Multi-part Dynamic Key Generation For Secure Data Encryption

Srivatsan Iyer *srivatsan.iyer.1990@gmail.com*
Software Engineer
Webshar LLC
Mulund, 400080, India

Tejas Arackal *tjeidolon@gmail.com*
Software Engineer
TCS TataCapital
Andheri Seepz, 400096, India

Abstract

Storage of user or application-generated user-specific private, confidential data on a third party storage provider comes with its own set of challenges. Although such data is usually encrypted while in transit, securely storing such data at rest presents unique security challenges. The first challenge is the generation of encryption keys to implement the desired threat containment. The second challenge is secure storage and management of these keys. This can be accomplished in several ways. A naive approach can be to trust the boundaries of a secure network and store the keys within these bounds in plain text. A more sophisticated method can be devised to calculate or infer the encryption key without explicitly storing it. This paper focuses on the latter approach. Additionally, the paper also describes the implementation of a system that in addition to exposing a set of REST APIs for secure CRUD operations also provides a means for sharing the data among specific users.

Keywords: Encryption, Dynamic Key Generation, Data At Rest Security, CRUD.

1. INTRODUCTION

Multiple international regulations[1], as well as expectations of users require that any system that stores private user-identifiable data take appropriate steps to ensure its privacy, security and integrity. Over and above these primary goals, are the requirements for the system to have very reliable and readily available storage[2]. Many applications, for reasons of business requirements, including ease of use, system robustness and data recovery choose to store data with third-party storage providers.

In typical cases such as above, the choice is between: a) Trust and allow the storage provider to encrypt the data before writing it out to the disk, or b) proactively encrypt the data prior to sending it over the wire. The prior option is very easy, and straightforward. If the storage provider does not encrypt data appropriately, the security of the data is at risk. The risk is multiplied when all the files are encrypted with the same key. The latter option, however, may be more secure especially since it gives the control of encryption to the application shepherding the data, or to the owner of the data itself.

This paper deals with the second option and tries to explore some of the many possibilities that come with it. As mentioned before, if the encryption is done at the premises, a mechanism is required to store the encryption key for the object being encrypted. If the key is lost, then the encrypted object might never get decrypted. It logically follows that management of the keys needs to be secure as well. In other words, the keys need to get the same treatment with respect to security as the original unencrypted objects. A very simple, unsophisticated approach would be to simply have an encryption key for the entire application, and all the data that it wants to store,

would be encrypted with this key. But this simplicity comes at the cost of security. If this key is compromised, then the entire data set is at risk.

This problem can be solved by assigning an encryption key per user, and storing it in the data store. This scheme is safer as compared to the previous one, because if a key is compromised only the files specific to that user are leaked; the remainder files are safe. Damages/leaks of this kind cannot be prevented fully. As the power of the hardware grows, it only becomes easier for attackers to orchestrate different kinds of attacks. This paper explores a very specific method to contain damages of these kinds.

This paper provides a brief overview of the architecture and framework of a system that can serve as a middleware between the application server and the storage layer. The benefit of such a system is that it can connect to existing data store to authenticate users and gives them access only to authorized resources.

2. LITERATURE SURVEY

The security of the storage in large scale applications has been a constant concern for a long time. Varied security models have been proposed considering the concerns of web business applications. Some common security factors such as encryption, access control, fault tolerance and high availability have been addressed in multi disparate view. While security models with trusted platform module implement a secure hardware location for encryption, authentication and attestation there are also some open source cloud computing applications that have their own implementation of security model.

The TCG trusted platform module (TPM)[3] implementation, using micro controller to store passwords, certificates & encryption keys, ensures security in terms of hardware-based cryptography, and hardware based approach to manage user authentication, network access and data protection. The confidentiality of the data is thus maintained by making it hard for the attacker to access information on computing device. The TPM is also considered a secure vault for storing keys, certificates and passwords. The module, by routinely inspecting the hardware status of the machines, is able to assert that hardware has not been tampered with. In one of the implementations of the specification, the authors of the paper maintain that storing the artifacts within the hardware is better than providing a software based security[4]. This approach will require changes to be implemented in the storage systems hardware in order to support TPM. As such there is a need for an approach that is much more portable and easily deployable than any TPM implementation.

