Forward Private Dynamic Searchable Symmetric Encryption

A Study on Two Existing Schemes

LIU Yicun

ABSTRACT

Dynamic Searchable Symmetric Encryption aims at making possible queries and updates over an encrypted database on an untrusted server, with minimum leakage about user's data to the server. DSSE is called 'dynamic' because it supports 'light-weighted' additions and deletions at the client's side rather than complex reencryption of the entire database for updates.

Recently as the discussion of cloud computing and other online service based on database increases, the security issue of DSSE is widely concerned. As a newly released research addresses, DSSE can leak considerable information under some devastating adaptive attacks that aim at inserting elements which match the previous query. To solve this problem, the concept of forward privacy was proposed.

In this article, we basically consider two schemes of DSSE, both of which achieved forward privacy according to their definition. The first one is *Σοφος – Forward Secure Searchable Encryption* by Bost. He defines forward privacy in top of different leakage patterns and create the Σοφος construction which not only consider to be forward secure, but also supports both addition and deletion. The second one is *Generalizing Forward-Secure Encrypted Search Generic Construction and New Data Structure* by Russell. He further differentiated forward privacy into several conditions, including fully forward privacy and half forward privacy. Moreover, a new data structure named 'Cascaded Triangles' was proposed, with improved updates efficiency and full forward privacy.

KEYWORDS

Dynamic Searchable Symmetric Encryption, DSSE, Forward Privacy, Cascaded Triangles, Σοφος, Construction, Implementation

1. Introduction

1.1 Motivating Problem and Multi-Party Tradeoff

In the age of cloud computing and cloud storage, online database stored at the server's side is a common choice for cloud service infrastructures and their users. When using the

online database, the user always want a solution which he can do queries and update dynamically, with as less leakage of his personal information as possible to the service.

In real world, absolute security with zero is very hard to achieve. In order to improve security, many people tried to build solutions with powerful techniques like multiple-party computation, fully homomorphic encryption, or Oblivious RAM. However, none of these solutions achieves both efficiency and security: they are either computationally expensive or impractical to apply on real case. Most of them are very slow and user often needs to download the entire database and do search locally [Nav15].

As a result, the user want a relatively 'light-weighted' solution: a construction which offers update and query protocols, so that he can apply such protocols without downloading and reencrypting the entire database again. The update should require affordable changes to the former database, which are handled at the server's side. Besides, the solution should only allow a small amount of controlled leakage to the server.

These motivating problems actually run with the tradeoff within efficiency, functionality and security. The way of design a DSSE is actually the way of finding a balanced point with the most wanted solution in the tradeoff.

1.2 Discussion of Functionality

As a secure version of the online database, the main function of DSSE is similar to common databases. It should support protocol of query and update so that the user can well maintain the database.

In order to facilitate discussion, both Bost and Russell use the idea of binary pairs to explain their constructions. In Bost's setting, a tuple of index/keywords set pairs (ind_i, W_i) is introduced, with index ind_i being the index of actual stored documents and keywords W_i being the key to find these documents. In Russell's build, a bipartite graph G with partitions X and Y is introduced. In actual use, X might be the set of keywords and Y might be the set of documents.

As for the query, both of their design runs under a given keyword. In Bost's setting, the query could be $DB(w) = \{(ind_i \mid w \in W\}$, which means finding the indices of the documents which contain specific keyword w. In Russell's setting, because the X and Y bipartite graph is more general, as a result the query can start either from x or y.

Although an agreement is reached in the query part, their design varies on update protocol. Both of them uses edge addition to support adding new mapping manually to the database (which is the common case when setup the database). Bost's deletion is simply edge deletion while Russell uses node deletion to delete all edges connecting to a node (e.g. deleting all documents which contain a specific keyword).

The question of which deletion is better is however under some debate. Edge deletion is considered to be a micro, fine grained modification of existing data. It allows the user to slightly change the inner connection between indices and keywords. However, such fine grained solution often comes with tradeoff. The user has to keep all of the mapping status locally so that he can know which edges need to delete when dealing with massive edge removal. In real world cases, the major application of SSE is for text file storage whose deletion comes with nodes rather than edges. Node deletion is considered to be more practical and convenient.

1.3 Why Forward Privacy

After the idea of DSSE been proposed, many security tests have been run on DSSE based on designed attacks. Recent work by Zhang [ZKP16] reveals a security vulnerability of many DSSE system. The attacker can run a devastating adaptive attack by injecting as few as ten new documents to the system. The attack can be done on almost every DSSE because in the previous setting without requirement of forward privacy, the server can learn whether the newly added documents matches a previous search query.

The problem arises as no forward privacy has been formally considered in almost all previous builds. In fact, the first idea of forward privacy was proposed by Stefanov [SPS14], required such information not to be leaked to the server. However, his proposal is a primitive one without formal explicit definition or notation. And his explanation based on leakage function of query but not update could be problematic. While forward privacy appears to be a very desirable property, not many solutions are known in the literature.

Bost then modified the previous definition by applying leakage function of updates rather than query to define forward privacy. However, the definition from Bost for forward privacy is still limited to a certain type of condition. Russell defines the forward privacy in a more specific way, in total 12 possible kinds of definitions are considered based on six different conditions of updates.

1.4 Data Structure Problem

The design of DSSE data structure is an essential part to realize those discussed functionalities. With query, edge addition, and node deletion (which is based on edge deletion) in mind, design a data structure which achieves both parallelizable operations and traversal efficiency is never an easy work.

