Sharing of Encrypted files in Blockchain Made Simpler

S. Sharmila Deva Selvi[1], Arinjita Paul[1], Siva Dirisala[2], Saswata Basu[2] and C. Pandu Rangan[1]

[1] Department of Computer Science and Engineering, IIT Madras, India.
sharmioshin@gmail.com, {arinjita,prangan}@cse.iitm.ac.in
[2] 0chain LLC, San Jose, USA.
{siva,saswata}@0chain.net

Abstract. Recently, blockchain technology has attracted much attention of the research community in several domains requiring transparency of data accountability, due to the removal of intermediate trust assumptions from third parties. One such application is enabling file sharing in blockchain enabled distributed cloud storage. Proxy re-encryption is a cryptographic primitive that allows such file sharing by re-encrypting ciphertexts towards legitimate users via semi-trusted proxies, without them learning any information about the underlying message. To facilitate secure data sharing in the distributed cloud, it is essential to construct efficient proxy re-encryption protocols. In this paper, we introduce the notion of proxy self re-encryption (**SE-PRE**) that is highly efficient, as compared to the existing PRE schemes in the literature. We show that our self encryption scheme is provably CCA secure based on the DLP assumption and our proxy re-encryption scheme with self encryption is CCA secure under the hardness of the Computational Diffie Hellman (CDH) and Discrete Logarithm (DLP) assumption. Our novel encryption scheme, called self encryption, has no exponentiation or costly pairing operation. Even the re-encryption in **SE-PRE** does not have such operations and this facilitates the service provider with efficiency gain.

1 Introduction

The recent explosion of data volumes and demand for computing resources have prompted individuals and organisations to outsource their storage and computation needs to online data centers, such as cloud storage. While data security is enforced by standard public-key encryption mechanisms in the cloud, secure data sharing is enabled by efficient cryptographic primitives such as proxy re-encryption (PRE). PRE enables re-encryption of ciphertexts from one public key into another via a semi-trusted third party termed $proxy$, who does not learn any information about the underlying plaintext. A user can delegate access to his files by constructing a special key, termed as re-encryption key, using which the proxy performs the ciphertext transformation towards a legitimate delegatee. PRE systems can be classified into unidirectional and bidirectional schemes based on the direction of delegation. They can also be classified into single-hop and multi-hop schemes based on the number of re-encryptions permitted. In this work, we focus on unidirectional and single-hop PRE schemes.

The current model of cloud storage is operated through centralised authorities, which makes such a system susceptible to single point failures and permanent loss of data. Recently, blockchain technology, initially designed as a financial ledger, has attracted the attention of researchers in a wide range of applications requiring accountable computing and auditability. Blockchain enabled distributed peer-to-peer cloud storage solutions are steadily replacing its centralised counterpart. A blockchain provides multiple parties to agree upon transactions and contracts in an immutable and auditable way. Decentralised applications such as dApp providers make use of this capability to provide services that are transacted in a publicly verifiable manner. When the service provided by the dApp is not directly from the dApp owner itself but from other third parties, it brings up additional challenges. How would the end user using the dApp trust that the unknown third party service provides used by the dApp are trust worthy? This issue is specifically addressed, for example, by the dApp called **0box** [1] provided by **0Chain**

[2]. Such a storage dApp allows any user to upload and share their files to their friends and families similar to many other popular storage services. However, most existing services trust the service provider and upload the content without any encryption. But 0box strives to provide zero-knowledge storage such that the third party storage providers will not know the uploaded content. This is achieved using an efficient CCA-secure proxy re-encryption scheme, outlined in this paper. When a user shares the encrypted content with a trusted party, he provides the re-encryption keys using the public key of the trusted party so that only that party is able to decrypt the content. By facilitating the associated transactions on the blockchain, this scheme provides end-to-end transparency and security for end users to procure storage services at highly competitive prices without worrying about the reputation of the storage providers. We first propose a novel self-encryption (SE) scheme, which is much more efficient than the standard CPA secure El-Gamal encryption scheme. This work is further extended to design a CCA-secure proxy re-encryption scheme (SE-PRE) that adds re-encryption functionality to self-encryption. Prior to our work, the most efficient PRE construction was reported in [13] by Selvi $et\ al$. We show that our PRE design is much more efficient than the scheme in [13].

Proxy Re-encryption (PRE) : Proxy re-encryption is a term coined by Blaze, Bleumer, and Strauss [7] and formalized by Ateniese, Fu, Green, and Hohenberger [3][4]. PRE has been studied extensively for almost two decades [7][3][4][13][12]. A good survey of the PRE schemes and security models of PRE can be found in [11] [8].

2 Preliminaries

In this section we give the definitions of various assumptions adopted for proving the security of the proposed schemes, the general and security model of **SE** and **SE-PRE** schemes.

2.1 Definition

Definition 1. ***Discrete Logarithm Problem (DLP)*** *: The discrete logarithm problem in a cyclic group $\mathbb{G}$ of order q is, given (q, P, Y) such that q is a large prime, P, $Y \in \mathbb{G}$, find $a \in \mathbb{Z}_q^*$ such that $Y = aP$.*

Definition 2. ***Computation Diffie Hellman Problem (CDH)*** *: The Computation Diffie Hellman Problem in a cyclic group $\mathbb{G}$ of order q is, given (q, P, aP, bP) such that q is a large prime, P, aP, $bP \in \mathbb{G}$, find Q such that $Q = abP$, where $a \in \mathbb{Z}_q^*$.*

2.2 Generic Model of Self-Encryption (SE) :

The self encryption (**SE**) is a novel primitive that allows an user to store their files securely with minimal computation overhead. This primitive is different from the traditional public key encryption approach as encryption can be done only by the owner of the file who possess the private key related to the public key which is used for encrypting the file. It has the following algorithms :

1. ***Setup****(κ): This algorithm is run by the trusted entity. On input of a security parameter κ, the **Setup** algorithm will output the system parameters $Params$.*
2. ***KeyGen****(U_i, $Params$): This algorithm is run by the user U_i. This is used to generate a public and private key pair (PK_i, SK_i) for the user U_i.*
3. ***Self-Encrypt****(m, t_w, SK_i, PK_i, $Params$): The encryption algorithm is run only by the user U_i. This algorithm requires the knowledge of the private key SK_i corresponding to the public key PK_i of user U_i. This takes as input the message m, the tag t_w, the private key*

SK_i and public key PK_i of user U_i. It will output a ciphertext C which is the encryption of message m under the public key PK_i and tag t_w. This approach differs from the traditional public key encryption where the encrypt algorithm can be run by any user.

4. ***Self-Decrypt**(C, SK_i, PK_i, $Params$): The decryption algorithm is run by the user U_i. On input of the ciphertext C, the private key SK_i and the public key PK_i of user U_i, this will output the message m if C is a valid self encryption of m under PK_i, SK_i and t_w. Otherwise, it returns $\perp$.*

2.3 Generic Model of Proxy Re-Encryption with Self-Encryption(SE-PRE):

The SE-PRE is a proxy re-encryption primitive that uses a self encryption scheme as the base algorithm and provides a mechanism to delegate the self-encrypted ciphertext. The SE-PRE scheme consists of the following algorithms:

1. ***Setup**(κ):The setup algorithm takes as input a security parameter κ. This will output the system parameters $Params$. This algorithm is run by a trusted party.*
2. ***KeyGen**(U_i, $Params$): The key generation algorithm generates a public and private key pair (PK_i, SK_i) of user U_i. This algorithm is run by a user U_i.*
3. ***ReKeyGen**(SK_i, PK_i, PK_j, c_w, $Params$): The re-encryption key generation algorithm takes as input a private key SK_i of delegator U_i, public key PK_i of delegator U_i, public key PK_j of delegatee U_j and condition c_w under which proxy can re-encrypt. It outputs a re-encryption key $RK_{i \to j}$. This is executed by the user U_i.*
4. ***Self-Encrypt**(m, t_w, SK_i, PK_i, $Params$): The self encryption algorithm takes as input the message m, the tag t_w, the private key SK_i of user U_i and public key PK_i of the user U_i. It outputs a ciphertext C which is the encryption of message m under the public key PK_i, private key SK_i and tag t_w. This algorithm is executed by the user U_i.*
5. ***Re-Encrypt**(C, PK_i, PK_j, c_w, $RK_{i \to j}$, $Params$): The re-encryption algorithm takes as input a self-encrypted ciphertext C, the delegator's public key PK_i, the delegatee's public key PK_j, the condition c_w and a re-encryption key $RK_{i \to j}$ corresponding to c_w. It outputs a ciphertext D which is the encryption of same m under public key PK_j of user U_j. This is run by a proxy who is provided with the re-encryption key $RK_{i \to j}$.*
6. ***Self-Decrypt**(C, SK_i, PK_i, $Params$): The self decryption algorithm is run by the user U_i. This will take as input the ciphertext C, the private key SK_i of user U_i and public key PK_i of user U_i. It will output the message m if C is a valid encryption of m under PK_i and SK_i of user U_i and tag t_w. If C is not valid, this algorithm returns $\perp$.*
7. ***Re-Decrypt**(D, SK_j, $Params$): The re-decryption algorithm takes as input a re-encrypted ciphertext D and a private key SK_j of user U_j. It outputs a message $m \in \mathcal{M}$, if D is a valid re-encrypted ciphertext of message m or the error symbol $\perp$ if D is invalid. This algorithm is run by the user U_j*

2.4 Security Model

In this section we present the security model for the self-encryption scheme and the proxy re-encryption scheme. The security model gives details about the restrictions and oracle accesses given to the adversary. It is modelled as a game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.

2.5 Security Model for Self-Encryption

The security of Self-Encryption (SE) scheme against chosen ciphertext attacks(**IND-SE-CCA**) is demonstrated as a game between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. The game is as follows:

- ***Setup :*** *$\mathcal{C}$ takes a security parameter κ and runs the $Setup(\kappa)$ algorithm to generate the system parameters $Params$. It provides $Params$ to $\mathcal{A}$. $\mathcal{C}$ then runs $KeyGen(U, Params)$ to generate a private and public key pair SK, PK of user U and provides PK to $\mathcal{A}$. SK is kept by $\mathcal{A}$.*
- ***Phase-1 :*** *$\mathcal{A}$ can adaptively issue queries to the following oracles provided by $\mathcal{C}$:*
  - ***Self-Encrypt(m, t_w) Oracle :*** *$\mathcal{C}$ runs the **Self-Encrypt**(m, t_w, SK, PK, $Params$) algorithm to generate the ciphertext C and returns it to $\mathcal{A}$.*
  - ***Self-Decrypt(C, PK) Oracle :*** *$\mathcal{C}$ runs the **Self-Decrypt**(C, SK, PK, $Params$) and returns the output to $\mathcal{A}$.*
- ***Challenge :*** *After getting sufficient training, $\mathcal{A}$ submits two messages m_0, m_1 from $\mathcal{M}$ of equal length and a tag $t_w{}^*$ to $\mathcal{C}$. $\mathcal{C}$ picks a random bit $\delta \in \{0, 1\}$ and outputs the ciphertext C^* =**Self-Encrypt**$(m_\delta, t_w{}^*, SK, PK)$.*
- ***Phase-2 :*** *On receiving the challenge C^*, $\mathcal{A}$ is allowed to access the various oracles provided in **Phase-1** with the restrictions given below :*
  1. ***Self-Decrypt**(C^*) query is not allowed.*
  2. *Self-Encrypt$(m_\delta, t_w{}^*$ query is not allowed.*
- ***Guess :*** *After getting sufficient training, $\mathcal{A}$ will output its guess δ'. $\mathcal{A}$ wins the game if δ=δ'*

2.6 Security Model for Proxy Re-Encryption with Self-Encryption

In this section we provide the security model for the SE-PRE scheme. The model involves the security of original ciphertext as well as transformed ciphertext. The ciphertext that can be re-encrypted is called the original ciphertext and the output of the re-encryption is called the transformed ciphertext.

Security of Original Ciphertext The security of Proxy Re-Encryption with Self-Encryption (SE-PRE) schemes against chosen ciphertext attacks(**IND-SE-PRE-CCA**$_O$) for the original ciphertext is modelled as a game between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. The security game is described below :

- ***Setup :*** *$\mathcal{C}$ takes a security parameter κ and runs the $Setup(\kappa)$ algorithm to generate the system parameters $Params$. The $Params$ is then given to $\mathcal{A}$.*
- ***Phase-1 :*** *On receiving the system parameters, a target public key PK_T and tag $t_w{}^*$, $\mathcal{A}$ is allowed to access **Keygen, Self-Encrypt, Self-Decrypt, Rekey, Re-Encrypt, Re-Decrypt** algorithms. $\mathcal{A}$ simulates the algorithms as oracles and $\mathcal{A}$ can adaptively issue queries to these oracles. The various oracles provided by $\mathcal{C}$ are:*
  - ***Corrupted KeyGen(U_i) :*** *$\mathcal{C}$ runs the $KeyGen(U_i, Params)$ to obtain the public and private key pair (PK_i, SK_i). $\mathcal{C}$ returns both SK_i and PK_i to $\mathcal{A}$.*
  - ***Uncorrupted KeyGen(U_i) :*** *$\mathcal{C}$ runs the **KeyGen**$(U_i, Params)$ to obtain the public and private key pair (PK_i, SK_i) and returns PK_i to $\mathcal{A}$. SK_i is not provided to $\mathcal{A}$.*
  - ***ReKeyGen(U_i, U_j):*** *$\mathcal{C}$ runs the **ReKeyGen**$(SK_i, PK_i, PK_j, c_w, Params)$ to obtain the re-encryption key $RK_{i \to j}$ and returns it to $\mathcal{A}$.*
  - **Self-Encrypt**(m, t_w, PK_i) **:** *$\mathcal{C}$ runs the $Self-Encrypt(m, t_w, SK_i, PK_i, Params)$ to obtain the ciphertext C and returns it to $\mathcal{A}$.*
  - ***Re-Encrypt(C, PK_i, PK_j, c_w) :*** *$\mathcal{C}$ runs the **Re-Encrypt**$(C, PK_i, c_w, RK_{i \to j}, Params)$ to obtain the ciphertext D and returns it to $\mathcal{A}$. Here, $RK_{i \to j}$ is the re-encryption key from PK_i to PK_j under the condition c_w*
  - ***Self-Decrypt(C, PK_i) :*** *$\mathcal{C}$ runs the **Self-Decrypt**$(C, SK_i, PK_i, Params)$ and returns the output to $\mathcal{A}$.*

- ***Re-Decrypt***(D, PK_j)***:*** *$\mathcal{C}$ runs the* ***Re-Decrypt***$(D, SK_j, PK_j, Params)$ *and returns the output to $\mathcal{A}$.*
  For the ***ReKey, Encrypt, Re-Encrypt, Decrypt, Re-Decrypt*** *oracle queries it is required that public keys PK_i and PK_j are generated beforehand.*

– ***Challenge :*** *On getting sufficient training, $\mathcal{A}$ will output two equal-length plaintexts m_0, $m_1 \in \mathcal{M}$. Here, the constraint is: PK_T is generated using* ***Uncorrupted Keygen*** *and* ***Rekey***(PK_T, PK_j, c_w)*, is not queried in* ***Phase-1*** *for $c_w = {t_w}^*$ $\mathcal{C}$ flips a random coin $\delta \in \{0, 1\}$, and sets the challenge ciphertext C^* =* ***Self-Encrypt***$(m_\delta, {t_w}^*, SK_T, PK_T, Params)$*. $\mathcal{C}$ then provide C^* as challenge to $\mathcal{A}$.*