OpenStack, an open source cloud computing software, uses an object storage system Swift[5], comprising of security model for encryption, authorization, access control and key management. The system provides different locations for the encrypted data and the encrypted keys associated with encryption. It maintains a master key for encrypting all other keys thus forming a two level encryption hierarchy. Additionally no implicit mapping is maintained between the data and the keys. The Openstack security model has its data encrypted and stored separately from the key that was used to encrypt it. For high-value data, OpenStack supports “dual-locking” in which every object is encrypted with a combination of “service key” and user-specific key[6]. Although this is secure, it can be improved by rotating keys among various files of the user so that if a key combination is somehow leaked only one file is affected.

OwnCloud, an open source project with similar core functionalities, is a “file-hosting” application that provides a web UI as well as APIs for desktop clients[7]. The system by default uses the server’s file system for storing incoming data, and can be configured to use different storage backend. Similarly, for user management it can be configured to work with many relational database servers. All the files being stored are encrypted with the password of the user. This has several implications. Firstly, since the application needs to encrypt and decrypt the data using clear text password, it should be able to retrieve the clear text password. This implies that the

passwords are not hashed. Secondly, since the user's login password and the data encryption key are identical, losing the password is equivalent to losing all the data. Thirdly, since all these passwords must be stored in the database, it becomes a high-risk component. If somehow the database gets compromised, data of all the users will be jeopardized.

As such there is a need for an approach that is much more portable & easily deployable than TPM implementation at the same time it should not store the direct key used for encryption either in encrypted or decrypted format within the system.

3. PROPOSED SYSTEM

To avoid the shortcomings of present security models, a model with additional improvements needs to implemented. The guidelines for this system are: a) using the same encryption key for multiple files increases the extent of damage if any key is compromised; b) for maximum security, system needs to use different keys for different files c) The key should be constructed by retrieving information from multiple sources. d) The system should let only authorized users access the data. The proposed design caters to all of the above requirements.

The solution that this paper proposes is named "Secure Data Storage Manager" (SDSM). SDSM is an application that operates from within an HTTP Web server placed in a secure network. It uses a User Data Store for storing various attributes of a user and a Metadata store (a NoSQL datastore) for storing metadata of the files stored in the storage layer. Both User Data Store and Metadata Store share the same secure network The two main functions of SDSM is to keep the information consistent between the metadata store and the storage layer, and to employ an appropriate encryption and decryption algorithm to deal with user data.