Early structure of DSSE doesn't consider much about parallelization, which would lead to low performance in modern paralleled computation environment. The first DSSE proposed by Kamara [KPR12] uses linked list for storage and only supports unparalleled traversal and update. In most practical cases, user often wants a parallelizable solution which supports parallel traversal. The first parallel DSSE scheme utilizes binary search tree as its data structure. However, maintaining this structure for traversal efficiency may involves many complex rebalancing operations and rotations. And letting server to do such operations would lead to many leakage problems.

Consider the binary tree solution, it is thought to be inefficient and insecure because it requires to change too much existing data in the update protocol. If we can minimize the changes to a small number or a constant, there would be much less leakage and the efficiency of update protocol could be well improved. By that thinking, a new data structure 'Cascaded Triangles' was proposed. This structure supports parallel queries and updates, and adding or deleting data only affects a constant amount of existing data. Thanks to cascaded triangles, the new construction features minimal leakage, optimal query and update computation complexity up to a constant factor.

2. General DSSE

2.1 Notations

Let λ be the security parameter, poly(λ) and negl(λ) denotes any polynomial and negligible functions respectively. Let $*$ denotes the wildcard character. $\{0,1\}^n$ denotes the set of n-bit strings while $\{0,1\}^*$ denotes the set of arbitrary long bit strings and $\{0,1\}^\lambda$ denotes the set of λ-bit strings. Φ denotes the empty set. If x is a set, $x \leftarrow X$ (the same as $x \$ \leftarrow X$ in Bost's notation) samples an element x uniformly form X. If A is an algorithm, $x \leftarrow A$ means that x is the output of A, $\oplus$ denotes the bit-wise XOR operation.

2.2 Data Representation

A. In Bost's design, the database is built based on a tuple of index/keyword pairs. Common representation of the database could be:

$$DB = (\text{indi}, \text{Wi})_{i=1}^{p}, \text{ with } \text{indi} \in \{0, 1\}^l \text{ and } \text{Wi} \in \{0, 1\}^*.$$

The keywords of the database DB are denoted as set W, which is a combination of all W_i.

$$W = \bigcup_{i=1}^{D} Wi$$

The number of all documents in the DB is denoted as D, the total number of keywords is denoted as W, and the total number of document/keyword pairs is denoted as N (Noted that we use italic here for the number).

$$D,\ W = |W|,\ N = \sum_{i=1}^{D} |Wi|$$

DB(w) is denoted as the documents set which contains a specific keyword w, which is often used in query representation. In that way, the number of all binary pairs N can also be represented as:

$$N = \sum_{w \in W} |DB(w)|$$

Although Bost himself didn't explicitly defines the representation of the protocols for query, edge addition and edge deletion, but let q denote the query operation and u denote the update operation:

1) **Query**: q = DB(w), which means finding all the documents contains keywords w.
2) **Edge Addition**: u = (add, ind_i , W_i), which means adding an edge connecting document index ind_i and keyword W_i.
3) **Edge Deletion**: u = (del, ind_i , W_i), which means deleting an edge connecting document index ind_i and keyword W_i.

B. In Russell's design, the database is represented in a more general style. Instead of specifying the document index and keyword, bipartite graph contains X and Y was used as a generic representation.

Let X, Y and W be sets where X and Y disjoint, i.e. $X \cap Y = \varnothing$. Let G be a labeled bipartite graph with edges labeled w.

$$\text{i.e. } w \in W$$

Different from Bost's tuple which only contains two elements, tuple treats edge here as its third member. In that way, a tuple can be denoted as:

$$(x, y, w) \in X \times Y \times W$$

By the help of wildcard character $*$, a node y can be denoted as the combination of all the edges connects to node y, such as:

$$(*, y, *) \text{ or similarly } (x, *, *) \text{ for a node x}$$

Let q denote the query operation and u denote the update operation:

1) **Query**: q = $(*, y, *)$ or q = $(x, *, *)$, the query can be start from every node at both side.
2) **Edge Addition**: u = (add, x, y, w), which means adding a new edge w, connecting node x and y.
3) **Node Deletion**: u = (del, x, $*$, $*$) or u = (del, $*$, y, $*$), which is based on edge deletion and deletes all the edges connecting to node x or y.

2.3 DSSE Protocols

Both of the two design have three DSSE protocols: **Setup**, **Search/Query**, and **Update**.

A.

1) Setup(DB) which outputs a pair (EDB, K, σ), which takes unencrypted database DB as input and outputs secret key K, encrypted database EDB, and client's state σ. The setup step is done at the client's side, and then the EDB and σ is outsourced to the server.

2) Search (K, q, σ; EDB) = (Search_C(K, q, σ), Seach_S(EDB)). The client inputs the secret key K, the state σ, and the query q to the server which stores EDB. For single-keyword search scheme, the query q can be substituted as a unique keyword w.

3) Update(K, σ, op, in; EDB) = (Update_C(K, σ, op, in), Update_S(EDB)). The client input its secret key K and state σ, an operation op and an input in parsed as the index ind and a set W of keywords to the server which stores EDB. The server then updates its EDB to the newest state.

B.

1) (K, EDB) ← Setup (1^λ, |X|, |Y|, |W|). The user inputs the security parameter λ, the size |X|, |Y| and |Z| of the spaces. It outputs a key K, and an (initially empty) encrypted database EDB, which is outsourced to the user.

2) (K', R), (EDB', R) ← Qry_e((K,q), EDB). The user inputs the secret key K and a query q. The server has the encrypted database EDB. The user outputs a possibly updated key K', while the server outputs possibly updated encrypted database EDB'. R is the result, outputted by both the user and the server.

