaluation of the SeDas prototype, the results demonstrate that SeDas meets all the privacy problems. The prototype system imposes reasonably low runtime overhead.

- SeDas supports security for files and random encryption keys stored in a hard disk drive (HDD) or solid state drive (SSD), respectively.

In this paper the proposed system focus on the related key distribution algorithm, Shamir's algorithm, this is used as the core algorithm to implement client (users) distributing keys in the object storage system. Using these methods we can implement a safety destruct with equal divided key Shares. Based on active storage framework [5] and an object-based storage interface to store and manage the equally divided key. Through functionality and security properties evaluation of the SeDas prototype, the results demonstrate that SeDas meets all the privacy-preserving goals. The prototype system imposes low run time overhead. SeDas supports security for files and random encryption keys stored in a hard disk drive (HDD) or solid state drive (SSD), respectively. To improve the key split up in SeDas we can use Short Share Secret Sharing Algorithm.

## 3. Architecture Diagram

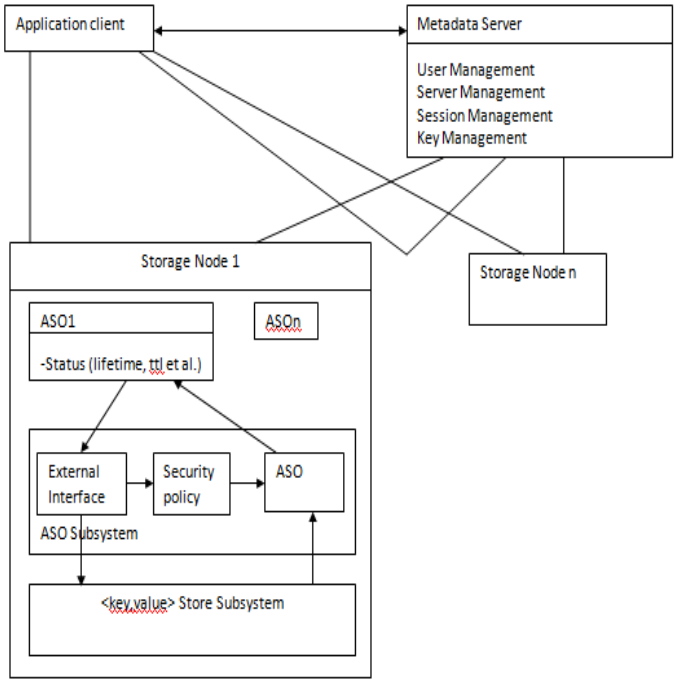


**Fig 1:** SeDas Architecture

Fig 1 shows the active storage framework. In this there are three parties Metadata server (MDS): MDS is responsible for user management, server management, session management and file metadata management. Application node: The application node is used to store the services of SeDas. Storage node: Storage node is a client and it contains two subsystems:

**Key value store subsystem**
The key value store subsystem that is based on the object storage component is used for managing objects stored in storage node. The object ID is used as a key. The related data and their attributes are stored as values.

**Active storage objects (ASO) runtime subsystem.**
The ASO is based on the active storage agent module in the object-based storage system is used to process active storage [5] request from users and manage method objects and policy objects.

### 3.1 Active Storage Object
An active storage object derives from a user object and has a time-to-live (TTL) property. The TTL value is used to start the self-destruct operation. The TTL value of a user object is infinite so that a user object will not be deleted until a user deletes it manually. The TTL value of an active storage object is limited so an active object will be deleted when the value of the associated policy object is true. Interfaces extended by Active Storage Object class are used to manage TTL value.

### 3.2 Data Process
To use the SeDas system, user's applications should implement logic of data process and act as a client node. There are two different processes: uploading and downloading.

#### 3.2.1 Uploading file process
The user uploads a file to a storage system and stores the key in the key management system, the user should specify the file, the key and *TTL* as arguments for the uploading procedure. This process assumes data and key has been read from the file. The procedure uses a common encryption algorithm or user-defined encryption algorithm. After uploading data to storage server, key shares generated by Shamir Secret Sharing algorithm [2] will be used to create active storage object (ASO) in storage node in the SeDas system