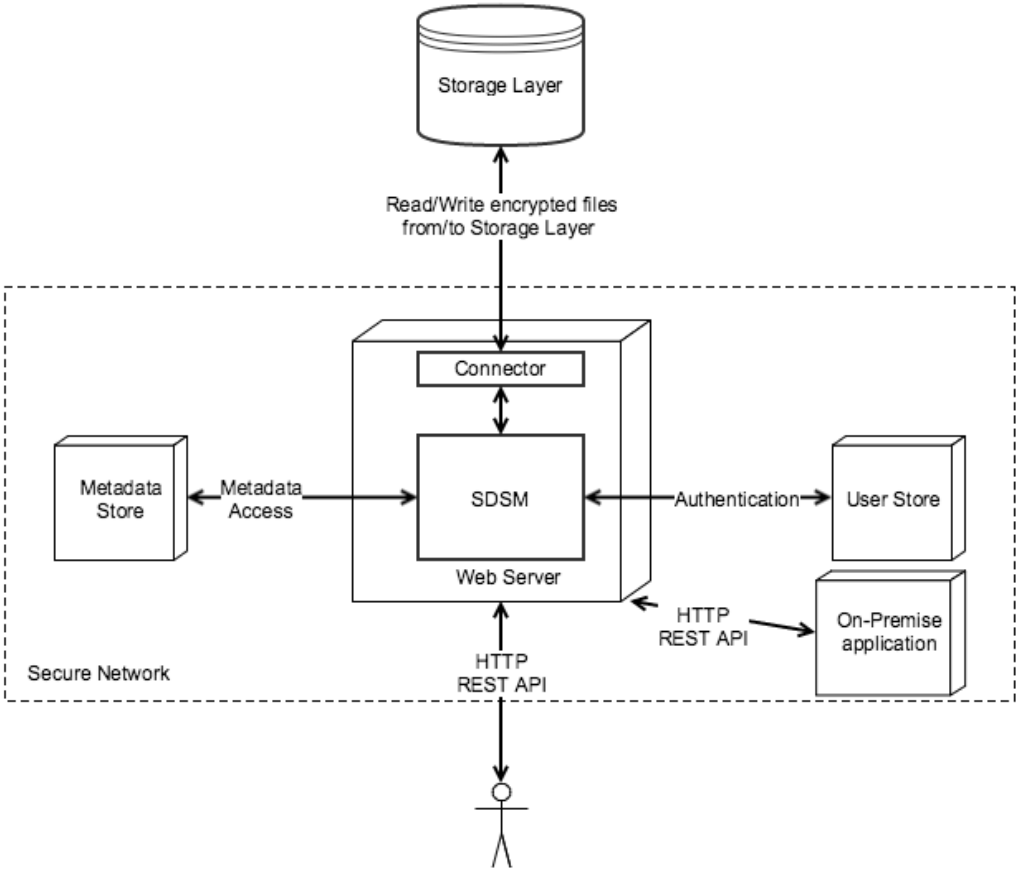

FIGURE 1: Proposed system architecture.

3.1 Metadata Store

The metadata store is essentially a key-value store, in which the keys are full file path on the storage layer. The value stored against each key is a complex structure that contains the following:

1. Timestamp of file creation and timestamp of last modification to the file.
2. Owner information
3. Hash of the file content
4. Access log list

Every file stored in the storage layer has a corresponding entry in the metadata store. The look-up key in this design is the full file path itself.

3.2 Encryption/Decryption

3.2.1 Preliminaries

Let's assume a mathematical function *'f'* to encrypt, and *'g'* to decrypt. Further assuming the cipher text to be represented by 'C' and original message to be represented by 'M', then the usual encryption/decryption process can be mathematically represented as:

$$C \equiv f_x(M),$$

$$M \equiv g_x(C),$$

$$M \equiv g_x(f_x(M)),$$

where, f_x and g_x are the encryption and decryption functions respectively, each initialised with the key 'x'.

Let's denote the metadata store by the letter *'S'*, the file path in consideration by letter 'P', the master key by letter 'K', function to retrieve the metadata structure against a file path by letter *'h'*. Note that metadata value *'V'* is encrypted with the master key. Now,

$$h(S, P) \rightarrow \{ V \text{ if } P \text{ exists, else null } \}.$$

Let there be a function 'j' that reads decrypted metadata structure to retrieve the last update timestamp *'T'* for the file. Assuming 'V' exists, mathematically,

$$j(g_K(V)) \rightarrow T.$$

The encrypted private key of the user who owns the file is represented by *'Q'*.

3.2.2 Encryption

Before encryption we update the metadata store for the access being made, specifically, there is an entry appended the access logs with mode as write. In other words, the function *j(..)*, returns the current timestamp.

To generate the encryption key $'R_1'$ for the file at P_1, we need to take the following steps:

$$h(S, P_1) \rightarrow V_1,$$

$$j(g_K(V_1)) \rightarrow T_1,$$

$$sha\text{-}256(\ string(T_1) + g_K(Q_1)) \rightarrow R,$$

where '+' operator stands for concatenation of strings.

