An extended abstract of this paper appears at ACNS 2017.

Forward-Secure Searchable Encryption on Labeled Bipartite Graphs*

(Full Version)

Russell W. F. Lai and Sherman S. M. Chow**

Department of Information Engineering,
The Chinese University of Hong Kong,
Shatin, N.T., Hong Kong
{russell, sherman}@ie.cuhk.edu.hk

July 4, 2017

Abstract. Forward privacy is a trending security notion of dynamic searchable symmetric encryption (DSSE). It guarantees the privacy of newly added data against the server who has knowledge of previous queries. The notion was very recently formalized by Bost (CCS '16) independently, yet the definition given is imprecise to capture how forward secure a scheme is. We further the study of forward privacy by proposing a generalized definition parametrized by a set of updates and restrictions on them. We then construct two forward private DSSE schemes over labeled bipartite graphs, as a generalization of those supporting keyword search over text files. The first is a generic construction from any DSSE, and the other is a concrete construction from scratch. For the latter, we designed a novel data structure called cascaded triangles, in which traversals can be performed in parallel while updates only affect the local regions around the updated nodes. Besides neighbor queries, our schemes support flexible edge additions and intelligent node deletions: The server can delete all edges connected to a given node, without having the client specify all the edges.

1 Introduction

In searchable symmetric encryption (SSE), an encrypted database can be queried with minimal leakage of information about the plaintext database to the hosting server. The client is additionally allowed to update the encrypted database in dynamic SSE (DSSE) without reencrypting from scratch. Since its introduction [SWP00], many SSE schemes with different trade-offs between efficiency, security, and query expressiveness have been proposed [BHJP14]. Most earlier schemes were not dynamic. The first sublinear dynamic SSE scheme was proposed by Kamara *et al.* [KPR12], but the query and update operations are inherently sequential. Some later schemes [KP13, SPS14, HK14] feature parallelizable algorithms for queries and updates. Parallelism made DSSE an attractive solution for outsourcing data to cloud platform which fully leverages the multiprocessors.

1.1 Security and Forward Privacy of SSE Schemes

Ideally, the knowledge of an encrypted database, together with a sequence of adaptively issued queries and updates, should not reveal any information about the plaintext database and the query results to the server. Although this can be achieved theoretically through techniques involving obfuscation [CCC+16] or oblivious RAM [GMP16], the resulting solutions are not particularly efficient. Typically, a practical DSSE scheme tolerates the leakage of search and access patterns during queries, and some internal structure of the encrypted

* Sherman Chow is supported in part by General Research Fund (Grant No. 14201914) and the Early Career Award from Research Grants Council, Hong Kong; and Huawei Innovation Research Program (HIRP) 2015 (Project No. YB2015110147).

** Corresponding Author

database during updates. Formally, the security is parametrized by a set of leakage functions describing these leakages. While some leakages seem to pose no harm, some have been exploited in attacks [IKK12, ZKP16].

Forward privacy, advocated by Stefanov *et al.* [SPS14], requires that newly added data remains private against the server, who has knowledge about previous queries. The property is arguably essential to all DSSE schemes, for otherwise, the ability to update in DSSE is somewhat useless as future data are less protected. Indeed, one of the recent attacks by Zhang *et al.* [ZKP16] exploits the leakage during updates in non-forward-private schemes.

Only a limited number of solutions [SPS14,RG15,GMP16,Bos16] in the literature claimed to have forward privacy. The notion is not well understood in the earlier works [SPS14, RG15, GMP16], and is only formally defined recently by Bost [Bos16]. However, we argue that this definition cannot precisely describe in what sense a DSSE scheme is forward private. (See discussion in Section 3.2 for details.)

1.2 Our Formulation

We consider DSSE over labeled bipartite graphs, where nodes can be partitioned into two disjoint subsets X and Y, such that edges never connect two nodes from the same partition, and each edge is labeled with data from the set W. A neighbor query on node $x \in X$ (or $y \in Y$), returns a sequence of $(x, y, w) \in X \times Y \times W$ tuples if (x, y) is an edge on the graph labeled with w.

This abstract setting captures typical DSSE queries such as keyword searches over files (considering X and Y as the sets of keywords and files respectively), and labeled subgraphs queries over general graphs (considering X and Y as sets of nodes with outgoing and incoming edges respectively, and W as the set of edge labels). To generalize, we also consider neighbor queries over the entire bipartite graphs, *i.e.*, both X and Y, which enables interesting bi-directional searching applications. Bi-directional search is useful to efficiently support update in DSSE [KPR12] (since deleting a file in a DSSE supporting keyword searches implicitly requires finding all keywords the file contains). It also opens possibilities of interesting new queries such as related keyword search (which first searches for documents containing the queried keyword, then collects other keywords which are also contained in many of the matching documents).

1.3 Update Functionalities of DSSE Schemes

While the query functionality of SSE for keyword search is somewhat standard, the supported update types have large variations. Updates include additions and deletions, and can be edge-based or node-based. Schemes supporting only node additions are reasonable for some data type: *e.g.*, x as keywords and y as text files. Yet, edge updates allow fine-grained modification of existing data. In particular, schemes supporting edge additions are superior to those supporting only node additions, as the latter can be simulated by the former.