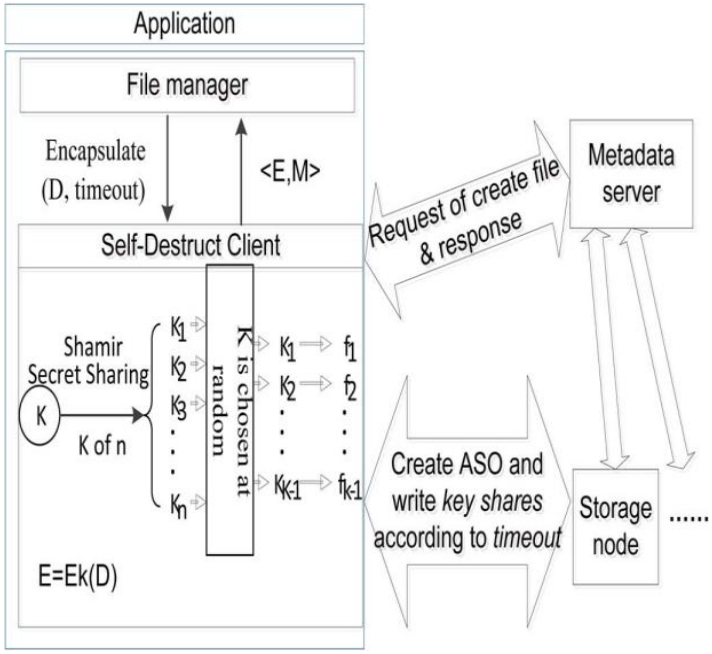


**Fig 2:** Uploading file process.

#### 3.2.2 Downloading file process
Any user who has relevant permission can download data stored in the data storage system. The data must be decrypted before use. The whole process is implemented in

user's application. This process assumes that encrypted data and Meta information of the key has been read from the downloaded file. Before decrypting, client should try to get key shares from storage nodes in the SeDas system. If the time expires been, the client cannot get enough key shares to rebuilt the key successfully. If the associated ASO of the key has been destructed, the client cannot rebuild the key so he can read only the encrypted data.

**Algorithm 1:** Uploading the file using Shamir secret key algorithm for key splitting.
Procedure Upload File (data, key, TTL)
Data: data read from this file to be uploaded
Key: data read from the key
TTL: time-to-live of the key
Begin//encrypt the input data with the key
Buffer = Encrypt (data, key)
Connect to a data storage server;
if failed then return fail;
Create file in the data storage server and write buffer into it;
// use Shamir Secret Sharing algorithm to get key shares
// k is count of data servers in the SeDas system
Shared keys [1....k] = Shamir Secret Sharing Split (n, k, key)
For i from 1 to k then
Connect to DS[i]
If successful then create_object (shared kyes[i], TTL);
Else
For j from 1 to i then
Delete key shares created before this one;
End for
Return fail;
End if
End for
Return successful;
End.

**Algorithm 2:** Uploading the file using short share secret key for content splitting.
Procedure Upload File (data, key, TTL)
Data: data read from this file to be uploaded
Key: data read from the key
TTL: time-to-live of the key
Begin
//encrypt the input data with the key
Buffer = Encrypt (data, key)
Connect to a data storage server;
if failed then return fail;
Create file in the data storage server and write buffer into it;
// use Short Share Secret Sharing algorithm to get key shares
// k is count of data servers in the SeDas system
Shared Content [1...k] = Block Split (n, k and buffer)
Shared keys [1....k] = Short Share Secret Sharing Split (n, k, key)
For i from 1 to k then
Connect to DS[i]
If successful then
create_object (sharedkyes[i], TTL);
create_object (sharedContent[i], TTL);
Else
For j from 1 to i then
Delete key shares created before this one;
Delete Content shares created before this one;
End for
Return fail;
End if
End for
Return successful;
End

## 4. Results and Discussion

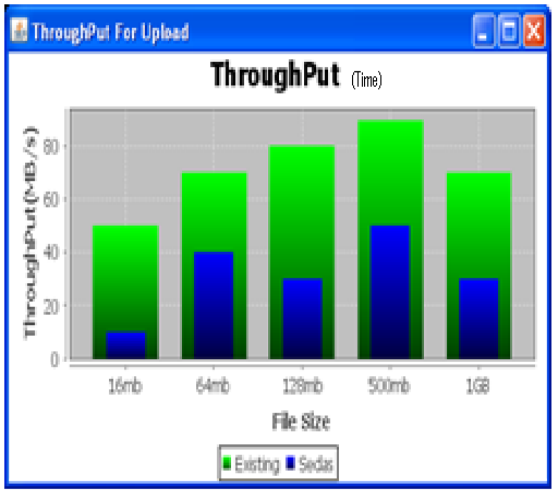


**Fig 3:** Throughput for Upload

In result Fig 3 shows the throughput for upload. X axis represents the throughput Y axis represents the file size. Throughput is how much mb is calculated in one second. In existing system, 16mb was uploaded in 50s, where as in proposed system it takes 10s to upload a 16mb file. Compare to the existing system the performance of proposed system is higher. This graph clearly shows the proposed system reduces the throughput over the existing system by an average of 15.5% and up to 55% for the uploading.