The final ciphertext for the original message M_1 will be generated as:

$$f_{R1} (M_1) \rightarrow C_1.$$

The above encrypted payload will be uploaded to the storage layer

3.2.3 Decryption

To generate the symmetric decryption key *'S_1'* for the file at P_1, we need to take the following steps:

$$h(S, P_1) \rightarrow V_1$$

$$j(g_k(V_1)) \rightarrow T_1$$

$$sha\text{-}256(string(T_1) + g_K(Q_1)) \rightarrow R_1$$

The original message M_1 will be recreated from the ciphertext C_1 (read from the storage layer) as:

$$g_{R1}(C_1) \rightarrow M_1.$$

4. DESIGN

Secure Data Storage Manager System will consist of a web application server, a User Data Store (UDS), a Metadata Store (MS), a Temporary Credential Store (TCS) and a storage layer. The system expects only an interface to each of the sub systems. This allows the modules to be swapped out for a better alternative. Additionally, the application will benefit from not only flexibility, but also reliability because such systems are built purely for a singular purpose.

The SDSM will essentially be a set of REST API endpoints that clients can connect to. The user data, including the private keys of the users are securely stored within the UDS such as LDAP[8] for example. The credentials from the UDS are used for user authentication. Once authenticated, file metadata is retrieved from MS, for retrieving information related to the current request. The core of the SDSM is the set of procedures that operate upon the credentials and information retrieved from User Data store and metadata store respectively, to generate a symmetric key that will be used for encryption and decryption.

4.1 User Datastore Design

Every user account that is linked with our system will have a unique entry in the User Data Store (UDS). Each time a user tries to log into our system, authentication will be performed against UDS. Apart from general user information, every entry contains the hashed password, the secret key, and the user GUID. The password is used for authentication, while the secret key, which is generated when the user is onboarded, is used for encryption. It is assumed that this *secret key* is communicated to the end user via a different channel. The secret key and the hashed password are encrypted with the Master Encryption Key (MEK) of the system.

4.2 Temporary Credential Store Design

When a user authenticates with the system, the system will generate a temporary credential for use by clients in further requests. The schema for it is shown in the image below.

```
KEY = "0D513ACBB4DA4E60A7C37B5C1E6E5B87"
VALUE = "ZDhkMjFiM2VhZmYyNGYzNDk0NzJlNjQ5ZGRlZjkzYjYwYTg3MzBkOCAgLQo="
EXPIRE = 300 seconds
```

FIGURE 2: Temporary Credential Record Structure.

The key represents the user GUID and the value is a randomly generated string encoded in Base64 format when the user is successfully authenticated. The SDSM system can make use of the expiry feature typically provided by most NoSQL databases. The expiration of temporary credential implies that the requesting user will have to re-authenticate upon expiry of five minutes. This design of temporary credential reduces the use of passwords in requests.

4.3 Metadata Record Design

The preferred design to store the metadata entries is a simple key value store instead of a relation database. Every file stored in SDSM has a record in the Key-Value datastore. Multiple metadata attributes are associated with the file such as filename, path of file in the storage layer, creation time, owner information, file sharing information, access logs. The benefit of the key value store is that all the information that relates to a file is stored in a precise structure at one place. SDSM will seldom need to operate across multiple files. Also, the flexible schema allows the access logs as well as sharing information to be contained within the structure itself.