– ***Phase-2 :*** *$\mathcal{A}$ can adaptively query as in $Phase - 1$ with the following restrictions:*
  1. *$\mathcal{A}$ cannot issue* ***Corrupted KeyGen***(U_T) *query.*
  2. *$\mathcal{A}$ cannot issue* ***Self-Decrypt***$(C^*, PK_T, {t_w}^*)$ *query.*
  3. *$\mathcal{A}$ cannot issue* ***Re-Encrypt***(C^*, PK_T, PK_j) *query on C^* from PK_T to PK_j if PK_j is $Corrupted$.*
  4. *$\mathcal{A}$ cannot issue* ***ReKey***(PK_T, PK_j, c_w) *query if c_w=${t_w}^*$.*
  5. *$\mathcal{A}$ cannot issue* ***Re-Decrypt*** *query on D^*, PK_j if D^* is the output of* ***Re-Encrypt***(C^*, PK_T, PK_j, c_w) *and $c_w = {t_w}^*$.*

– ***Guess :*** *Finally, $\mathcal{A}$ outputs a guess $\delta' \in \{0, 1\}$ and wins the game if $\delta' = \delta$.*

Security of Transformed Ciphertext The security of transformed of Proxy Re-Encryption with Self-Encryption(SE-PRE) scheme against chosen ciphertext attacks(**IND-SE-PRE-CCA**$_T$) is modelled as a game between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. This is achieved by:

– ***Setup :****$\mathcal{C}$ takes a security parameter κ and runs the $Setup(\kappa)$ algorithm and gives the resulting system parameters $Params$, a target public key PK_T and tag ${t_w}^*$ to $\mathcal{A}$.*
Phase-1 :*This phase is similar to the* ***Phase-1*** *of* ***IND-SE-PRE-CCA***$_O$*. We do not provide* ***Re-Encrypt*** *oracle as we are providing all the re-encryption keys for the adversary*

– ***Challenge :*** *Once $\mathcal{A}$ decides $Phase - 1$ is over, it outputs two equal-length plaintexts m_0, $m_1 \in \mathcal{M}$. $\mathcal{C}$ flips a random coin $\delta \in \{0, 1\}$, and sets the challenge ciphertext as follows:*
  - *Compute C^* =* ***Self-Encrypt***$(m_\delta, {t_w}^*, SK_i, PK_i, Params)$*, $(PK_i, SK_i$ be the public, private key pair of user U_i and U_i can be honest or corrupt.*
  - *Sets D^* =* ***Re-Encrypt***$(C^*, RK_{i \to T})$ *which is then sent to $\mathcal{A}$.*

– ***Phase-2 :*** *$\mathcal{A}$ adaptively issues queries as in $Phase - 1$, and $\mathcal{C}$ answers them as before with the following restrictions:*
  1. *$\mathcal{A}$ cannot issue* ***Corrupted KeyGen***(U_T) *query.*
  2. *$\mathcal{A}$ cannot issue* ***Re-Decrypt***(D^*, PK_T) *query*

– ***Guess :*** *Finally, $\mathcal{A}$ outputs a guess $\delta' \in \{0, 1\}$ and wins the game if $\delta' = \delta$.*

3 The Self-Encrypt(SE) Scheme :

Self-Encrypt scheme is a special kind of encryption primitive that allows a user to store file securely in cloud or any distributed storage. In this approach the owner of the file uses his/her private key to encrypt the file. This significantly reduces the computation involved in storing the file. We provide the self encryption scheme and the prove its CCA security in the random oracle model.

3.1 The Scheme

The SE scheme consist of the following algorithms:

- **Setup**(κ):
  - Let $\mathbb{G}$ be an additive cyclic group of prime order q. Let P be a generator of group $\mathbb{G}$.
  - Let Δ = $\langle$**Sym.Encrypt**, **Sym.Decrypt**$\rangle$ be any symmetric key encryption scheme. We may assume that Δ is a symmetric key encryption algorithm that uses messages of block size k.
  - Choose the hash functions,

$$H_1 : \{0,1\}^{l_t} \times {\mathbb{Z}_q}^* \rightarrow {\mathbb{Z}_q}^*$$
$$H_2 : {\mathbb{Z}_q}^* \rightarrow \{0,1\}^{l_k}$$
$$H_3 : \{0,1\}^{l_m} \times \mathbb{G} \rightarrow \{0,1\}^{l_3}$$

  - Here l_t is the size of the tag, l_m is the size of the message and l_k is the size of the symmetric key used by the symmetric key encryption scheme Δ. Also, l_3 is dependent on the security parameter κ.
  - Output $Params = \langle q, \mathbb{G}, P, H_1(), H_2(), H_3(), \Delta\rangle$
- **KeyGen**(U, $Params$): The $KeyGen$ algorithm generates the public and private key of the user U by performing the following steps:
  - Choose a random integer $x \xleftarrow{R} {\mathbb{Z}_q}^*$
  - Output $PK = \langle X = xP\rangle$ and $SK = \langle x\rangle$.
- **Self-Encrypt**(m, t_w, SK, PK, $Params$): On input of message m, tag t_w, private key $SK = x$ of user U, public key $PK = xP$ of user U and the public parameters $Params$ this algorithm will generate the self encryption as follows:
  - Choose random $t \in {\mathbb{Z}_q}^*$
  - Set h_t = $H_1(t_w, x)$.
  - Set C_1 = $t + h_t$.
  - Compute Key = $H_2(t)$
  - C_2 = $\{\hat{C}_i\}_{(for\, i=1\, to\, l)}$ and $\hat{C}_i$=Sym.Encrypt (M_i, Key) for all $i = 1$ to l, l is the number of blocks. Assume that $m = M_1\, M_2 \ldots M_l$ where $|M_i|$=k and k is the block size of Δ.
  - $C_3 = H_3(m, t)$
  - Output the ciphertext C = $\langle C_1, C_2, C_3, t_w\rangle$.
- **Self-Decrypt**(C, SK_i, $Params$): Self decryption algorithm is used to decrypt the files that are previously encrypted by the user U using his/her private key. This algorithm does the following:
  - h_t = $H_1(t_w, SK_i)$.
  - t = $C_1 - h_t$
  - Key = $H_2(t)$
  - Compute M_i=**Sym.Decrypt**($\hat{C}_i, Key$) for all i=1 to l and construct $m = M_1\ M_2\ \ldots\ M_l$.
  - If $C_3 \stackrel{?}{=} H_3(m, t)$ then, output m. Else, Output $\perp$.

  Correctness of t:

$$\begin{aligned} RHS &= C_1 - h_t \\ &= (t + h_t) - h_t \\ &= t; \\ &= LHS \end{aligned}$$

3.2 Security Proof:

Theorem 1. *If there exists a (γ, ϵ) adversary $\mathcal{A}$ with an advantage ϵ that can break the **IND-SE-CCA** security of the SE scheme, then $\mathcal{C}$ can solve the discrete log problem with advantage ϵ' where,*

$$\epsilon^{'} \geq \epsilon$$

Proof. In this section we formally prove the security of **SE** scheme in the random oracle model. The **IND-SE-CCA** security of the SE scheme is reduced to the discrete logarithm problem(DLP). The challenger $\mathcal{A}$ is given with the instance of DLP (i.e given (q, P, Y) such that q is a large prime, $P, Y \in \mathbb{G}$, find a such that $Y = aP$.) If there exist an adversary $\mathcal{A}$ that can break the $IND - SE - CCA$ security of the **SE** scheme, then $\mathcal{C}$ can make use of $\mathcal{A}$ to solve the discrete logarithm problem, which is assumed to be hard. Thus the existence of such adversary is not possible.