The benefits of supporting only edge deletions are however questionable, as they require the client to know about the edge to be deleted. It is unrealistic for the motivating application of SSE for keyword search: The client needs to know all keywords of a given file to completely remove the file from the server. It is desirable for a DSSE scheme to support node deletions: upon provided a node y from the client, the server can intelligently remove all edges connecting y.

To the best of our knowledge, most existing schemes only support either edge-based updates or node-based updates[1]. Supporting edge additions and node deletions simultaneously, while confining leakage, poses some technical challenges.

1.4 SSE as a Data Structure Problem

With edge additions and node deletions in mind, it is not an easy task to devise a parallel and dynamic (let alone forward private) SSE scheme. Intuitively, for data structures supporting parallel traversal, maintaining the traversal efficiency after an update often requires some global adjustment of the data structure. Consider

[1] A few exceptions include Lai-Chow [LC16] and a modified version of the one by Kamara *et al.* [KPR12]. However, these schemes leak substantial information during updates.

a balanced binary search tree. A series of deletions can degenerate the tree into a linked list which requires sequential access; and balancing the tree may require the rotation of multiple tree nodes. (That may explain why the first parallel DSSE [KP13] utilizing a red-black tree which only stores all the files in the leaf level, resulting in an efficiency loss when compared with storing some of them in internal nodes.) In the context of DSSE, delegating the maintenance work to the server often implies excessive leakage of the internal data structure.

A notable approach for (non-dynamic) SSE schemes is the invert-index used by Curtmola *et al.* [CGKO06, CGKO11], in which the encrypted database consists of an index mapping hashed keywords to sets of files containing the keywords. This inverted-index allows the server to search in time linear in the number of matching files, which is optimal. Many subsequent works follow this framework (explicitly or implicitly), which utilize some data structure to represent the sets of data, pointed by the (hashed) queries in the index. The efficiency of queries and updates correspond to the efficiency of traversing and updating the sets respectively. On the other hand, the leakage of the internal structure of the encrypted database during updates corresponds to the amount of information required or changed to update the data structure storing the sets. Most efforts for designing (D)SSE schemes is dedicated to choosing or designing this data structure.

1.5 Our Results

This work furthers the study of forward privacy of DSSE schemes over labeled bipartite graphs. We present three technical results. First, we give another formal, generalized definition of forward privacy. Specifically, our definition is parametrized by a set of updates and a restriction function on these updates. It generalizes the only existing one by Bost [Bos16], by increasing the number of classes of leakage functions allowed, yet making each class more specific. Our definition still captures the essence of forward privacy even though the leakage in different classes might vary substantially. Since different existing SSE schemes implicitly assumed different flavors of forward privacy, we believe that our generalized, parameterized definition of forward privacy is of particular interest.

Second, we propose a simple generic construction of forward private DSSE from any DSSE, which preserves the efficiency of the base scheme. The forward privacy obtained is for edge additions, such that the addition of an edge (x, y) does not leak both x and y, hence hiding the edge. This generic transformation provides insights of what constitutes forward privacy in DSSE. Since the result applies on any DSSE, again we believe it is of independent interest.

Lastly, we construct a DSSE scheme from scratch which achieves a stronger forward privacy for edge additions, such that the addition of an edge (x, y) does not leak either x or y. Our construction utilizes a specially crafted data structure named cascaded triangles[2], which supports parallel queries and updates, and has the property that adding or deleting data only affects a constant amount of existing data. Thanks to cascaded triangles, our construction features minimal leakage, and optimal query and update complexity in terms of both computation and communication up to a constant factor.

Both of our constructions support flexible edge additions and intelligent node deletions: The server can delete all edges connected to a given node, without having the client specify all the edges. It is one of a few in the literature[1].

2 Definitions

We present the necessary definitions for data representation and DSSE. For more detailed explanations, we refer the readers to Appendix A.

[2] While the design of cascaded triangles is original, we do not rule out the possibility that there are similar data structures outside the literature of SSE. To the best of our knowledge, we are unaware of any common similar data structure. There are false relatives such as fractional cascading which solves totally different problems.

2.1 Notations

Let λ be the security parameter. We use $\mathsf{poly}(\lambda)$ and $\mathsf{negl}(\lambda)$ to denote any polynomial and negligible functions respectively. $[n]$ denotes the set of positive integers not larger than n. $*$ denotes the wildcard character. $\{0,1\}^n$ denotes the set of n-bit strings while $\{0,1\}^*$ denotes the set of arbitrarily long bit strings. ϕ denotes the empty set. If X is a set, $x \leftarrow X$ samples an element x uniformly from 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

Let $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{W}$ be sets, where $\mathcal{X}$ and $\mathcal{Y}$ are disjoint, *i.e.*, $\mathcal{X} \cap \mathcal{Y} = \phi$. We denote by $\mathcal{G} = \mathcal{G}(\mathcal{X}, \mathcal{Y}, \mathcal{W})$ a set of labeled bipartite graphs specified by these sets. For a labeled bipartite graph $G \in \mathcal{G}$, its edges are labeled with $w \in W \subseteq \mathcal{W}$, and are running across $X \subseteq \mathcal{X}$ and $Y \subseteq \mathcal{Y}$. Each edge can be uniquely represented by the tuple $(x, y, w) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{W}$. The neighbor query function Qry maps a node $q = x$ (or $q = y$) and a graph G to a set of all edges connecting node x, denoted by $(x, *, *)$ (or a set of all edges connecting node y, denoted by $(*, y, *)$). Similarly, we use $(x, y, *)$ to denote the set of edges in the form of $(x, y, \cdot)$, which should be a singleton. The update function Udt maps an update u and a graph G to a new graph G'. The update $u = (\mathsf{Op}, \cdot, \cdot, \cdot)$ where $\mathsf{Op} = \mathsf{Add}$ or $\mathsf{Op} = \mathsf{Del}$ takes one of the following forms:

1. Edge Addition: $u = (\mathsf{Add}, x, y, w)$ adds the edge (x, y, w) to G.
2. Node Deletion: $u = (\mathsf{Del}, q)$ deletes the set of edges $(q, *, *)$ (or $(*, q, *)$) from G. Since $\mathcal{X}$ and $\mathcal{Y}$ are disjoint, there is no ambiguity.

2.3 Dynamic Searchable Symmetric Encryption (DSSE)

We present a definition of DSSE for the labeled bipartite graphs defined above, and its security against adaptive chosen query attack (CQA2).

Definition 1. *A dynamic symmetric searchable encryption (DSSE) scheme for the space of labeled bipartite graphs specified by $\mathcal{G}$ is a tuple of algorithms and interactive protocols* $\mathsf{DSSE}.(\mathsf{Setup}, \mathsf{Qry}_e, \mathsf{Udt}_e)$ *such that:*

- $(K, \mathsf{EDB}) \leftarrow \mathsf{Setup}(1^\lambda)$*: In the setup algorithm, the user inputs the security parameter λ. It outputs a secret key K, and an (initially empty) encrypted database* EDB *to be outsourced to the server. Alternatively, one can define a setup algorithm which takes as input the security parameter λ, and a graph G. In this case, the algorithm outputs a key K and an encrypted database* EDB *encrypting G.*

- $((K', R), (\mathsf{EDB}', R) \leftarrow \mathsf{Qry}_e((K, q), \mathsf{EDB})$*: In the query protocol, the user inputs a secret key K and a query $q \in \mathcal{X} \cup \mathcal{Y}$. The server inputs the encrypted database* EDB*. The user outputs a possibly updated key K', while the server outputs a possibly updated encrypted database* EDB'*. Both the user and the server output a sequence of responses R. For non-interactive schemes, the user first runs $(K', \tau_q) \leftarrow \mathsf{QryTkn}(K, q)$ to generate a query token τ_q. The server then runs $(\mathsf{EDB}', R) \leftarrow \mathsf{Qry}_e(\tau_q, \mathsf{EDB})$ and outputs the query results R.*

- $(K', \mathsf{EDB}') \leftarrow \mathsf{Udt}_e((K, u), \mathsf{EDB})$*: In the update protocol, the user inputs a secret key K and an update $u \in \{\mathsf{Add}, \mathsf{Del}\} \times (\mathcal{X} \cup \mathcal{Y})$. The server inputs the encrypted database* EDB*. The user outputs a possibly updated key K'. The server outputs an updated encrypted database* EDB'*. For non-interactive schemes, the user first runs $(K', \tau_u) \leftarrow \mathsf{UdtTkn}(K, u)$ to generate an update token τ_u. The server then runs $\mathsf{EDB}' \leftarrow \mathsf{Udt}_e(\tau_u, \mathsf{EDB})$ and outputs the updated database* EDB'*.*

A DSSE *scheme for the space $\mathcal{G}$ is said to be correct if, for all $\lambda \in \mathbb{N}$, all K and* EDB *output by* $\mathsf{Setup}(1^\lambda)$*, and all sequences of queries and updates, the responses to the plaintext queries equal those to the corresponding encrypted queries.*

Definition 2 (CQA2-security). *Let $\mathcal{E}$ be a DSSE scheme as defined in Definition 1. Consider the following probabilistic experiments, where $\mathcal{A}$ is a stateful adversary, $\mathcal{S}$ is a stateful simulator, $\mathcal{Z}$ denotes the environment, and $\mathcal{L}_e$, $\mathcal{L}_q$ and $\mathcal{L}_u$ are stateful leakage algorithms:*

$\mathbf{Real}_{\mathcal{E},\mathcal{A},\mathcal{Z}}(1^\lambda)$*: The adversary* $\mathcal{A}$ *acts as the server. The environment* $\mathcal{Z}$ *sends a message "setup" to the client, who executes the setup algorithm* Setup *and sends the encrypted database* EDB *to* $\mathcal{A}$*. At each time step, the environment* $\mathcal{Z}$ *chooses either a query* q *or an update* u*. For the former, the client with input* q *engages in the query protocol* Qry_e *with* $\mathcal{A}$*. For the latter, the client with input* u *engages in the update protocol* Udt_e *with* $\mathcal{A}$*. The environment* $\mathcal{Z}$ *observes the output of the client, i.e., either "abort", "update success", or the query results. The environment* $\mathcal{Z}$ *returns a bit* b *that is output by the experiment.*
$\mathbf{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\lambda)$*: The simulator* $\mathcal{S}$*, i.e., the ideal-world adversary, acts as the server. The environment* $\mathcal{Z}$ *sends a message "setup" to the client, who forwards this message to the ideal functionality* $\mathcal{F}$*, who notifies* $\mathcal{S}$ *of* $\mathcal{L}_e(\mathcal{G})$*. At each time step, the environment* $\mathcal{Z}$ *chooses either a query* q *or an update* u*. For the former, the client sends the query* q *to* $\mathcal{F}$*, who notifies* $\mathcal{S}$ *of* $\mathcal{L}_q(q)$*. For the latter, the client sends the update* u *to* $\mathcal{F}$*, who notifies* $\mathcal{S}$ *of* $\mathcal{L}_u(u)$*.* $\mathcal{S}$ *sends* $\mathcal{F}$ *either the message "abort" or "continue". As a result,* $\mathcal{F}$ *sends the client "abort", "update success", or the query results. The environment* $\mathcal{Z}$ *observes these messages, and returns a bit* b *that is output by the experiment.*