Like the user data store, all the values in the metadata store are encrypted with the MEK of the system. A typical decrypted record looks like below:

```
KEY = "/0D513ACBB4DA4E60A7C37B5C1E6E5B87/path/to/FileName.bin"
VALUE = {
            "timestamp_created" : "1407035466734",
            "timestamp_modified": "1407035497781",
            "hash": "50c29cb0b5cca6ddebd8b07b67407427c3ac0f9d",
            "owner" : "0D513ACBB4DA4E60A7C37B5C1E6E5B87",
            "shared_with": [{
                    "user": "9CEFC8F5447847D68469D8C202DE09FF",
                    "mode": "read,write"
                },{
                    "user": "3289098D55C3486E8A1CA25AF6A97AA3",
                    "mode": "read"
            }],
            "access_logs": [{
                    "user": "3289098D55C3486E8A1CA25AF6A97AA3",
                    "access_type": "create",
                    "timestamp": "1407035466734"
                },{
                    "user": "3289098D55C3486E8A1CA25AF6A97AA3",
                    "access_type": "read",
                    "timestamp": "1407035486734"
                },{
                    "user": "9CEFC8F5447847D68469D8C202DE09FF",
                    "access_type": "write",
                    "timestamp": "1407035497781"
            }]
        }
```

FIGURE 3: File Metadata Record Structure.

4.4 Interaction and Algorithm Design

The SDSM will support 3 categories of HTTPS requests -- a) Request to authenticate and generate a temporary credential, b) Request to perform CRUD operation on the files, c) Request to modify sharing and permission information.

The section below explains a set of algorithms to be followed for authentication and read/write operations. Other operations can be designed similar to the ones described below.

4.4.1 Generate Temporary Credential Request

The client will authenticate using an HTTPS POST request to the SDSM with the following parameters:

1. User GUID
2. Hashed Password

The server side will perform the following operation to authenticate the user and generate temporary credential for the user account.

1. Retrieve the GUID and hashed password from the request.
2. Retrieve values from LDAP. Decrypt the hashed password using MEK and proceed to authenticating the user
3. If request passes authentication, proceed with creation of temporary credential, else throw an error.
4. To generate the temporary credential SDSM will generate a secure random string and encode using Base64 to form temporary credential. This will be returned to the client.
5. SDSM will then create an entry into the Temporary Credential Store with Key as user GUID, temporary credential encrypted with MEK as value. This entry auto-expires after 300 seconds.

4.4.2 Create Secure Item Request

The client will send an HTTPS PUT request to the SDSM with the following parameters:

1. User GUID
2. Temporary credentials
3. File Content(FC)
4. File Path (with filename, relative to user's home directory)

The server side will perform the following operation in response to the request.

1. Retrieve the temporary credential from the requests, and validate it by looking up the user GUID in the Temporary Credential Store. Use MEK to decrypt the value in the Temporary Credential Store.
2. Retrieve the file content from the request.
3. Retrieve the secret key from the LDAP using the GUID. Decrypt it using MEK.
4. Concatenate the file content with the user's secret key and the timestamp provided within the request. Generate a SHA-256 hash.
5. Validate the hash for both the supplied and the server generated hash. If not valid throw an error.
6. CleanPath <- Sanitise-input-path(PathFromHTTPRequest)
7. FilePath <- "/" + <User GUID> + "/" + CleanPath
8. Ensure there is not entry against FilePath. Signal error otherwise.
9. Create a new record within the metadata store, with the following attributes, without committing to the store:
    a) Key <- FilePath
    b) Current user as the owner.
    c) Current timestamp for created and last modified
    d) SHA 256 Hash of the file content
    e) Shared with None
    f) Access Logs <- [{"user": "<current user>","access_type": "create","timestamp": "<time>"}]
10. Encrypt the incoming file
    a) Encryption key <- SHA-256 (<owner's secret key> + <last timestamp of last update/create from access logs>)
    b) Encrypt the incoming file with the above password.

11. Proceed to uploading the encrypted file to the Storage Layer
    a) File Name: Substring after the last slash.
    b) File Path: FilePath (from point 6)
12. Upon success, encrypt the new record using Master key, commit to metadata store and signal success. Else, rollback and signal error.