   In non-interactive scheme, aextra token generation and passing step is required.
   (K', τ_q) ← QryTkn(K,q) to generate a token τ_q.
   (EDB', R) ← Qry_e(τ_q, EDB) to pass the token from the user to the server so that the server can run the query.

3) (K', EDB') ← Udt_e((K,u), EDB) The user inputs the secret key K and an update u. The server has the encrypted database EDB. The user outputs a possibly key K' while the server outputs a possibly EDB'.

   In non-interactive scheme, extra token generation and pass step is required.
   (K', τ_u) ← UdtTkn(K, u) to generate a update token τ_u.
   EDB' ← Udt_e(τ_u, EDB) to pass the token and runs the update.

2.4 Discussion

Though both A and B provide a general solution of DSSE, **B** differs from (or better than in some situation) **A** in the following perspectives.

Firstly, **B** adapts node deletion rather than edge deletion in **A**. In most practical cases, node deletion is considered be more useful. Not only the user doesn't need to keep the mapping of node and edges locally when deleting, but also less leakage would be concerned because deleting a node rather than several edges would leak less information about the mapping status.

Secondly, **B** supports bi-directional searches while **A**'s search is mostly single-directional. Although in most case query over one of the partitions are already sufficient to naturally capture a wide range of applications as well as these queries uses semi-private data [CK10], compatibility of bi-directional searches would still bring benefits when the data need to be searched from each side (e.g. ID card number and name).

Moreover, **B** considers the iteration of secret keys while **A** doesn't. In the following part with forward privacy involved, it is needed to change secret Key for many times to achieve the requirement of security. In that case, **B** is more favored because a forward private model can directly build on the general DSSE case without much modifications.

3. Forward Privacy

3.1 Preliminaries of Common Leakage

A. Let L denotes as the leakage function which measures the content of leakage.

In Bost's setting, when leakage is limited to search queries, we denote leakage function as $L_q(q)$, which is defined as (q, SearchPattern). When leakage is spread to updates, which means repetition of updated keywords also leaked, we denote leakage function as $L_u(u)$ which is defined as (u, QueryPattern).

More formally, Bost introduces the concept of query list Q, which denotes the historical queires. Q keeps entries represented as (i,w), which means searching keyword w in the i-th query. Or (i,op,in),which means an update with operation op and input in in the i-th update. (i is a timestamp which is initialized to 0).

Based on idea of query list, he further defines the **SearchPattern** and **QueryPattern** as followed:

SearchPattern: sp(x) = {j: (j, x) ∈ Q}, which only match the search queries.
QueryPattern: qp(x) = {j: (j, x) ∈ Q or (j, op, in) ∈ Q and x appears at in} , which means it matches either search queries or search updates.

Bost also introduces the notation **HistDB(w)**, which means the documents historically added to DB which matches the keyword w. Similarly, **UpHist(w)** means the documents historically updated in DB which matches the keyword w.

B. In Russell's design, the idea of dummy queries is provided, so that some of the leakage function can be defined as:

$$\mathcal{L}_u(\mathsf{Add}, x, y, w) = (\tilde{x}, \tilde{\mathcal{L}_u}(\mathsf{Add}, \tilde{x}, y, w)) \text{ for dummy node } \tilde{x} \leftarrow \{0,1\}^\lambda$$
$$\mathcal{L}_u(\mathsf{Del}, x, *, *) = (x, \{\tilde{\mathcal{L}_u}(\mathsf{Del}, \tilde{x}_i, *, *)\}_{i=1}^{c_x}),$$
$$\mathcal{L}_u(\mathsf{Del}, *, y, *) = (y, \tilde{\mathcal{L}_u}(\mathsf{Del}, *, y, *)),$$
$$\mathcal{L}_q(x) = (x, \mathsf{AP}_t(x), \{\tilde{\mathcal{L}_q}(\tilde{x}_i), \tilde{\mathcal{L}_u}(\mathsf{Del}, \tilde{x}_i, *, *)\}_{i=1}^{c_x}),$$

Where x̃i denotes the dummy queries leaked during the addition of edges connecting x after the previous query on x.

3.2 Definitions

The general idea of forward privacy was described as an update does not leak any information about the former updated keywords. Based this idea, different definition of forward privacy is defined.

A. A L-adaptive-secure SSE scheme Σ is forward private if the update leakage function L^{updt} can be written as:

$$\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, \mathsf{in}) = \mathcal{L}'(\mathsf{op}, \{(\mathsf{ind}_i, \mu_i)\})$$

where $\{(ind_i, u_i)\}$ is the set of modified documents paired with the number ui of modified keywords for the updated document ind_i.

Bost's definition can be seen as a slight modified version of the previous arbitrary definition by Stefanov [SPS14]. Bost realized that some previous problem when defining forward privacy based on the added documents rather than updated documents. In fact, the previous forward privacy was captured by leakage function by query rather than leakage function by updates, which utters the initial expectation of forward privacy.

B. In Russell's definition, forward privacy was more precisely defined with multiple conditions of update. At first, according to different connection condition of two edges to be added in the bipartite labelled graph, forward privacy was separated into two cases: **half-forward privacy** and **fully-forward privacy**.