We say that DSSE *is* $(\mathcal{L}_e,\mathcal{L}_q,\mathcal{L}_u)$*-secure against adaptive dynamic chosen-query attacks (CQA2) if for all honest-but-curious PPT adversaries* $\mathcal{A}$*, there exists a PPT simulator* $\mathcal{S}$ *such that for all non-uniform polynomial time environment* $\mathcal{Z}$

$$|\Pr[\mathbf{Real}_{\mathcal{E},\mathcal{A},\mathcal{Z}}(1^\lambda)=1]-\Pr[\mathbf{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\lambda)=1]|\leq \mathsf{negl}(\lambda)\,.$$

3 Forward Privacy in DSSE

Intuitively, forward privacy ensures that newly added data remains hidden to the server who might have learned some secrets during previous queries, until it must be revealed by a later query. To formalize, instead of tinkering with the CQA2-security definition of DSSE, we define forward privacy based on the property of the leakage function $\mathcal{L}_u$. This is more convenient since the information leaked by $\mathcal{L}_u$ is sufficient for the simulator in the CQA2-security definition to simulate the updates. Similar to the semantic security of encryption schemes, we require that the leakages $\mathcal{L}_u$ on a pair of updates are indistinguishable, capturing the idea that not even a single bit about the update is leaked to the server. Note that the definition does not limit the types of the update u. Indeed, we can consider forward privacy for not only additions, but also deletions. In layman terms, suppose that the server learns during the query protocol about the association of q with some data, which is then deleted by the client. The server should not notice that the data are deleted until q is queried again: By that time the server can compare the query results and discover the deletion.

3.1 Our Definition

We first give a general definition of forward privacy parametrized by a set of updates and a restriction function, then discuss useful ways to parameterize it.

Definition 3 (Forward Privacy). *Let* DSSE *be a* $(\mathcal{L}_e,\mathcal{L}_q,\mathcal{L}_u)$*-CQA2 secure DSSE scheme for labeled bipartite graphs specified by* $\mathcal{G}$*. Let* $\mathcal{U}$ *be a set of updates and restriction* $p:\mathcal{U}^2\to\{0,1\}$ *be a predicate function. We say that* DSSE *is:* $(\mathcal{U},p)$*-forward private, if for any* $u_b\in\mathcal{U}$ *where* $b\in\{0,1\}$ *such that* $p(u_0,u_1)=1$*, and any PPT distinguisher* $\mathcal{D}$*, it holds that*

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

Table 1 lists some useful combinations of $\mathcal{U}$ and p, denoted by $\mathcal{U}_i$ and p_i respectively for $i\in[6]$. One may also consider a set $\mathcal{U}$ which is a union of some of the (disjoint) $\mathcal{U}_i$'s, and a restriction p which is a composition of the corresponding p_i's: For $p_i:\mathcal{U}_i^2\to\{0,1\}$, define $p=p_i+p_j:(\mathcal{U}_i\cup\mathcal{U}_j)^2\to\{0,1\}$ such that $(p_i+p_j)(u)=1$ if $u\in\mathcal{U}_i$ and $p_i(u)=1$, or $u\in\mathcal{U}_j$ and $p_j(u)=1$.

Note that there might exist schemes which are $(\mathcal{U}_i,p_i)$- and $(\mathcal{U}_j,p_j)$-forward private but not $(\mathcal{U}_i\cup\mathcal{U}_j,p_i+p_j)$-forward private. For example, if $\mathcal{U}_i$ and $\mathcal{U}_j$ are sets of additions and deletions respectively, while the distinguisher cannot tell which addition or deletion is chosen, it can separate additions from deletions.