The challenger $\mathcal{C}$ sets the public key $PK = Y$(PK=aP) and the corresponding private key $SK = x = a$(which is not known to $\mathcal{C}$). $\mathcal{C}$ then provides PK to $\mathcal{A}$. $\mathcal{A}$ has access to various algorithms of **SE** and the hash functions as oracles. $\mathcal{C}$ simulates the hash functions and the **Self-Encrypt, Self-Decrypt** algorithms as described below:

- **Phase-1** : $\mathcal{A}$ is given to access all the oracles as defined in the security model **IND-SE-CCA**. Here it should be noted that $\mathcal{C}$ which does not have the knowledge of private key $SK = a$ provides the functionalities **Self-Encrypt, Self-Decrypt** algorithm.
  - The hash functions involved in the **SE** scheme are simulated as random oracles. To provide consistent output, $\mathcal{C}$ maintains the lists L_{H_1}, L_{H_2} and L_{H_3} corresponding to the hash function H_1, H_2 and H_3 involved in the **SE** scheme.
    - H_1 Oracle: When a query with input (t_w, x) is made, the tuple $\langle t_w, x, h_t \rangle$ is retrieved from L_{H_1} and h_t is returned, if (t_w, x) is already there in L_{H_1} list. Otherwise, $\mathcal{C}$ does the following:
      - If $xP = Y$, then abort. This is because $\mathcal{C}$ obtains the solution to DLP i.e $x = a$.
      - Pick $h_t \in \mathbb{G}$.
      - If h_t is already present in L_{H_1} list, go to previous step.
      - Store $\langle t_w, x, h_t \rangle$ in L_{H_1} list and output h_t.
    - H_2 Oracle: When a query with t is made, $\mathcal{C}$ the tuple $\langle t, Key \rangle$ from LH_2 list is retrieved and will return Key, if (t) is already present in L_{H_2} list. Otherwise, $\mathcal{C}$ does the following:
      - Pick $Key \in \{0,1\}^{l_k}$.
      - If Key is already present in L_{H_2} list, go to previous step.
      - Store $\langle t, Key \rangle$ in L_{H_2} list and return Key.
    - H_3 Oracle: When a query with input (m, T) is made, $\mathcal{C}$ retrieves the tuple $\langle m, T, \alpha \rangle$ from L_{H_3} list and returns α, if (m, T) is already present in L_{H_3} list. Otherwise, $\mathcal{C}$ does the following:
      - Pick $\alpha \in \{0,1\}^{l_3}$.
      - If α is already present in L_{H_3} list, go to previous step.
      - Store $\langle m, T, \alpha \rangle$ in L_{H_3} list and return α.
  - **Self-Encrypt** Oracle : When a **Self-Encrypt** query is made with (m, t_w) as input, $\mathcal{C}$ does the following:
    - Choose random $t \in {\mathbb{Z}_q}^*$
    - Set $h_t = H_1(t_w, x)$.
    - Set $C_1 = t + h_t$.
    - Compute $Key = H_2(t)$
    - $C_2 = \{\hat{C}_i\}_{(for\, i=1\, to\, l)}$ and $\hat{C}_i$=Sym.Encrypt (M_i, Key) for all $i = 1$ to l, l is the number of blocks. Assume that $m = M_1\ M_2\ ...\ M_l$ where $|M_i|$=k and k is the block size of Δ.

* $C_3 = H_3(m,\ t)$
* Output the ciphertext C = $\langle C_1, C_2, C_3, t_w \rangle$.
* Output the self-encrypted ciphertext C to $\mathcal{A}$.

- **Self-Decrypt** Oracle: When a **Self-Decrypt** query is made with $C = \langle C_1,\ C_2,\ C_3,\ t_w \rangle$ as input, $\mathcal{C}$ performs the following:
  * If C is in $L_{Encrypt}$ list, pick m corresponding to C from the tuple $\langle C,\ m \rangle$ in $L_{Encrypt}$ list and output m.
  * If $(t_w, -)$ is present in L_{H_1} list then, retrieve h_t corresponding to $(t_w,\ -)$ from L_{H_1} list. Else, it returns $\perp$
  * $T = C_1 - ht$
  * $Key = H_2(t)$
  * Compute M_i = **Sym.Decrypt**($\hat{C}_i, Key$) for all i=1 to l and construct $m = M_1\ M_2\ ...\ M_l$.
  * If $C_3 \stackrel{?}{=} H_3(m,\ t)$ then, output m. Else, it output $\perp$.

– **Challenge Phase :** After the first phase of training is over, $\mathcal{A}$ provides $m_0,\ m_1 \in \mathcal{M}$, ${t_w}^*$ such that $(m_0, {t_w}^*)$ or $(m_1, {t_w}^*)$ was not queried to **Self-Encrypt** oracle during **Phase-1** and provides to $\mathcal{C}$. $\mathcal{C}$ now generates the challenge ciphertext C^* = **Self-Encrypt**$(m_\delta, {t_w}^*)$ and $\delta\ \in_R\ \{0, 1\}$

– **Phase-2 :** $\mathcal{A}$ can interact with all the oracles as in **Phase-1** but with the following restrictions:

- $\mathcal{A}$ cannot make the query **Self-Decrypt**(C^*)
- $\mathcal{A}$ cannot make the query **Self-Encrypt**$(m_\delta, {t_w}^*)$, $\delta\ \in\ \{0, 1\}$

– **Guess:** Once **Phase-2** is over, $\mathcal{A}$ output its guess δ'. $\mathcal{A}$ wins the game if $\delta = \delta'$.

4 The Proxy Re-Encryption with Self Encryption Scheme(SE-PRE) :

In this section we present a proxy re-encryption scheme which uses the self encryption proposed in **Section 3**. The **SE** scheme is modified such a way that it allows verifiability of ciphertext by proxy during re-encryption without knowing the message. It helps in achieving CCA security of SE-PRE. This also helps in avoiding the DDOS attack being launched on Proxy's service. The proxy is equipped with a method to identify invalid ciphertext so that it will serve its functionality only to valid input. Also.the **SE-PRE** algorithm can be deployed in a simple and efficient manner than using the traditional PRE schemes available till date.

4.1 The Scheme:

In this section we present the proxy re-encryption scheme **SE-PRE** that uses private encryption algorithm. The **SE-PRE** proposed here uses a novel approach, consisting of the following algorithms.

– **Setup**(κ):

- Let $\mathbb{G}$ be an additive cyclic group of prime order q. Let P be a generator of group $\mathbb{G}$.
- Let Δ = $\langle$**Sym.Encrypt**, **Sym.Decrypt**$\rangle$ be any symmetric key encryption scheme. We may assume that Δ is a symmetric encryption algorithm working on block of size k.

- Choose the hash functions,

$$H_0 : \{0,1\}^{l_t} \rightarrow \{0,1\}^{l_0}$$
$$H_1 : \{0,1\}^{l_t} \times {\mathbb{Z}_q}^* \times \mathbb{G} \rightarrow {\mathbb{Z}_q}^*$$
$$H_2 : {\mathbb{Z}_q}^* \rightarrow \{0,1\}^{l_k}$$
$$H_3 : \{0,1\}^{l_m} \times {\mathbb{Z}_q}^* \rightarrow \{0,1\}^{l_3}$$
$$H_4 : {\mathbb{Z}_q}^* \times \{0,1\}^{(l_c+l_3+l_5)} \times \mathbb{G} \rightarrow {\mathbb{Z}_q}^*$$
$$H_5 : \{0,1\}^{l_t} \times {\mathbb{Z}_q}^* \times \mathbb{G} \rightarrow \{0,1\}^{l_5}$$
$$H_6 : \{0,1\}^{l_w} \times \mathbb{G} \times \mathbb{G} \times \mathbb{G} \rightarrow {\mathbb{Z}_q}^*$$
$$H_7 : {\mathbb{Z}_q}^* \times \mathbb{G} \rightarrow {\mathbb{Z}_q}^*$$
$$H_8 : \mathbb{G} \times \mathbb{G} \rightarrow \{0,1\}^{(l_w+l_p)}$$
$$H_9 : \{0,1\}^{l_u} \times {\mathbb{Z}_q}^* \rightarrow {\mathbb{Z}_q}^*$$
$$H_c : \{0,1\}^* \rightarrow \{0,1\}^{l_c}$$

  - Here l_t is size of the tag, l_m is size of the message, l_c is the size of the ciphertext, l_p is κ and l_k is size of the symmetric key used in the encryption scheme Δ. Also, l_ω, l_u, l_0, l_3 and l_5 are dependent on the security parameter κ.
  - Output $Params = \langle q, P, G, P, H_i()_{(for\, i=0\, to\, 9)}, H_c(), \Delta \rangle$

– **KeyGen(U_i, $Params$)**: The $KeyGen$ algorithm generates the public and private key of the user U_i by performing the following:
  - Choose a random integer $x_i \xleftarrow{R} {\mathbb{Z}_q}^*$
  - Output $PK_i = \langle X_i = x_i P \rangle$ and $SK = \langle x_i \rangle$.