Let DSSE be a (L_q, L_u)-CQA2 secure DSSE scheme for labeled bipartite graphs defined by the spaces X, Y, and W. We say that DSSE is:

1) **Half-forward Private**: if for any u_b =(Add, x_b, y_b,w) where b =0, 1, and $x_0 = x_1$ or $y_0 = y_1$, then for any PPT distinguisher D, it holds that:

$$|\Pr[\mathcal{D}(\mathcal{L}_u(u_0)) = 1] - \Pr[\mathcal{D}(\mathcal{L}_u(u_1)) = 1]| \leq \mathsf{negl}(\lambda)\,.$$

   The inequation above means that the possibility of successfully distinguishing the two updates u_0 and u_1 is less than the negligible function.

2) **Fully-forward Private**: if for any u_b =(Add, x_b, y_b,w) where b =0, 1, then for any PPT distinguisher D, it holds that:

$$|\Pr[\mathcal{D}(\mathcal{L}_u(u_0)) = 1] - \Pr[\mathcal{D}(\mathcal{L}_u(u_1)) = 1]| \leq \mathsf{negl}(\lambda)\,.$$

   The inequation above means that the possibility of successfully distinguishing the two updates u_0 and u_1 is less than the negligible function.

The main idea of this differentiation is whether hide one end of edge or hide both ends. Half forward privacy only hides one end of the connection (x, y) while fully-forward privacy hides both ends of the connection, which makes attacks aiming at specific leakage harder to succeed.

Recall that half-forward privacy needs to satisfy the condition of $x_0 = x_1$ or $y_0 = y_1$, so if we implement two updates of adding edges, there would be a high possibility that the two new edges are connecting to the shared node. Such vulnerability can be caught by attacks which analyze the update rate of different nodes. In that way, maintaining half-forward privacy may be not an ideal secure measurement.

However, half-forward privacy and fully-forward privacy may not be the whole picture. Later an exhaustive version of forward privacy was proposed, this time with six different cases of update U_i with or without restriction pi (if without, p_i is simply set to be 1):

i	Sets of updates $\mathcal{U}_i$	Possible conditions such that $p_i = 1$ (If there are no restrictions, $p_i \equiv 1$)
1	$\{(\mathsf{Add}, x, y, w)\}$	$x_0 = x_1$ or $y_0 = y_1$
2	$\{(\mathsf{Add}, \mathrm{x}, y, \mathrm{w})\}$	$y_0 = y_1$
3	$\{(\mathsf{Add}, x, \mathrm{y}, \mathrm{w})\}$	$x_0 = x_1$
4	$\{(\mathsf{Del}, x, y, *)\}$	$x_0 = x_1$ or $y_0 = y_1$
5	$\{(\mathsf{Del}, x, *, *)\}$	$\|(x_0, *, *)\| = \|(x_1, *, *)\|$
6	$\{(\mathsf{Del}, *, y, *)\}$	$\|(*, y_0, *)\| = \|(*, y_1, *)\|$

Now, not only do we consider the shared node condition for two edge additions, but also for deletions, too (here means deleting the same node as condition 5 or 6 states).

The previous forward privacy in **A** could be represented by ($U_1 \cup U_4$, 1)-forward privacy. And the half-forward privacy and fully-forward privacy in **B** could be the case U_1, with and without restrictions.

3.3 The Need and Constraints

The study of real world consequence of SSE scheme leakage addresses the urgency of defining forward privacy too. Islam [IKK12] and Cash [CGPR15] shows that specific file injection to the server could cause considerable leakage, for both static and dynamic SSE schemes. Zhang then improves the file injection attack [ZKP16] and introduces the non-adaptive and adaptive attack.

In the adaptive attack, which is consider to be very efficient for non-forward-private scheme, the attacker could reveal a previously searched keyword w by submitting log2T new documents if he has partial knowledge of the database, or W/T + logT new documents if he doesn't. In a practical test run by Zhang, if T = 200, leakage for query pattern could be revealed just after submitting less than 10 new documents.

However, though forward privacy is believed to be a significant improvement in security, it also brings downside in terms of efficiency.

Constraints on Storage

Because the previous design of deletion by Bost is actually a 'lazy deletion'. In order to avoid possible leakage when reclaiming space, the deleted entry is often marked as 'empty' rather than actually being deleted from storage. If some cases in actual need frequently deletion, 'lazy deletion' may not be a good idea because it won't release the storage resources taken by the deleted edges.

Constraints on Locality

In dynamic scheme, there always exists tradeoff between the locality and forward privacy. Firstly, with previous build like ORAM which needs frequent access to the disk and slow down the process, the discussion of memory locality may not be worth the effort because it bottleneck is at the disk. Secondly, the initial of forward privacy which requires new updates unrelated to existing contents makes locality come with high price. Somehow to achieve both forward privacy and efficiency large modification of the EDB needs to be done in searches and updates, and it is commonly considered to be not worth it. Finally, with the help of modern SSD, Bost claims that the evaluation of his implementation can also be done without locality.

3.4 Discussion

Both **A** and **B** realize that it could be problematic in previous definition when defining forward privacy based on the added documents rather than updated documents. And both of them defines forward privacy based on leakage function on update rather than leakage function on query, which suits the initial idea of forward privacy based on updates.

However, in detailed definition, **B** gives a more comprehensive specification than **A**. With six different updates and their restriction in mind, we can define up to 12 kinds of forward privacy, and the forward privacy of A is one of them. Although not every case will be practically used and in fact only the half-forward privacy and fully-forward privacy are implemented latter in **B**, giving a whole picture of the definition is somehow meaning.