Table 1. Useful combinations of $\mathcal{U}$ and p as parameters for forward privacy. (See Section 2.2 and Appendix A.2 for details about the update operations.)

i	Type	Sets of updates $\mathcal{U}_i$	Updates $u_b \in \mathcal{U}_i$ $b \in \{0,1\}$	Possible conditions such that $p_i = 1$ (If there are no restrictions, $p_i \equiv 1$)
1	edge	$\{(\mathsf{Add}, x, y, w)\}$	$(\mathsf{Add}, x_b, y_b, w_b)$	$x_0 = x_1$ or $y_0 = y_1$
2	node	$\{(\mathsf{Add}, \mathbf{x}, y, \mathbf{w})\}$	$(\mathsf{Add}, \mathbf{x}_b, y_b, \mathbf{w}_b)$	$y_0 = y_1$
3	node	$\{(\mathsf{Add}, x, \mathbf{y}, \mathbf{w})\}$	$(\mathsf{Add}, x_b, \mathbf{y}_b, \mathbf{w}_b)$	$x_0 = x_1$
4	edge	$\{(\mathsf{Del}, x, y)\}$	(Del, x_b, y_b)	$x_0 = x_1$ or $y_0 = y_1$
5	node	$\{(\mathsf{Del}, x)\}$	(Del, x_b)	$\lvert(x_0, *, *)\rvert = \lvert(x_1, *, *)\rvert$
6	node	$\{(\mathsf{Del}, y)\}$	(Del, y_b)	$\lvert(*, y_0, *)\rvert = \lvert(*, y_1, *)\rvert$

3.2 Bost's Definition

The forward privacy definition of Bost [Bos16] requires that the leakage function of $u = (\mathsf{Op}, x, y, w)$ can be written as a function of the operation Op and the node y, where Op can be addition or deletion. We argue that the range of leakage functions which satisfy this requirement is so wide, such that the definition does not precisely describe in what sense a DSSE scheme is forward secure. At one extreme, consider a scheme of which the leakage function is given by $\mathcal{L}_u(\mathsf{Op}, x, y, w) = (\mathsf{Op}, y)$. If $\mathcal{L}_u$ accurately (not overly) captures the leakage, then adding or removing edges to or from a node y leaks the identity of the node y itself. Such a scheme is vulnerable to frequency attacks: The attacker keeps a table mapping each y to the number of times y is updated. With the aid of external information, it can possibly extract information about y, such as its importance. Similar attacks have been demonstrated using search patterns [LZWT14]. At another extreme, Bost's construction [Bos16] leaks nothing during updates, *i.e.*, $\mathcal{L}_u(\mathsf{Op}, x, y, w) = \phi$. On the other hand, Bost's definition is also not general enough. There are other types of leakage functions, *e.g.*, $\mathcal{L}_u(\mathsf{Op}, x, y, w) = (\mathsf{Op}, x)$, which intuitively capture forward privacy, but are not covered by this definition. In contrast, our definition of $(\mathcal{U}, p)$-forward privacy classified different types of update in a more fine-grained manner.

In the perspective of our model, Bost's definition can be regarded as special cases of $(\mathcal{U}_1, p_1)$- (achieved by our generic construction in Section 4) and $(\mathcal{U}_4, p_4)$-forward privacy. Intuitively, the restrictions p_1 and p_4 mean that an update $u = (\mathsf{Op}, x, y, w)$ is protected by hiding one end of the connection between x and y. Therefore, similar to the above, it might be the case that the updates u_0 and u_1, where $u_b = (\mathsf{Add}, x_b, y, w)$ for $b \in \{0, 1\}$, are linkable as they correspond to the same node y, making the scheme vulnerable to the same frequency attack. On the other hand, $(\mathcal{U}_1, 1)$- (achieved by our concrete construction in Section 5) and $(\mathcal{U}_4, 1)$-forward privacy completely hides the relation (x, y), making the addition and deletion of an edge (x, y, w) oblivious respectively (since w can be hidden simply by symmetric key encryption). Whether or not the stronger forward privacy is needed depends on the specific application scenarios.

3.3 Forward Privacy for Deletions

In the rest of this work, we focus on $\mathcal{U} = \mathcal{U}_1 = \{(\mathsf{Add}, x, y, w)\}$, with and without restrictions, *i.e.*, $p = p_1$ and $p \equiv 1$, respectively. In other words, we do not consider forward privacy for deletions. To argue for this design decision, we observe that while the scheme of Bost [Bos16] performs "lazy edge deletion" (adding "deleted" edges rather than actually deleting), ours perform actual node deletion. The former increases the size of the encrypted database (by one edge), hence it is possible to make (edge) additions and deletions indistinguishable. The latter allows immediate space reclamation. This makes edge additions (which usually increase the size of the encrypted database) and node deletions easily distinguishable. Furthermore, since actual deletions of different nodes may result in a shrink of the database size in varying degrees, they are also easily distinguishable. In effect, we trade "forward privacy for (lazy) deletions" for efficiency. If forward privacy for deletions is a concern and lazy deletions are acceptable, using a similar technique of maintaining another instance of encrypted database for lazy deletions [Bos16], schemes which are forward private for edge additions can be generically transformed to provide forward privacy for both edge additions and node