– **RekeyGen**(SK_i, PK_i, PK_j, c_w, $Params$) : This algorithm generates the re-encryption key required to translate a ciphertext of user U_i into a ciphertext of user U_j. This is run by the user U_i. The ciphertext to be re-encrypted is encrypted under the public key PK_i of user U_i and with the condition c_w, which are specified by user U_i. This algorithm works as follows:
  - Choose $\omega \xleftarrow{R} \in \{0,1\}^{l_\omega}$
  - Compute $h_c = H_1(c_w, x_i, X_i) \in \mathbb{Z}_q*$
  - Compute $r = H_6(\omega, x_i X_j, X_i, X_j) \in \mathbb{Z}_q*$
  - Compute $s = H_7(r, X_j) \in \mathbb{Z}_q*$
  - Compute $\gamma = rX_j$
  - Compute the re-encryption key $RK_{i \rightarrow j} = \langle R_1, R_2, R_3, R_4, R_5, R_6 \rangle$ where,

$$R_1 = s - h_c \in \mathbb{Z}_q*$$
$$R_2 = rP \in \mathbb{G}$$
$$R_3 = (\omega || X_i) \oplus H_8(\gamma, X_j) \in \{0,1\}^{l_\omega + l_g}$$
$$R_4 = H_6(\omega, \gamma, X_i, X_j) \in \mathbb{Z}_q*$$
$$R_5 = H_5(t_w, x_i, X_i) \in \{0,1\}^{l_5}$$
$$R_6 = H_0(t_w)$$

  - Output the re-encryption key $RK_{i \rightarrow j} = \langle R_1, R_2, R_3, R_4, R_5, R_6 \rangle$

– **Self-Encrypt(m, t_w, SK_i, PK_i, $Params$)**: On input of message m, tag t_w, private key SK_i, public key PK_i of user U_i and the public parameters $Params$
  - Choose random $\omega \in \mathbb{Z}_q*$
  - Set $h_t = H_1(t_w, x_i, X_i) \in \mathbb{Z}_q*$
  - Compute $C_1 = t + h_t$.

- Compute Key = $H_2(t)$
- Compute C_2 = $\{\hat{C}_i\}_{(for\ i=1\ to\ l)}$ and $\hat{C}_i$=**Sym.Encrypt** (M_i, Key) for all $i = 1$ to l, l is the number of blocks. Assume that m = $M_1\ M_2\ ...\ M_l$ where $|M_i|$=k and k is the block size of Δ.
- Set C_3 = $H_3(m,\ t)$
- Find α = $H_5(t_w,\ x_i,\ X_i) \in \{0,1\}^{l_5}$
- C_4 = $H_4(C_1,\ C_2,\ C_3,\ \alpha,\ X)$
- Set C_5 = $H_0(t_w)$
- Output the ciphertext C = $\langle C_1,\ H_c(C_2),\ C_3,\ C_4,\ C_5)\rangle$.

– **Re-Encrypt**($C,\ PK_i,\ PK_j,\ c_w,\ RK_{i\rightarrow j},\ Params$) : This algorithm is run by the proxy which is given with the re-encryption key $RK_{i\rightarrow j}$ by user U_i. This generates the re-encryption of a ciphertext encrypted under public key PK_i of user U_i under the condition c_w into a ciphertext encrypted under public key PK_j of user U_j. This algorithm does not perform any complex computation and this greatly reduces the computational overhead on the entity that performs the role of a proxy. This algorithm does the following computations :

- If $C_4 \neq H_4(C_1,\ H_c(C_2),\ C_3,\ R_5,\ t_w,\ X)$ OR $C_5 \neq R_6$, then it returns $\perp$
- Set D_2 = C_2, D_3 = C_3, D_4 = R_2, D_5 = R_3
- Choose $u \in \{0,\ 1\}^{l_u}$
- Compute β = $H_9(u,\ R_4) \in {\mathbb{Z}_q}^*$
- Compute $D_1\ =\ \beta(C_1 + R_1) \in {\mathbb{Z}_q}^*$
- Set D_6 = u
- Output the re-encrypted ciphertext D = $\langle D_1,\ D_2,\ D_3,\ D_4,\ D_5,\ D_6\rangle$

– **Self-Decrypt**(C**,** SK_i**,** $Params$): Self-Decrypt algorithm is used to decrypt the self-encrypted ciphertext C of a user that is stored by him in the cloud. This algorithm performs the following:

- Find α = $H_5(t_w,\ x_i,\ X_i) \in \{0,1\}^{l_5}$
- If $C_4 \neq H_4(C_1,\ C_2,\ C_3,\ \alpha,\ t_w,\ X)$, then it returns $\perp$
- h_t = $H_1(t_w,\ SK_i)$.
- t = $C_1 - h_t$
- Key = $H_2(t)$
- Compute M_i=**Sym.Decrypt**($\hat{C}_i, Key$) for all i=1 to l and construct $m = M_1\ M_2\ ...\ M_l$.
- If $C_3 \stackrel{?}{=} H_3(m,\ t)$ then, output m. Else, Output $\perp$.

Correctness of t:

$$\begin{aligned} RHS &= C_1 - h_t \\ &= (t + h_t) - h_t \\ &= t \\ &= LHS \end{aligned}$$

– **Re-Decrypt**(D**,** SK_j**,** $Params$): The **Re-Decrypt** algorithm is used to decrypt the re-encrypted ciphertext D. This algorithm does the following:

- Compute γ = $x_j D_4$
- Compute $\omega || X_i$ = $D_5 \oplus H_8(\gamma,\ X_j)$
- Compute r = $H_6(\omega,\ x_j X_i,\ X_i,\ X_j) \in \mathbb{Z}_q*$
- Compute s = $H_7(r,\ X_j) \in \mathbb{Z}_q*$
- ρ = $H_6(\omega,\ \gamma,\ X_i,\ X_j) \in \mathbb{Z}_q*$
- Compute β = $H_9(D_6,\ \rho) \in \mathbb{Z}_q*$
- Compute t = $\beta^{-1}(D_1) - s$
- Find Key = $H_2(t)$
- Compute M_i=**Sym.Decrypt**($\hat{C}_i, Key$) for all i=1 to l and construct $m = M_1\ M_2\ ...\ M_l$.
- If ($C_3 \stackrel{?}{=} H_3(m, t)$) then, output m. Else, it returns $\perp$.

Correctness of T:

$$\begin{aligned} RHS &= \beta^{-1}D_1 - s \\ &= \beta^{-1}[\beta(C_1 + R_1)] - s \\ &= [(t + h_t) + (s - h_c)] - s;\ Here\, h_t = h_c \\ &= (t + s) - s \\ &= t \\ &= LHS \end{aligned}$$

4.2 Security Proof:

In this section, we formally prove the security of **SE-PRE** scheme. We prove the original ciphertext and the transformed ciphertext security in the random oracle model.

Security of the Original Ciphertext

Theorem 2. *If a (γ, ϵ) adversary $\mathcal{A}$ with an advantage ϵ breaks the **IND-SE-PRE-CCA**$_O$ security of the SE-PRE scheme in time γ, then $\mathcal{C}$ can solve the discrete log problem or CDH with advantage ϵ' where,*

$$\epsilon^{'} \geq \frac{1}{q_t}\epsilon$$

Here, q_t is the number of queries to H_6 oracle.