Another major difference of **A** and **B** it the way they treat deletion. In order to be not distinguishable by some adversaries when reclaiming the storage marked deleted, **A** uses the idea of '**lazy deletion**' which only marks the deleted entry as empty and simply gives up the idea of reclamation, trading storage (or efficiency) for security. This measurement needs more storage and may suffers significant efficiency loss in some real cases requires frequently deletions. On the contrary, **B** thinks that the forward privacy only for addition is sufficient in real cases and chooses to achieve higher efficiency in deletion, which reclaims the storage after deletion.

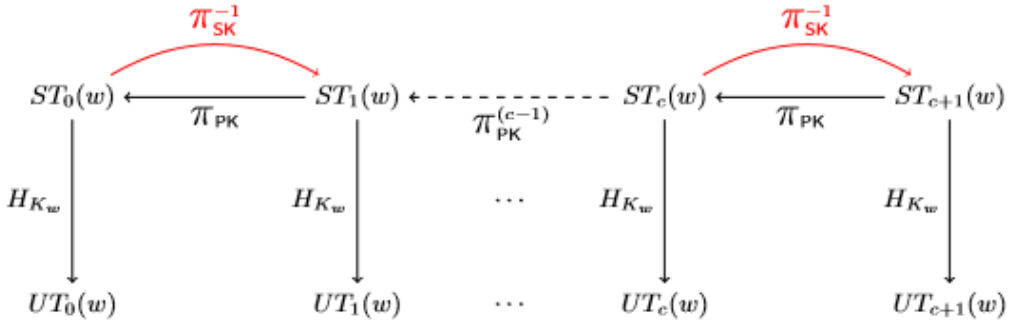
In my understanding, maybe there exists some other solutions. We might use some techniques to make storage reclamation 'not easily distinguishable', which could reduce the loss of efficiency and achieve forward privacy in deletion. For instance, delay the storage reclamation after a pseudo random time after marked 'empty' may fool the adversary?

4. Forward Private DSSE

4.1 Preliminaries

A. The key idea of Bost's design is based on trapdoor permutation.

A trapdoor permutations is a function that is easy to computer in one direction, yet difficuilt to compute in the opposite direction without special information. For instance, RSA is trapdoor permutation which is easy to compute with private key but hard to compute without the knowledge of private key, and the 'trapdoor' here is the special information private key.



Here, $UT_c(w)$ denotes the location to store the indices for a specific keyword w. In that location, a indexed list of documents matching keywords w was stored like (ind_0, ..., ind_n). And $UT_{c+1}(w)$ denotes adding a new index for a new document matching keywords w.

When the client wants to perform a search on query w, he will issue a search token that allows the server to recompute the update token and location of the entries matching w. In general, we want the update token for a given w to be unlinkable until a search token $ST_c(w)$ is issued. In the figure shown above, we want the search token $ST_c(w)$ to be unrelated with the update token $UT_i(w)$ when $i > c$.

By the help of trapdoor permutation, the server will be able to compute $ST_{i-1}(w)$ from $ST_i(w)$ using a public key. And the client is the only one to be able to construct $ST_{i+1}(w)$ using the secret key from himself.

B. In Russell's design, the forward private DSSE scheme is denoted as ε. Each keyword keeps a table of PRF keys K_x and a counter c_x. A dictionary γ^Q maps the query on keywords $q = x$ to a PRF key K_x and the counter c_x.

To improve efficiency, when a query result is open and the plaintext result is returned, we would not consider the plaintext to be worth protected anymore and simply throw it to somewhere specially keeps the queried plaintext. In that way, a plaintext graph $\widehat{G}$ and a ciphertext graph G is introduced.

The main idea of the design is to achieve forward privacy by locally keep the dictionary γ^Q.

4.2 Constructions of Forward Private DSSE

A. The Σοφος Construction

In the setup step, SK and PK are generated by KeyGen algorithm and W and T was

created as two empty map. Then it outsources the (T, PK) to the server and (K_S,SK) with W to the client.

Setup()

$K_S \xleftarrow{\$} \{0,1\}^\lambda$
$(\mathsf{SK}, \mathsf{PK}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$\mathbf{W}, \mathbf{T} \leftarrow$ empty map
return $((\mathbf{T}, \mathsf{PK}), (\mathbf{K_S}, \mathsf{SK}), \mathbf{W})$

In the search (query) protocol based on query q = w at the client's side, the client first use the PRF function F to generate a one-time key K_w, then the client checks whether the index w of his table W contains the search token ST_c and the counter c. And set c = n_w − 1. If q = w has never been searched in history, which means $(ST_c, c) = \perp$, the client returns ∅. In the end, the client sends one-time key K_w, search token ST_c (if any) and counter c (if any) to the server.

Client:
$K_w \leftarrow F_{K_S}(w)$
$(ST_c, c) \leftarrow \mathbf{W}[w]$ ▷ $c = n_w - 1$
if $(ST_c, c) = \perp$
  return $\emptyset$
Send (K_w, ST_c, c) to the server.

At the server side, the server uses the one time key K_w, and the previous search token ST_i as an input to a hash function H_1 to generate the Update token UT_i respectively. In this step, each search token ST_i with i <= c is obtained by the public key PK. Use the update token UT_i, we can check whether it has been updated in table T, returns the encrypted index e. By XOR operation with result of hash function H_2, the unencrypted version ind is found and returned to the user.