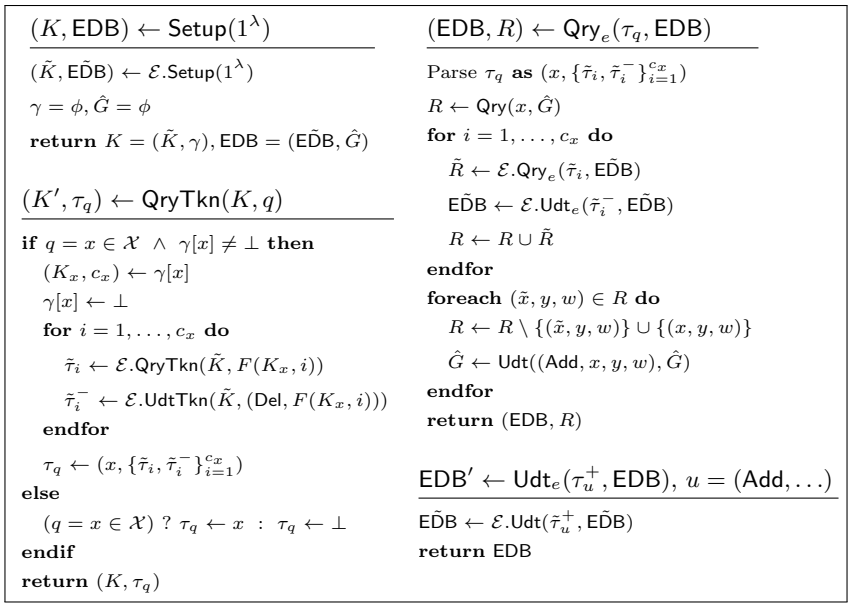

Fig. 1. Algorithms of generic construction for forward privacy

deletions (simultaneously): To delete a node $y \in Y$, the client adds an edge connecting y to a special "deleted" node in X. We leave the details to the full version of this paper. In this sense, forward privacy for additions is a key property. We also believe that it is sufficient for practical applications by itself.

4 Forward Privacy from any DSSE

In this section, we will show that a DSSE scheme with $(\mathcal{U}_1, p_1)$-forward privacy can be constructed from any DSSE scheme $\mathcal{E}$, where $\mathcal{U}_1$ and p_1 are defined in Table 1. For simplicity of our description below, we assume the base scheme to be non-interactive, so that the resulting scheme is also non-interactive[3]. Our transformation can be easily adapted to interactive schemes.

Our construction is inspired by that of Rizomiliotis and Gritzalis [RG15] which uses fresh keys for newly added data. The main idea is to locally maintain a table γ of pseudorandom function (PRF) keys K_x and counters c_x for each query $q = x$, so that adding an edge (x, y, w) is translated to adding another edge $(F(K_x, c_x), y, w)$. Our scheme also adopts the technique of Hahn and Kerschbaum [HK14], who observe that when the set $(x, *, *)$ is leaked upon querying on x, there is no need to protect the set by encryption any longer. To speed up subsequent queries, the server should thus transfer the set encrypted in the scheme to a plaintext bipartite graph $\hat{G}$. We assume an efficient data structure for representing the graph $\hat{G}$, so that neighbor queries, edge additions, and node deletions in $\hat{G}$ are parallelizable and have time complexity linear in the number of affected nodes only. There might be many ways to construct such a data structure. Cascaded triangles introduced in Section 5 is one example.

4.1 Our Construction

Let $\mathcal{E}$ be a DSSE scheme for $\mathcal{G} = \mathcal{G}(\{0,1\}^\lambda, \mathcal{Y}, \mathcal{W})$, and $F : \{0,1\}^\lambda \times \mathcal{X} \to \{0,1\}^\lambda$ be a pseudorandom function (PRF). We construct a DSSE scheme for $\mathcal{G}' = \mathcal{G}'(\mathcal{X}, \mathcal{Y}, \mathcal{W})$. The resulting scheme supports queries over $\mathcal{X}$, assuming the base scheme supports queries over $\{0,1\}^\lambda$. Figures 1 and 2 formally describe the construction.

[3] Candidate base schemes include [LC16] and a modified version of [KPR12].

$(K', \tau_u^+) \leftarrow \mathsf{UdtTkn}(K, u = (\mathsf{Add}, \ldots))$

parse u **as** (Add, x, y, w)
if $\gamma[x] = \bot$ **then**
  $K_x \leftarrow \{0,1\}^\lambda, c_x \leftarrow 1$
else
  $(K_x, c_x) \leftarrow \gamma[x], c_x \leftarrow c_x + 1$
endif
$\gamma[x] \leftarrow (K_x, c_x)$
$\tau_u^+ \leftarrow \mathcal{E}.\mathsf{UdtTkn}(\tilde{K}, (\mathsf{Add}, F(K_x, c_x), y, w))$
return (K, τ_u^+)

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

if $\tau_u^- = (x \in \mathcal{X}, \{\tilde{\tau}_i^-\}_{i=1}^{c_x})$ **then**
  $\hat{G} \leftarrow \mathsf{Udt}((\mathsf{Del}, x), \hat{G})$
  for $i = 1, \ldots, c_x$ **do**
    $\tilde{\mathsf{EDB}} \leftarrow \mathcal{E}.\mathsf{Udt}_e(\tau_i^-, \tilde{\mathsf{EDB}})$
  endfor
elseif $\tau_u^- = (y \in \mathcal{Y}, \tilde{\tau}^-)$ **then**
  $\hat{G} \leftarrow \mathsf{Udt}((\mathsf{Del}, y), \hat{G})$
  $\tilde{\mathsf{EDB}} \leftarrow \mathcal{E}.\mathsf{Udt}_e(\tau^-, \tilde{\mathsf{EDB}})$
endif
return EDB