4.4.3 Read Secure Item Request

The client will send an HTTP GET request to the SDSM with the following parameters:

1. User GUID
2. Owner GUID
3. Temporary credentials
4. File Path (with filename, relative to owner-user's home directory)

In response to the request above, the server will perform the following actions:

1. Retrieve the temporary credential from the requests, and validate it by looking up the user GUID in the Temporary Credential Store. Use MEK to decrypt the value in the Temporary Credential Store.
2. CleanPath <- Sanitise-input-path(PathFromHTTPRequest)
3. FilePath <- "/" + <Owner GUID> + "/" + CleanPath
4. Check if FilePath exists in the metadata store. If not, error.
5. if User GUID != Owner GUID:
    a) Look into "shared_with" key. Read the array, and search for the element with "user" equal to the current user GUID. If none exist, signal item not found.
    b) If an item is found, check if "read" is a substring in "mode". If not, signal item not available for requested action.
6. Append to access_logs: {"user": "<guid>","access_type": "read","timestamp": "<time>"},
7. Fetch Owner's secret key from LDAP. Decrypt it using MEK.
8. Fetch the last update/create timestamp from the access_log
9. Key <- SHA-256(<owner's secret key> + <last timestamp of last update/create from access logs>)
10. From the storage layer, read the file at path = FilePath.
11. Decrypt the file using the Password in step 9.
12. Calculate the hash value of the decrypted file. Check if the hash within the metadata store is equal to the hash so generated. If not signal File externally modified error.
13. Send the decrypted file back to the user as the response to REST API request.

5. DISCUSSION

When storing the data on the third party storage device, confidential data should always be encrypted. As discussed before, encryption comes with the overhead of storing the encryption keys. A system can either be trusted to keep these keys safely, or the keys can be split securely between multiple systems. The SDSM relies on the fact that the chances of both systems getting simultaneously compromised is extremely thin. Even if one of the systems is attacked, the attacker will not be able to retrieve the key to decrypt the user data.

In the event of UDS getting compromised, the attacker will not be able to retrieve the password because they are hashed and encrypted with MEK. This also prevents the possibility of attacker sending the hashed password to obtain the temporary credentials. The private key is similarly encrypted. In the event of a breach in the MS, with every entry in the store encrypted, it is difficult for an attacker to retrieve the actual metadata value. The TCS, too, is encrypted similarly. This prevents the attacker from stealing the temporary credentials and making a request.

If the MEK is compromised it would culminate into a high risk situation. This can be prevented by using a highly secure mode of obtaining the MEK, which is out of the scope of the paper. The main benefit of the proposed system is that in the event of such an exposure, the MEK can be regenerated, and all encrypted values within dependent data stores -- TCS, MS and UDS can be re-encrypted. The data that stays on the storage layer can stay untouched. Finally, there is a possibility of a slightly different kind of attack wherein the encryption key for a file is somehow guessed by an attacker at the storage layer. In this case, the mechanism of the encryption key generation ensures that the key is distinct for every file. In other words, every other file will remain untouched.

Having seen the ability of the system to mitigate threats, it's worthwhile to compare it against the systems presented above. Trusted Computing Group's TPM needs a special hardware to be installed in the server where the data resides reducing the portability of the system. Moreover, if the encryption key is somehow compromised, the entire data on the disk can be decrypted. However, this is not the case in the proposed system as every file is encrypted with a different key.

When compared to OwnCloud's server side encryption mechanism, the encryption methodology provided in this paper is more reliable and resilient. The encryption in SDSM uses hashed passwords, providing an additional layer of security. The passwords in the OwnCloud's implementation provide for both authentication and encryption functionality. Thus the loss or modification of the login password has impact on encryption. Further, if the user store backend is a separate dedicated system like LDAP, then a change in the password without OwnCloud's knowledge will cause the data to be inaccessible until the password is reset to its original value. Moreover, change of password involves additional process in which all the objects stored in the storage layer will be re-encrypted with the new password. The separation of password for authentication and secret key for encryption in the proposed system ensures that the object still remains accessible even in the event of password modification or loss.