Server:
for $i = c$ **to** 0 **do**
  $UT_i \leftarrow H_1(K_w, ST_i)$
  $e \leftarrow \mathbf{T}[UT_i]$
  $\mathsf{ind} \leftarrow e \oplus H_2(K_w, ST_i)$
  Output each ind
  $ST_{i-1} \leftarrow \pi_{\mathsf{PK}}(ST_i)$
end for

As for update, we only consider addition in this part. When the client needs to add a new edge connecting w and ind, he needs to first generate the one time key by PRF function F. Then he checks the table W for previous searched token ST_c and counter c, if none, he generates the new search token ST_0 and set c to be -1. If any, he uses his secret key SK to generate the new search token ST_{c+1} and updates (ST_{c+1}, c+1) at W[w]. Then he uses two hash function to get the update token UT_{c+1} for c+1 and the encrypted index e by second hash with XOR operation, then outsources them to the

server.

Client:

$K_w \leftarrow F(K_S, w)$
$(ST_c, c) \leftarrow \mathbf{W}[w]$
if $(ST_c, c) = \perp$ **then**
  $ST_0 \xleftarrow{\$} \mathcal{M}$, $c \leftarrow -1$
else
  $ST_{c+1} \leftarrow \pi_{\mathsf{SK}}^{-1}(ST_c)$
end if
$\mathbf{W}[w] \leftarrow (ST_{c+1}, c+1)$
$UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
$e \leftarrow \mathsf{ind} \oplus H_2(K_w, ST_{c+1})$
Send (UT_{c+1}, e) to the server.

The server' s task in addition is very simple, it just uses encrypted index e to update $T[UT_{c+1}]$.

Server:

$\mathbf{T}[UT_{c+1}] \leftarrow e$

B. Forward Privacy for Any DSSE

In the setup step of Russell's forward private DSSE, the user inputs a security parameter λ, which generate the key $\widetilde{K}$ and the initial $\widetilde{\mathrm{EDB}}$. The dictionary γ^Q and plaintext graph $\widehat{G}$ is set to empty. The the key $\widetilde{K}$ and the dictionary γ^Q is given to the client and the initial $\widetilde{\mathrm{EDB}}$ and the empty plaintext graph $\widehat{G}$ is outsourced to the server.

$(K, \mathsf{EDB}) \leftarrow \mathsf{Setup}(1^\lambda)$

$(\tilde{K}, \tilde{\mathsf{EDB}}) \leftarrow \mathcal{E}.\mathsf{Setup}(1^\lambda)$
$\gamma^Q = \phi, \hat{G} = \phi$
return $K = (\tilde{K}, \gamma^Q), \mathsf{EDB} = (\tilde{\mathsf{EDB}}, \hat{G})$

In the addition protocol at the client's side, u = (Add, x, y, w) is sent by the client, means adding a new edge (x, y). Based on the different condition on node x or y, there are two cases to be concerned. The first case is node x is a singleton, which means there are no edges connecting to node x before, thus, we simply generate the key by the PRF function and set counter as 1 and updates $\gamma^Q[x]$. The other case is that node x is not a singleton, thus we simply add 1 to the counter and then update$\gamma^Q[x]$. In the last, an update token is generated to be outsourced to the server.

$(K', \tau_u^+) \leftarrow \mathsf{UpdateToken}(K, u = (\mathsf{Add}, x, y, w))$

1 : **if** $\gamma^{\mathcal{Q}}[x] = \perp$ **then**
2 : $\quad K_x \leftarrow \{0,1\}^\lambda$
3 : $\quad c_x \leftarrow 1$
4 : **else**
5 : $\quad (K_x, c_x) \leftarrow \gamma^{\mathcal{Q}}[x]$
6 : $\quad c_x \leftarrow c_x + 1$
7 : **endif**
8 : $\gamma^{\mathcal{Q}}[x] \leftarrow (K_x, c_x)$
9 : $\tau_u^+ \leftarrow \mathcal{E}.\mathsf{UpdateToken}(\tilde{K}, (\mathsf{Add}, F(K_x, c_x), y, w))$
10 : **return** (K, τ_u^+)

At the server's side, the server simply updates the EDB by the received update token.

1 : $\tilde{\mathsf{EDB}} \leftarrow \mathcal{E}.\mathsf{Update}(\tilde{\tau}_u^+, \tilde{\mathsf{EDB}})$
2 : **return** EDB

The deletion protocol at the client side is similar to the addition one. The main difference is now we delete node rather than edges, which means we have more edges to deal with. The update token for node deletion here is actually the combination of all the update tokens for the edges connecting to the node. We first retrieve the table to get tokens of edge deletions and then 'encapsulate' the edge deletion tokens into one node deletion tokens. Noted that though edge deletion is actually involved in the whole process, but the client only need to tell the server which node to delete and the server first retrieves all the edges and then delete them, so the efficiency and leakage problem we concerned before can be gone.

1 : **if** $u = (\mathsf{Del}, x, *, *)$ **then**
2 : $\quad$ **if** $\gamma^{\mathcal{Q}}[x] \neq \perp$ **then**
3 : $\quad\quad (K_x, c_x) \leftarrow \gamma^{\mathcal{Q}}[x]$
4 : $\quad\quad \gamma^{\mathcal{Q}}[x] \leftarrow \perp$
5 : $\quad\quad$ **for** $i = 1, \ldots, c_x$ **do**
6 : $\quad\quad\quad \tilde{\tau}_i^- \leftarrow \mathcal{E}.\mathsf{UpdateToken}(\tilde{K}, (\mathsf{Del}, F(K_x, i), *, *))$
7 : $\quad\quad$ **endfor**
8 : $\quad\quad \tau_u^- \leftarrow (x, \{\tilde{\tau}_i^-\}_{i=1}^{c_x})$
9 : $\quad$ **else**
10 : $\quad\quad \tau_u^- \leftarrow x$
11 : $\quad$ **endif**

The server's side operation is based on the deletion token received and then the server

deletes all the edges in encrypted version respectively.