Proof. We formally prove the original ciphertext security **IND-SE-PRE-CCA**$_O$ of **SE-PRE** scheme in the random oracle model. The **IND-SE-PRE-CCA**$_O$ security of the **SE-PRE** scheme is reduced to the discrete logarithm problem(DLP) or the Computational Diffie Hellman Problem (CDH). The challenger $\mathcal{C}$ is given with the instance of CDH $(q, P, Y = aP, Y' = a^2P, Z = bP)$ such that q is a large prime, P, Y, Y' $Z \in \mathbb{G}$. Now, we show that if there exist an adversary $\mathcal{A}$ that can break **IND-SE-PRE-CCA**$_O$ security of the **SE-PRE** scheme then $\mathcal{C}$ can make use of that adversary $\mathcal{A}$ to solve the discrete logarithm problem or the CDH problem, which are assumed to be hard. Hence the existence of such an adversary $\mathcal{A}$ is not possible. Now, adversary $\mathcal{A}$ is provided the access to various **Self-Encrypt, Self-Decrypt, ReKey, Re-Encrypt, Re-Decrypt** algorithms and the hash functions as oracles. $\mathcal{C}$ provides PK_T = Y and ${t_w}^*$ to $\mathcal{A}$. The game is demonstrated below:

- **Phase-1** : $\mathcal{A}$ interacts with $\mathcal{C}$ adhering to the restrictions given in the security model **IND-SE-PRE-CCA**$_O$. $\mathcal{A}$ submits the target condition ${t_w}^*$ to $\mathcal{C}$. $\mathcal{C}$ answers the queries made to various **Corrupted KeyGen, Uncorrupted Keygen, Self-Encrypt, Self-Decrypt, ReKey, Re-Encrypt, Re-Decrypt** oracles and the hash oracles as given below:
  - The hash functions are modelled as random oracles. In order to provide consistent output, $\mathcal{C}$ maintains various lists L_{H_c} and L_{H_i} (for i = 0 to 9) corresponding to the hash function H_c and H_i, (for i = 0 to 9). Oracles H_1, H_2 and H_3 are simulated as given in the proof of **SE** scheme. The other hash oracles are simulated as follows:
    - H_0 Oracle: On input of t_w, $\mathcal{C}$ checks whether a tuple $\langle t_w, h_0 \rangle$ corresponding to t_w is there in L_{H_0} list. If it is already present then return h_0, else pick $h_0 \in \{0,1\}^{l_0}$. Store $\langle t_w,\ h_0 \rangle$ in L_{H_0} list and return h_0.
    - H_c Oracle: On input of $\bar{C}$, $\mathcal{C}$ checks whether a tuple $\langle \bar{C},\ h_c \rangle$ corresponding to $\bar{C}$ is already there in L_{H_c} list. If it is present then h_c is returned, else pick $h_c \in \{0,1\}^{l_c}$. Store $\langle \bar{C},\ h_c \rangle$ in L_{H_c} list and return h_c.
    - H_1 Oracle: When a query with input (t_w, x, X) is made, the tuple $\langle t_w,\ x,\ X\, h_t \rangle$ is retrieved from L_{H_1} and h_t is returned, if (t_w, x, X) is already there in L_{H_1} list. Otherwise, $\mathcal{C}$ does the following:

- If $X = X_i \; in \; L_{Honest}$ and $\bar{x}_i Y = X$, then abort. This is because $\mathcal{C}$ obtains the solution to DLP i.e $\bar{x}_i^{-1} x = a$.
- Pick $h_t \in \mathbb{G}$.
- If h_t is already present in L_{H_1} list, go to previous step.
- Store $\langle t_w, x, X, h_t \rangle$ in L_{H_1} list and output h_t.

* H_4 Oracle: When H_4 oracle is queried with $(C_1, C_2, C_3, \alpha, t_w)$ as input, $\mathcal{C}$ retrieves the tuple $\langle C_1, C_2, C_3, C_4, t_w, h_4 \rangle$ from L_{H_4} list and returns h_4, if an entry corresponding to $(C_1, C_2, C_3, \alpha, t_w)$ is already there in L_{H_4} list. Otherwise, $\mathcal{C}$ does the following:
  - Pick $h_4 \in {\mathbb{Z}_q}^*$.
  - If h_4 is already present in L_{H_4} list, go to previous step.
  - Store $\langle C_1, C_2, C_3, \alpha, t_w, h_4 \rangle$ in L_{H_4} list and return h_4.
* H_5 Oracle: When H_5 oracle is queried with (t_w, x, X) as input, $\mathcal{C}$ retrieves the tuple $\langle t_w, x, X, h_5 \rangle$ from L_{H_5} list and returns β, if an entry corresponding to (t_w, x, X) is already there in L_{H_5} list. Otherwise, $\mathcal{C}$ does the following:
  - Pick $h_5 \in {\mathbb{Z}_q}^*$.
  - If h_5 is already present in L_{H_5} list, go to previous step.
  - Store $\langle t_w, x, X, h_5 \rangle$ in L_{H_5} list and return h_5.
* H_6 Oracle: When H_6 oracle is queried with (ω, γ, X, Y) as input, $\mathcal{C}$ retrieves the tuple $\langle \omega, \gamma, X, Y, h_6 \rangle$ from L_{H_6} list and returns h_6, if an entry corresponding to (ω, γ, X) is there in L_{H_6} list. Otherwise, $\mathcal{C}$ does the following:
  - Pick $h_6 \in {\mathbb{Z}_q}^*$.
  - If h_6 is already present in L_{H_6} list, go to previous step.
  - Store $\langle \omega, \gamma, X, Y, h_6 \rangle$ in L_{H_6} list and return h_6.
* H_7 Oracle: When H_7 oracle is queried with (r, X) as input, $\mathcal{C}$ retrieves the tuple $\langle r, X, h_7 \rangle$ from L_{H_7} list and returns h_7, if an entry corresponding to (r, X) is already there in L_{H_7} list. Otherwise, $\mathcal{C}$ does the following:
  - Pick $h_7 \in {\mathbb{Z}_q}^*$.
  - If h_7 is already present in L_{H_7} list, go to previous step.
  - Store $\langle r, X, h_7 \rangle$ in L_{H_7} list and return h_7.
* H_8 Oracle: When H_8 oracle is queried with (γ, X) as input, $\mathcal{C}$ retrieves the tuple $\langle \gamma, X, h_8 \rangle$ from L_{H_8} list and returns h_8, if an entry corresponding to (γ, X) is already there in L_{H_8} list. Otherwise, $\mathcal{C}$ does the following:
  - Pick $h_8 \in \{0,1\}^{l_w + l_p}$.
  - If h_8 is already present in L_{H_8} list, go to previous step.
  - Store $\langle \gamma, X, h_8 \rangle$ in L_{H_8} list and return h_8.
* H_9 Oracle: When H_9 oracle is queried with (u, r) as input, $\mathcal{C}$ retrieves the tuple $\langle u, r, h_9 \rangle$ from L_{H_9} list and returns h_9, if an entry corresponding to (u, r) is already there in L_{H_9} list. Otherwise, $\mathcal{C}$ does the following:
  - Pick $h_9 \in {\mathbb{Z}_q}^*$.
  - If h_9 is already present in L_{H_9} list, go to previous step.
  - Store $\langle u, r, h_9 \rangle$ in L_{H_9} list and return h_9.

- **Corrupted KeyGen** Oracle: When a query is made with input U_i, $\mathcal{C}$ does the follows:
  * Picks a random $x_i \in {\mathbb{Z}_q}^*$.
  * Computes $X_i = x_i P$
  * Sets $PK_i = \langle X_i \rangle$ and $SK_i = \langle x_i \rangle$.
  * Stores $\langle U_i, x_i, X_i \rangle$ in $L_{Corrupt}$ list and returns (SK_i, PK_i). Here $\mathcal{C}$ knows the private key SK_i of user U_i.
- **Uncorrupted Keygen** Oracle :When a query is made with input U_i, the $\mathcal{C}$ performs the following:
  * Picks a random $\bar{x}_i \in {\mathbb{Z}_q}^*$.
  * Computes $X_i = \bar{x}_i Y = \bar{x}_i a P = x_i P$, where $x_i = \bar{x}_i a$

- Sets $PK_i = \langle X_i \rangle$. By definition $SK_i = \langle x_i = \bar{x}_i a \rangle$. This is not known to $\mathcal{C}$; a is the discrete log corresponding to Y in the DLP which is given as the challenge to $\mathcal{C}$.
- Stores $\langle U_i, -, \bar{x}_i, X_i \rangle$ in L_{Honest} list and returns PK_i.