Interestingly, OpenStack offers dual locking. In this design, each object is encrypted with a combination of system's service key and user-specific key. This is more secure than a system that encrypts data using a fixed key. In this system, however, all files belonging to a specific user are still encrypted with the same key. SDSM takes this one step ahead and provides for different keys per file. In other words, encryption key of every file in the storage layer is unique.

6. FUTURE SCOPE

The system presented in the paper has a lot of scope for improvement. Some of them are described below.

The problem of concurrent updates is outside of the scope of this paper. The challenge in such concurrency control is that improper design can very easily lead to resource starvation. The system being *stateless* adds to the challenge of implementing it effectively. Since the concurrency-specific information will also be stored in the metadata store, there exists a possibility of stale concurrency specific information if it is being replicated or if the accesses to the store are not transactional and atomic. Designing a replication-aware system that implements efficient concurrency control would be a great improvement.

Currently, the update process overwrites the data at a particular location and leaves no way to retrieve the older version. A more favorable system would perform a *soft-deletion* instead of *hard-deletion*. In other words, data can be *versioned*. Fortunately, the non-relational structure of the metadata lends itself to storing version information. However, there are a few challenges involved with this enhancement — a) no two versions can have the same path on the file system. b) key of the metadata record no longer will be equivalent the actual path on the file system, c) with latest update timestamp being used for generating the decryption key, older versions can be decrypted

since they were encrypted at a different time. Fortunately, a system so designed need not focus on solving the problem of concurrent updates.

7. CONCLUSION

The challenges in securely storing users' confidential data into a datastore have given rise to an approach that would be vendor independent and theft secure. The SDSM system proposed in this paper is based on a set of guidelines for a very secure system, free of implementation dependencies. The elimination of single instance of encryption key by adopting a dynamic approach to determine encryption key, prevents the attacker from getting an access to the encryption key. The algorithm presented in the paper demonstrates the concept of dynamic generation of encryption key, as well as key rotation across files. This combination makes the system extremely secure.

8. REFERENCES

[1] E. McCallister, T. Grance, K. Scarfone. "Guide to Protecting the Confidentiality of Personally Identifiable Information (PII)." Internet: http://csrc.nist.gov/publications/nistpubs/800-122/sp800-122.pdf, Apr. 2010 [Sep 30, 2014].

[2] Taylor, N.E., Ives, Z.G. "Reliable storage and querying for collaborative data sharing systems." in Proc. International Conference on Data Engineering (ICDE), 2010, pp. 40-51.

[3] Trusted Computing Group. "TCG Specification Architecture Overview." Internet: http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf, Aug. 2, 2007 [Aug. 10, 2014].

[4] A. Patel and M. Kumar. (2013, Apr.). "A Proposed Model for Data Security of Cloud Storage Using Trusted Platform Module." International Journal of Advanced Research in Computer Science and Software Engineering. [On-line]. 3(4), pp. 862-866. Available: http://www.ijarcsse.com/docs/papers/Volume_3/4_April2013/V3I4-0430.pdf [Aug. 10, 2014].

[5] OpenStack. "Object Encryption: Extending Swift." Internet: https://wiki.openstack.org/wiki/ObjectEncryption, Jul. 8, 2013 [Aug. 10, 2014].

[6] OpenStack. "KeyManager" Internet: https://wiki.openstack.org/wiki/KeyManager, Apr. 23, 2013 [Sep. 28, 2014].

[7] OwnCloud. "ownCloud Administrators Manual" Internet: http://doc.owncloud.org/server/6.0/admin_manual/configuration/configuration_encryption.html Sep 9, 2014 [Sep . 28, 2014].

[8] T. Howes. (1995, Jul.). "The Lightweight Directory Access Protocol: X.500 Lite." CITI Technical Report. [On-line]. 95(8), pp. 1-9. Available: http://www.openldap.org/pub/umich/ldap.pdf [Aug. 10, 2014].