$\mathsf{EDB}' \leftarrow \mathsf{Update}_e(\tau_u^-, \mathsf{EDB})$ for $u = (\mathsf{Del}, \cdot, \cdot, \cdot)$

1 : **if** $\tau_u^- = (x \in \mathcal{X}, \{\tilde{\tau}_i^-\}_{i=1}^{c_x})$ **then**
2 : $\quad \hat{G} \leftarrow \mathsf{Update}((\mathsf{Del}, x, *, *), \hat{G})$
3 : $\quad$ **for** $i = 1, \ldots, c_x$ **do**
4 : $\quad\quad \tilde{\mathsf{EDB}} \leftarrow \mathcal{E}.\mathsf{Update}_e(\tau_i^-, \tilde{\mathsf{EDB}})$
5 : $\quad$ **endfor**

The search part at the client's side is basically two steps: retrieve the table for query x (if any) and generate the delete token for node x.

$(K', \tau_q) \leftarrow \mathsf{QueryToken}(K, q)$

1 : **if** $q = x \in \mathcal{X} \ \wedge \ \gamma^{\mathcal{Q}}[x] \neq \perp$ **then**
2 : $\quad (K_x, c_x) \leftarrow \gamma^{\mathcal{Q}}[x]$
3 : $\quad \gamma^{\mathcal{Q}}[x] \leftarrow \perp$
4 : $\quad$ **for** $i = 1, \ldots, c_x$ **do**
5 : $\quad\quad \tilde{\tau}_i \leftarrow \mathcal{E}.\mathsf{QueryToken}(\tilde{K}, F(K_x, i))$
6 : $\quad\quad \tilde{\tau}_i^- \leftarrow \mathcal{E}.\mathsf{UpdateToken}(\tilde{K}, (\mathsf{Del}, F(K_x, i), *, *))$
7 : $\quad$ **endfor**
8 : $\quad \tau_q \leftarrow (x, \{\tilde{\tau}_i, \tilde{\tau}_i^-\}_{i=1}^{c_x})$

The search part at the server's side not only delete the edges connecting to node x, but also return the plaintext result and throw it the plaintext graph $\widehat{\mathsf{G}}$ for better efficiency.

1 : Parse τ_q as $(x, \{\tilde{\tau}_i, \tilde{\tau}_i^-\}_{i=1}^{c_x})$
2 : $R \leftarrow \mathsf{Query}(x, \hat{G})$
3 : **for** $i = 1, \ldots, c_x$ **do**
4 : $\quad \tilde{R} \leftarrow \mathcal{E}.\mathsf{Query}_e(\tilde{\tau}_i, \tilde{\mathsf{EDB}})$
5 : $\quad \tilde{\mathsf{EDB}} \leftarrow \mathcal{E}.\mathsf{Update}_e(\tilde{\tau}_i^-, \tilde{\mathsf{EDB}})$
6 : $\quad R \leftarrow R \cup \tilde{R}$
7 : **endfor**
8 : **foreach** $(\tilde{x}, y, w) \in R$ **do**
9 : $\quad R \leftarrow R \setminus \{(\tilde{x}, y, w)\} \cup \{(x, y, w)\}$
10 : $\quad \hat{G} \leftarrow \mathsf{Update}((\mathsf{Add}, x, y, w), \hat{G})$
11 : **endfor**
12 : **return** (EDB, R)

4.3 Discussion

Except the different ways of handling deletion as we discussed before, the two DSSE schemes is proved to be forward private by different approaches.

A uses the concept of trapdoor permutation and two table to store the historical search tokens and update tokens at client's and server's side, respectively. The server is provided with PK but no SK to ensure it cannot deduct the token in forward cases.

B's key idea is to maintain the table which contains the historical queries locally. And the server's manner is strictly controlled by the pass of tokens.

Both **A** and **B** tries to use some approaches for better efficiency. **A** uses some techniques in asymmetric encryption like Chinese Reminder Theory to reduce the additional computational expenses and release some stress on local storage caused by 'lazy deletion'. **B** uses a plaintext graph for the queried result and deletes the encrypted result from EDB after query, which also improves the efficiency.

5. New Data Structure

5.1 Overview and Notations

The simplified idea of cascaded triangles is to divide the previous one large binary tree into many cascaded small binary trees, named triangles.

The height of the cascaded triangles increases from the first to the last, with only the first two triangles could be of equivalent height.

$$h_1 \leq h_2 < h_3 < \ldots < h_k, \quad h \in \{0,1,2\}^{\lceil \lg n_q \rceil}$$

The advantage of division is that in each time of update, only small part of the whole data structure will be affected (mostly one or two triangles).

In the construction of cascaded triangles, three dictionaries are introduced as γQ, γD, and $\gamma\sigma$.
γQ [x] = ($addr_1$, ..., $addr_k$, h), which stores the root addresses of each triangle and their height.
γD [$\overline{addr}$] = addr, which is used to store the addresses of duals.
$\gamma\sigma$ [addr] = (a, b), with a = (x, y, w) being the edge stored in this node and b = (ch_1, ch_2) being the address of their child.

The first dictionary is stored at the client side, and the latter two dictionaries is stored at the server's side.