- **ReKeyGen** Oracle : On input of $(c_w,\ PK_i,\ PK_j)$, $\mathcal{C}$ does the following to generate the re-encryption key:
  - If ($PK_i \in L_{Corrupt}$)
    - Compute $RK_{i \to j}$ = **RekeyGen**(c_w, SK_i, PK_i, PK_j).
    - Store $\langle t_w, PK_i, PK_j, RK_{i \to j} \rangle$ in L_{ReKey} list.
    - Output $RK_{i \to j}$
  - If $t_w,\ PK_i,\ PK_j$ has an entry in L_{ReKey} list, retrieve and return $RK_{i \to j}$
  - If $PK_i \in L_{Honest}$, then
    - Choose $\omega \xleftarrow{R} \in \{0,1\}^{l_\omega}$
    - Choose h_c in $\in \mathbb{Z}_q*$ and store $\langle c_w,\ -,\ X_i \rangle$ in L_{H_1} list.
    - If X_j is corrupt $r = H_6(\omega,\ \bar{x}_i x_j aP,,\ X_i,\ X_j) \in \mathbb{Z}_q*$
    - If $t_w = {t_w}^*$ and $PK_i = PK_T$ then set $rb = H_6(\omega,\ \bar{x}_i a^2 P,\ X_i,\ X_j) \in \mathbb{Z}_q*$ (Here $\mathcal{C}$ does not know rb)
    - Compute $s = H_7(-,\ X_j) \in \mathbb{Z}_q*$
    - By definition $\gamma = rX_j = r\bar{x}_j abP$
    - Compute the re-encryption key $RK_{i \to j} = \langle R_1,\ R_2,\ R_3,\ R_4,\ R_5,\ R_6 \rangle$ where,

$$
\begin{aligned}
R_1 &= s - h_c \in \mathbb{Z}_q* \\
R_2 &= rbP \in \mathbb{G} \\
R_3 &= (\omega || X_i) \oplus H_8(-, X_j) \in \{0,1\}^{l_\omega + l_g} \\
R_4 &= H_6(\omega,\ -,\ X_i,\ X_j) \in \mathbb{Z}_q* \\
R_5 &= h_5 \in \{0,1\}^{l_5}, Store \langle t_w,\ -,\ X_i, h_5 \rangle in L_{H_5} list \\
R_6 &= H_0(t_w)
\end{aligned}
$$

    - Output the re-encryption key $RK_{i \to j} = \langle R_1,\ R_2,\ R_3,\ R_4,\ R_5,\ R_6 \rangle$

- **Self-Encrypt** Oracle : On input of $(m,\ t_w,\ PK_i)$, $\mathcal{C}$ does the following:
  - if PK_i corrupt, then run **Self-Encrypt**$(m,\ t_w,\ SK_i,\ PK_i)$
  - If PK_i is honest, then perform the following
    - Choose random $t \in \mathbb{Z}_q*$
    - Choose $h_t \in \mathbb{Z}_q*$ and store $\langle t_w,\ -,\ X_i,\ h_t \rangle$ in L_{H_1} list.
    - Compute $C_1 = t + h_t$.
    - Compute $Key = H_2(t)$
    - Compute $C_2 = \{\hat{C}_i\}_{(for\ i=1\ to\ l)}$ and $\hat{C}_i$=**Sym.Encrypt** (M_i, Key) for all $i = 1$ to l, l is the number of blocks. Assume that $m = M_1\ M_2\ ...\ M_l$ where $|M_i|$=k and k is the block size of Δ.
    - Set $C_3 = H_3(m,\ t)$
    - Choose $\alpha \in \{0,1\}^{l_5}$ and store $\langle t_w,\ x_i,\ X_i,\ h_5 = \alpha \rangle$ in L_{H_5} list.
    - $C_4 = H_4(C_1,\ C_2,\ C_3,\ \alpha,\ X)$
    - Set $C_5 = H_0(t_w)$
    - Store $\langle C, t_w, PK_i \rangle$ in $L_{Encrypt}$ list.
    - Output the ciphertext $C = \langle C_1,\ H_c(C_2),\ C_3,\ C_4,\ C_5) \rangle$.

- **Re-Encrypt** Oracle : On input $C = \langle C_1,\ C_2,\ C_3,\ C_4,\ C_5,\ PK_i,\ PK_j \rangle$, $\mathcal{C}$ does the following:
  - Generate the re-encryption key $RK_{i \to j}$ = **ReKey**(PK_i, PK_j)
  - Run D = **Re-Encrypt**$(C, RK_{i \to j}, PK_i, PK_j)$ as per the algorithm.
  - Store $\langle C, D, PK_i, PK_j, t_w \rangle$ in $L_{Re-Encrypt}$ list and output D

- **Self-Decrypt** Oracle: On input $C = \langle C_1, C_2, C_3, C_4, C_5, t_w, PK_i \rangle$, $\mathcal{C}$ does the following:
  - If an entry for (C, t_w, X_i) is there in $L_{Encrypt}$ list, return the corresponding m.
  - if PK_i corrupt, then run **Self-Decrypt**(C, SK_i)
  - If PK_i is honest, then perform the following:
    - Find $\langle t_w, x, X, h_5 \rangle$ from L_{H_5} list such that $(X_i = xP)$ OR $(X = X_i$ AND $x = -)$. Set $\alpha = h_5$. If there is no such entry then query $\alpha = H_5(t_w, -, X_i)$.
    - If $C_4 \neq H_4(C_1, C_2, C_3, \alpha, t_w, X)$, then it returns $\perp$ Find $\langle t_w, x, X, h_1 \rangle$ from L_{H_1} list such that $(X_i = xP)$ OR $(X = X_i$ AND x=-). Set $\alpha = h_5$. If there is no such entry query $\alpha = H_5(t_w, -, X_i)$.
    - Find $\langle t_w, x, X, h_1 \rangle$ from L_{H_1} list such that $(X_i = xP)$ OR $(X = X_i$ AND x=-), then set $h_t = h_1$. If there is no such entry query then $h_t = H_1(t_w, -, X_i)$.
    - $t = C_1 - h_t$
    - $Key = H_2(t)$
    - Compute M_i=**Sym.Decrypt**$(\hat{C}_i, Key)$ for all i=1 to l and construct $m = M_1 M_2 ... M_l$.
    - If $C_3 \stackrel{?}{=} H_3(m, t)$ then, output m. Else, Output $\perp$.
- **Re-Decrypt** Oracle : On input of ciphertext D, $\mathcal{C}$ does the following:
  - if PK_j corrupt, then run **Re-Decrypt**(D, SK_j)
  - If PK_i and PK_j are honest, then perform the following
    1. Pick γ, h_8 from $\langle \gamma, X, h_8 \rangle$ L_{H_8} list for $X = X_j$ and
    2. For each γ retrieved from L_{H_8} list: compute $\omega || X_i = D_5 \oplus h_8$ and $r = H_6(\omega, \bar{x}_j \bar{x}_i a^2 P, X_i, X_j)$. If $\gamma = rX_j$, then proceed with next step, else continue checking with other γ retrieved from L_{H_8} list. If no such γ is found, then it returns $\perp$.
    3. Compute $s = H_7(r, X_j) \in \mathbb{Z}_q*$
    4. $\rho = H_6(\omega, \gamma, X_i, X_j) \in \mathbb{Z}_q*$
    5. Compute $\beta = H_9(D_6, \rho) \in \mathbb{Z}_q*$
    6. Compute $t = \beta^{-1}(D_1) - s$
    7. Find $Key = H_2(t)$
    8. Compute M_i=**Sym.Decrypt**$(\hat{C}_i, Key)$ for all i=1 to l and construct $m = M_1 M_2 ... M_l$.
    9. If $(C_3 \stackrel{?}{=} H_3(m, t))$ then, output m. Else, it returns $\perp$.