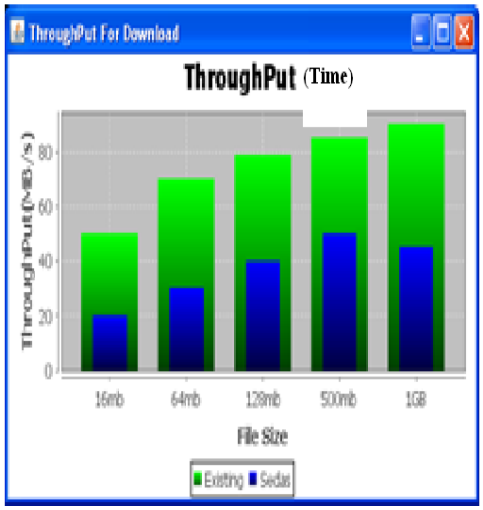


**Fig 4:** Throughput for Download

In Result Fig.4 shows the throughput for upload. X axis represents the throughput Y axis represents the file size. In existing system, 16mb was downloaded in 50s, where as in proposed system it takes 18s to download a 16mb file.

Compare to the existing system the performance of proposed system is higher. Fig.4 shows that proposed system reduces the throughput over the existing system by an average of 18% and up to 50.75% for the downloading. Fig.4 and Fig.5 shows the throughput results for the different schemes. The throughput decreases because upload/download processes require much more CPU computation and finishing encryption/ decryption processes in the proposed system, compared with the existing system.

**Result Table**

**Table 1: Performance evaluation for uploading**

| FILE SIZE | EXISTING | PROPOSED |
|---|---|---|
| 16MB | 50S | 20S |
| 64MB | 65S | 45S |
| 124MB | 80S | 30S |
| 500MB | 85S | 50S |
| 1GB | 30S | 25S |

Table1 shows the performance evaluation for uploading. In existing system, 16mb was uploaded in 50s, where as in proposed system it takes 20s to upload a 16mb file. Compare to the existing system the performance of proposed system is higher

**Table 2: Performance evaluation for downloading**

| FILE SIZE | EXISTING | PROPOSED |
|---|---|---|
| 16MB | 47S | 25S |
| 64MB | 59S | 43S |
| 124MB | 73S | 60S |
| 500MB | 86S | 73S |
| 1GB | 96S | 85S |

Table 2 shows the performance evaluation for downloading. In existing system, 16mb was uploaded in 47s, where as in proposed system it takes 25s to download a 16mb file. Compare to the existing system the performance of proposed system is higher.

## 5. Conclusion and Future Work

## Conclusion

Data privacy is essential in the Cloud environment. A new approach is introduced for protecting the data privacy from attackers which may obtain, from legal or other means, a user's stored data and private decryption keys. A novel aspect is the leveraging of the essential properties of active storage framework based on T10OSD standard. Personal data stored in the cloud may contain account numbers, secret codes and other necessary details that could be used and misused. SeDas uses the self-destruct operation without any action on the user's part. Measurement and experimental security analysis sheds insight into the practicability of this approach. Plan to release the current SeDas system will help to provide researchers with further valuable experience to inform future object-based storage system designs for Cloud services.

## Future Work

Short share secret scheme is mainly used for split the key and reconstruct the key for user. The key may be corrupted, and the malicious shares are also not identified and recovered. So future work extended to identify and recover the corrupted/malicious shares using Robust Secret Sharing scheme.

## References

[1]. R. Geambasu, T. Kohno, A. Levy, and H. M. Levy, "Vanish: Increasingdata privacy with self-destructing data," in Proc. USENIX Security Symp., Montreal, Canada, Aug. 2009, pp. 299–315.

[2]. A. Shamir, "How to share a secret," Commun. ACM, vol. 22, no. 11, pp. 612–613, 1979.

[3]. S. Wolchok, O. S. Hofmann, N. Heninger, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel, "Defeating vanish with low-cost sybil attacks against large DHEs," in Proc. Network and Distributed System Security Symp., 2010.

[4]. L. Zeng, Z. Shi, S. Xu, and D. Feng, "Safevanish: An improved data self-destruction for protecting data privacy," in Proc. Second Int. Conf. Cloud Computing Technology and Science (CloudCom), Indianapolis, IN, USA, Dec. 2010, pp. 521–528.

[5]. L. Qin and D. Feng, "Active storage framework for object-based storage device," in Proc. IEEE 20th Int. Conf. Advanced Information Networking and Applications (AINA), 2006.

[6]. R. Geambasu, J. Falkner, P. Gardner, T. Kohno, A. Krishnamurthy, and H. M. Levy, Experiences building security applications on DHTs UW-CSE-09-09-01, 2009, Tech. Rep..

[7]. Azureus, 2010 [Online]. Available: http://www.vuze.com/

[8]. S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "OpenDHT: A public DHT service and its uses," in Proc. ACM SIGCOMM, 2005.

[9]. J. R. Douceur, "The sybil attack," in Proc. IPTPS '01: Revised Papers from the First Int. Workshop on Peer-to-Peer Systems, 2002.

[10]. T. Cholez, I. Chrisment, and O. Festor, "Evaluation of sybil attack protection schemes in kad," in Proc. 3rd Int. Conf. Autonomous Infrastructure, Management and Security, Berlin, Germany, 2009, pp. 70–82.