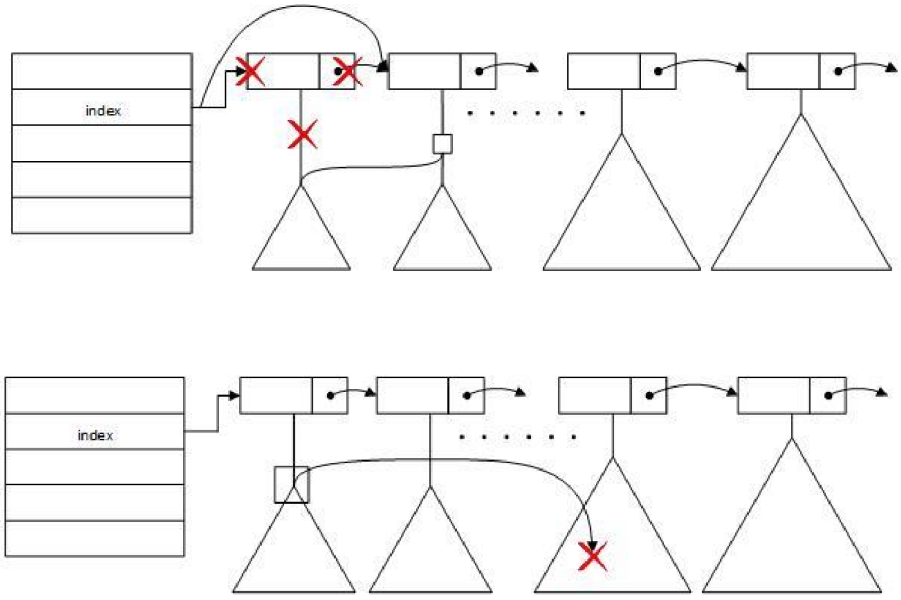
5.2 Plaintext Version of Cascaded Triangles

In the plaintext version of cascaded triangles. Query on x needs the client to traverse γQ [x] = ($addr_1$, ..., $addr_k$, h) locally. Once the root node of each triangle is obtained, the traversal can be continued with the help of the children node addresses stored in $\gamma\sigma$ [addr] = (a, b) at the server.

When dealing with update, keep in mind that the requirement on heights of the triangles should be maintained. In that way, division add combination of triangles would appear in deletion and addition.

In the addition part, we simply add the new node as a root of a new triangle, this triangle

has only one node, which is the root node. If there are another triangle of the same height, combine them into one new triangle.



In the deletion part, if we want to delete node x, Firstly, we traverse the set (x, *, *) using the traverse algorithm. Delete all the traverse nodes, based on the address of the root node. Then, use traverse result of x to find the duals in triangles based on y. Replace these nodes (in the middle of some triangles) with the smallest triangles.

- Lookup $\gamma^{\mathcal{D}}[\overline{\mathsf{addr}}_1] = \mathsf{addr}_1$.
- Delete $\gamma^{\mathcal{D}}[\mathsf{addr}]$ and $\gamma^{\mathcal{D}}[\overline{\mathsf{addr}}_1]$, and update $\gamma^{\mathcal{D}}[\mathsf{addr}_1] \leftarrow \overline{\mathsf{addr}}$ and $\gamma^{\mathcal{D}}[\overline{\mathsf{addr}}] \leftarrow \mathsf{addr}_1$.
- Lookup $\gamma^{\delta}[\overline{\mathsf{addr}}_1] = (\overline{a}_1, \overline{b}_1)$, where $\overline{b}_1 = (\overline{\mathsf{addr}}'_0, \overline{\mathsf{addr}}'_1)$.
- Update $\gamma^{\delta}[\overline{\mathsf{addr}}] \leftarrow (\overline{a}_1, \overline{b})$ and delete $\gamma^{\delta}[\overline{\mathsf{addr}}_1]$.
- Update $\gamma^{\mathcal{Q}}[y] = (\overline{\mathsf{addr}}'_0, \overline{\mathsf{addr}}'_1, \overline{\mathsf{addr}}_2, \ldots, \overline{\mathsf{addr}}_k, \overline{h} - 1)$, where $h - 1$ is the reverse process of $h + 1$ defined above.

5.3 Ciphertext Version of Cascaded Triangles

The main idea of the ciphertext version is to split G into two graphs, one is the plaintext graph and one is the ciphertext graph, they are stored at the client's side and server side respectively.

Now we only need to store the ciphertext in tuples of $\gamma\sigma$ [addr]. Two secret keys are used to access a = (x, y, w) and b = (ch_1, ch_2) respectively. Plus, the γQ [x] now additionally

stores the two secret keys associated to x.

The encrypted version is similar to the plaintext version, though with more detailed steps handling the plaintext graph and the ciphertext graph.

Traversal and Query

When retrieving the root address locally, the two secret keys are also retrieved. The client sends the result of the retrieval back to the server. The server uses the first key to get part of the query result. And then uses the second key to traverse the sub triangles. The server also returns the previous result in plaintext graph.

Addition

Now the clients send encrypted version of (a, b) denoted as (c_a, c^b), which is encrypted under two secret keys correspondingly retrieved by γQ [x]. When the γQ [x] is empty, the user need to generate two new secret keys for x.

Deletion

The step for deletion shares many identical features with traversal. But here the client will not send a_k. Instead, the server returns all c_a so the client can decrypt them locally. After acknowledging which edges to delete, the client and the server delete all edges connecting to node x, the ciphertext or the plaintext. As a result, the node x can be removed from both the ciphertext graph kept at the server's side and the plaintext graph kept at the client's side.