– **Challenge Phase :** Once the training is over, $\mathcal{A}$ provides $m_0, m_1 \in \mathcal{M}$ such that $(m_0, {t_w}^*)$ or $(m_1, {t_w}^*)$ was not queried to **Self-Encrypt** oracle during **Phase-1** and provides to $\mathcal{C}$. $\mathcal{C}$ generates the challenge ciphertext C^* = Self-Encrypt$(m_\delta, {t_w}^*)$ and $\delta \in_R \{0, 1\}$

– **Phase-2 :** $\mathcal{A}$ interacts with the oracles as in **Phase-1** with the following restrictions,
- $\mathcal{A}$ cannot make the query **Self-Decrypt**(C^*)
- $\mathcal{A}$ cannot make the query **Self-Encrypt**$(m_\delta, {t_w}^*)$, $\delta \in \{0, 1\}$
- $\mathcal{A}$ cannot make the query **Re-Encrypt**(C^*, PK_T, PK_j), $PK_j \in L_{Corrupt}$
- $\mathcal{A}$ cannot make the query **Re-Decrypt**(D, PK_j), if $D = Re{-}Encrypt(C^*, PK_T, PK_j)$ and $PK_j \in L_{Honest}$
- $\mathcal{A}$ cannot make the query **ReKey**$({t_w}^*, PK_T, PK_j)$ and $PK_j \in L_{Corrupt}$

– **Guess:** After getting sufficient training, $\mathcal{A}$ output the guess δ'. $\mathcal{A}$ wins the game if $\delta = \delta'$. $\mathcal{C}$ now picks randomly γ from L_{H_6} list and provides as solution to the CDH problem. □

Security of the Transformed Ciphertext

Theorem 3. *If a (t, ϵ) adversary $\mathcal{A}$ with an advantage ϵ breaks the **IND-SE-PRE-CCA**$_T$ security of the SE-PRE scheme, then $\mathcal{C}$ can solve the Computational Diffie Hellman(CDH) problem with advantage ϵ' where,*

$$\epsilon^{'} \geq \frac{1}{q_t}\epsilon$$

Here, q_t is the number of queries to H_6 oracle.

Proof. In this section we formally prove the security of the transformed ciphertext of **SE-PRE** scheme in the random oracle model. The security of the scheme is reduced to the CDH problem. The challenger $\mathcal{A}$ is given the instance of CDH (i.e given (q, P, aP, bP) such that q is a large prime, $P, aP, bP \in \mathbb{G}$, find Y such that $Y = abP$.) Now, if there exist an adversary who can break the **IND-SE-PRE-CCA**$_T$ security of **SE-PRE** then $\mathcal{C}$ can make use of $\mathcal{A}$ to solve the CDH problem, which is assumed to be hard. Hence such an $\mathcal{A}$ does not exist.

All the oracles are similar to that of **IND-SE-PRE-CCA**$_O$. The challenge ciphertext issued will be a transformed ciphertext. $\mathbb{A}$ has much more access than **IND-SE-PRE-CCA**$_O$:

- $\mathcal{A}$ is allowed to access the rekey from PK_T to any PK_j.
- $\mathcal{A}$ is allowed to access **Self-Decrypt** for PK_T

Challenge : After getting sufficient training $\mathcal{A}$ provides two messages m_0, m_1 and gives to $\mathcal{C}$. $\mathcal{C}$ generates the ciphertext as,

- Generate C^* = **Self-Encrypt**$(m_\delta, {t_w}^*, PK_i)$; $PK_i \in L_{Honest}$
- Generate D^* = **RE-Encrypt**$(C^*, {t_w}^*, PK_i, PK_T)$;
- Provides D^* to $\mathcal{A}$.

In **Phase-2**, $\mathcal{A}$ is now allowed to access all the oracles except **Re-Decrypt**(D^*, PK_T). The rest of the proof is similar to **IND-SE-PRE-CCA**$_O$. □

5 Experimental Analysis

In this section we provide the implementation results and time taken by various algorithms in **SE** and **SE-PRE** scheme. We compare the efficiency of our CCA secure **SE** scheme with the traditional CPA secure El-Gamal scheme (Weaker security than CCA) and report the same in **Table.1**. Also, we have compared our **SE-PRE** scheme with the only non-pairing unidirectional CCA secure PRE scheme by Selvi et al.[13] available. This is reported in **Table 2**. It is a known fact that pairing is very expensive than other group operations and hence we are not taking any pairing based schemes into consideration. The implementations are done on 2.4 GHz Intel Core i7 quad-core processor and the results have been reported below. . The programming language used is Go language [5], and the programming tool is Goland 2018.2. The cryptographic protocols are implemented using the edwards25519-curve [6], which is the current standard deployed in cryptocurrencies [10] for fast performances. From the performance comparison in Table 1, we note that our CCA secure self-encryption **SE** scheme is more efficient than the existing CPA-secure El-Gamal encryption scheme [9]. Also, from Table 2, it is evident that our self-proxy re-encryption **SE-PRE** scheme without bilinear pairing is more efficient than the existing pairing-free PRE scheme by Selvi $et\ al.$ [13].

Algorithm	CPA-Secure El-Gamal Scheme	Our CCA secure **SE** Scheme
Key Generation	612.947	591.677
Encryption	420.307	65.416
Decryption	300.052	41.65

Table 1: Performance Evaluation of the CPA secure El-Gamal encryption scheme and our Self-Encryption Scheme. (All timings reported are in microseconds.)

From the above results it is evident that our **SE** encryption scheme is practical and suitable for cloud based scenarios where the user themselves store their files. Also, the **SE-PRE** scheme provides a very efficient approach to share encrypted files mainly in block-chain.

Algorithm	CCA-Secure Selvi $et\ al.$ Scheme	Our CCA secure **SE-PRE** Scheme
Key Generation	714.271	579.702
First Level Encryption	1044.695	87.85
First Level Decryption	1554.78	60.356
Re-Encryption Key Generation	478.368	796.036
Re-Encryption	1087.52	23.216
Re-Decryption	1077.05	745.031

Table 2: Performance Evaluation of the efficient pairing-free unidirectional PRE scheme due to Chow $et\ al.$ and our Scheme. (All timings reported are in microseconds.)

6 Conclusion

In this paper, we have given a self encryption scheme **SE** based on discrete logarithm (DLP) assumption and then extended it to a Proxy Re-Encryption(**SE-PRE**) scheme suitable for block chain and distributed storage. First, we formally prove the CCA security of the **SE** and then the security of **SE-PRE** scheme in the random oracle model. We have also implemented our **SE-PRE** scheme using GO language. From the results of our implementation, it is evident that our **SE-PRE** scheme is much efficient than the techniques available in literature till date. This makes it more suitable for distributed applications. It will be interesting to see how one can design a multi-recipient or broadcast PRE based on the self encryption approach that will provide high efficiency gain in decentralised platforms.

References

1. 0box application by 0chain :https://0chain.net/zerobox.
2. Ochain website : https://0chain.net.
3. Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.
4. Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM Trans. Inf. Syst. Secur.*, 9(1):1–30, 2006.
5. Daniel J. Bernstein. The go programming language. https://golang.org/.
6. Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, pages 207–228, 2006.
7. Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptograp hy. In *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, pages 127–144, 1998.
8. Isaac Agudo David Nuez and Javier Lopez. A parametric family of attack models for proxy re-encryption. Cryptology ePrint Archive, Report 2016/293, 2016. `https://eprint.iacr.org/2016/293`.
9. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Information Theory*, 31(4):469–472, 1985.
10. Hartwig Mayer. Ecdsa security in bitcoin and ethereum: a research survey. *CoinFaabrik, June*, 28, 2016.
11. David Nuñez, Isaac Agudo, and Javier López. Proxy re-encryption: Analysis of constructions and its application to secure access delegation. *J. Network and Computer Applications*, 87:193–209, 2017.
12. S. Sharmila Deva Selvi, Arinjita Paul, and C. Pandu Rangan. An efficient non-transferable proxy re-encryption scheme. In *Applications and Techniques in Information Security - 8th International Conference, ATIS 2017, Auckland, New Zealand, July 6-7, 2017, Proceedings*, pages 35–47, 2017.
13. S. Sharmila Deva Selvi, Arinjita Paul, and Chandrasekaran Pandu Rangan. A provably-secure unidirectional proxy re-encryption scheme without pairing in the random oracle model. In *CANS*, volume 11261 of *Lecture Notes in Computer Science*, pages 459–469. Springer, 2017.