$(K', \tau_u^-) \leftarrow \mathsf{UdtTkn}(K, u = (\mathsf{Del}, \cdot))$

if $u = (\mathsf{Del}, x)$ **then**
  if $\gamma[x] \neq \bot$ **then**
    $(K_x, c_x) \leftarrow \gamma[x]$
    $\gamma[x] \leftarrow \bot$
    for $i = 1, \ldots, c_x$ **do**
      $r_i \leftarrow F(K_x, i)$
      $\tilde{\tau}_i^- \leftarrow \mathcal{E}.\mathsf{UdtTkn}(\tilde{K}, (\mathsf{Del}, r_i))$
    endfor
    $\tau_u^- \leftarrow (x, \{\tilde{\tau}_i^-\}_{i=1}^{c_x})$
  else
    $\tau_u^- \leftarrow x$
  endif
elseif $u = (\mathsf{Del}, y)$ **then**
  $\tilde{\tau}^- \leftarrow \mathcal{E}.\mathsf{UdtTkn}(\tilde{K}, (\mathsf{Del}, y))$
  $\tau_u^- \leftarrow (y, \tilde{\tau}^-)$
else
  $\tau_u^- \leftarrow \bot$
endif
return τ_u^-

Fig. 2. Algorithms of generic construction for forward privacy (cont.)

The setup algorithm initializes the base scheme $\mathcal{E}$, which yields a secret key $\tilde{K}$ and encrypted database $\tilde{\mathsf{EDB}}$. It also initializes an empty dictionary γ and an empty bipartite graph $\hat{G} \in \mathcal{G}'$. The new secret key K consists of $\tilde{K}$ and γ, while $\tilde{\mathsf{EDB}}$ and $\hat{G}$ are outsourced to the server. The dictionary γ maps a query $q = x \in \mathcal{X}$ to a PRF key K_x and a counter c_x.

To perform an update $u = (\mathsf{Add}, x, y, w)$, the client increments $c_x \leftarrow c_x + 1$, and transforms the update into $\tilde{u} = (\mathsf{Add}, F(K_x, c_x), y, w)$ of the base scheme. To perform an update $u = (\mathsf{Del}, x)$, the client removes the x-th row of γ, and sends to the server the update tokens for $(\mathsf{Del}, F(K_x, i))$ for $i \in [c_x]$. The update $u = (\mathsf{Del}, y)$ is processed as in the base scheme.

Finally, to query $q = x \in \mathcal{X}$, the client removes the x-th row of γ, and sends to the server the query tokens of $F(K_x, i)$ for $i \in [c_x]$. Given these tokens, the server retrieves the intended response R. Additionally, the client also sends the update tokens for $(\mathsf{Del}, F(K_x, i))$ for $i \in [c_x]$. The server uses these tokens to collect and remove the set of edges $R = (x, *, *)$ from the base scheme, and merge R to the plaintext graph $\hat{G}$.

The correctness follows directly from that of the base scheme $\mathcal{E}$.

4.2 Analysis

Efficiency. Our generic transformation almost preserves the efficiency of the underlying DSSE scheme. For most algorithms, the preservation is apparent. We highlight the slightly more complicated cases, namely, the query on x and the update (Del, x). In the former, c_x queries on $F(K_x, i)$ for $i \in [c_x]$ are executed, while in both cases c_x deletions $(\mathsf{Del}, F(K_x, i))$ for $i \in [c_x]$ are required. We analyze their efficiency assuming the following operations of the underlying DSSE scheme each takes constant time: the computation of the query token for each $F(K_x, i)$; the server computation for querying on each $F(K_x, i)$ (since by the pseudorandomness of F, the query should only return a single edge); the computation of each delete token; and the server computation for deleting each set $(F(K_x, i), *, *)$ (since each set is actually a singleton). Overall, the resulting scheme incurs $O(c_x)$ computation and communication costs for both the client and

the server where c_x is the number of newly matched data item. These are extra costs on top of the costs for retrieving the previously matched data (in plaintext), *i.e.*, constant computation cost of the client, sublinear computation cost of the server, and sublinear communication cost of both. Since c_x is reset to zero whenever x is queried, the amortized extra costs of both the client and the server are low.

Security. Let $\mathcal{E}$ be an $(\tilde{\mathcal{L}}_e, \tilde{\mathcal{L}}_q, \tilde{\mathcal{L}}_u)$-CQA2 secure DSSE scheme for labeled bipartite graphs specified by $\mathcal{G}$; and $F : \{0,1\}^\lambda \times \mathbb{N} \to \{0,1\}^\lambda$ be a pseudorandom function. We construct a simulator which, when given the leakage defined by the leakage functions $(\mathcal{L}_e, \mathcal{L}_q, \mathcal{L}_u)$, simulates the ideal functionality of the base scheme. These leakage functions are defined based on those $(\tilde{\mathcal{L}}_e, \tilde{\mathcal{L}}_q, \tilde{\mathcal{L}}_u)$ of the base scheme. Concretely, we define the leakage functions as follows:

- $\mathcal{L}_e(\mathcal{G}') = \tilde{\mathcal{L}}_e(\mathcal{G})$,
- $\mathcal{L}_u(\mathsf{Add}, x, y, w) = (\tilde{x}, \tilde{\mathcal{L}}_u(\mathsf{Add}, \tilde{x}, y, w))$ for dummy node $\tilde{x} \leftarrow \{0,1\}^\lambda$,
- $\mathcal{L}_u(\mathsf{Del}, x) = (x, \{\tilde{\mathcal{L}}_u(\mathsf{Del}, \tilde{x}_i)\}_{i=1}^{c_x})$,
- $\mathcal{L}_u(\mathsf{Del}, y) = (y, \tilde{\mathcal{L}}_u(\mathsf{Del}, y))$,
- $\mathcal{L}_q(x) = (x, \mathsf{AP}_t(x), \{\tilde{\mathcal{L}}_q(\tilde{x}_i), \tilde{\mathcal{L}}_u(\mathsf{Del}, \tilde{x}_i)\}_{i=1}^{c_x})$, for dummy nodes $\tilde{x}_i$ defined by $\mathcal{L}_u(\mathsf{Add}, x, y, \cdot)$ for updates after the previous query on x.

Theorem 1. *Assume that $\mathcal{E}$ is $(\tilde{\mathcal{L}}_e, \tilde{\mathcal{L}}_q, \tilde{\mathcal{L}}_u)$-CQA2 secure, and F is pseudorandom, then the above construction is $(\mathcal{L}_e, \mathcal{L}_q, \mathcal{L}_u)$-CQA2 secure. Furthermore, let p'_1 be a function such that $p'_1(u_0, u_1) = 1$ if and only if $y_0 = y_1$ and $w_0 = w_1$[4]. Then the construction is $(\mathcal{U}_1, p'_1)$-forward private, where $\mathcal{U}_1$ is defined in Table 1.*

Appendix B.1 shows the proof of Theorem 1.

Extension with ORAM. In the above construction, the size of the local storage is linear in the size of $\mathcal{X}$, in the worst case. To avoid such local storage, the client can outsource the table γ to a remote oblivious RAM (ORAM), similar to the solution of Rizomiliotis and Gritzalis [RG15]. Given a sequence of access operations to a RAM, an ORAM compiler can generate a (longer) sequence of access operations to the ORAM. The new sequence preserves the functionality of the original access sequence, while the oblivious access sequences generated from any two original access sequences of the same length are computationally indistinguishable. State-of-the-art ORAM schemes only incur a polylogarithmic bandwidth overhead in the size of the memory. Therefore, by substituting the local storage of γ by ORAM, the client can avoid a local storage at the cost of extra communication whenever the dictionary γ is accessed.

5 Forward Privacy from Scratch

We next construct (interactive) DSSE which achieves forward privacy directly. First, a new data structure, named *cascaded triangles*, is designed to represent labeled bipartite graphs which supports neighbor queries, edge additions, and node deletions efficiently. We then transform it into its encrypted version.

The construction of cascaded triangles is motivated by the following. Since the neighbor queries and node deletions require traversing the sets $(x, *, *)$ and $(*, y, *)$, their data structure representations are critical for the efficiency, and later the security of the resulting DSSE scheme. In particular, they determine whether the desired operations can be executed in parallel, and how much information has to be leaked to the server for performing such operations. For example, linked list [KPR12] traversal and updates are inherently sequential, yet updating only has a local effect (on the previous and next nodes). However, random binary search tree [LC15] exhibits parallel traversal and updates, yet updating affects (or leaks) the subtree rooted from the altered node. Thus, cascaded triangles is designed to support parallel traversal and local updates simultaneously.

[4] We can drop the restriction $w_0 = w_1$ by simply encrypting w during additions.

5.1 Warm Up: Plaintext Cascaded Triangles

Overview. Our goal is to store the bipartite graph G so that neighbor queries and deletions over X or Y can be executed in sublinear time. We can do so by pre-computing the set of edges connected to each x (and y), and storing the set by a data structure which allows efficient traversal. For any x, consider the set of edges connecting x. We pack this set into multiple perfect binary trees, called triangles, by first forming the largest triangle possible, subtracting the edges which are already packed, then continuing to form the next largest triangle. The resulting triangles thus have strictly decreasing (cascading) heights, except for the last two which may have equal heights. This invariant is to be maintained in any later updates. To add an edge connecting x, we check if the two shortest triangles have the same height. If so, we add a new node representing the new edge on top of the two triangles, merging them into one larger triangle. Otherwise, the new node is added as a new triangle of height 1. To delete an edge, we delete the node representing this edge by replacing it with the root of the shortest triangle, splitting the latter into two smaller triangles. We can see that the invariant is still maintained after each addition and each deletion. Finally, to traverse the data structure, one may use any (parallel) tree traversal algorithms.

Setup. Concretely, cascaded triangles consists of dictionaries γ, δ, and η. We can think of γ as local states stored at the client side, while δ and η are outsourced to the server. The dictionary η is the one to store the actual data. It maps an address addr to a tuple (a, b), where $b = (\mathsf{chd}_0, \mathsf{chd}_1)$ specifies the addresses of the left and right child respectively. b is maintained so that nodes in η form perfect binary trees (triangles). This means either both addresses $(\mathsf{chd}_0, \mathsf{chd}_1)$ are empty ($\perp$) or both are valid addresses occupied in η. To store an edge (x, y, w) into η, the edge is copied twice into $a^\triangle = a^\triangledown = (x, y, w)$. The tuples $(a^\triangle, b^\triangle)$ and $(a^\triangledown, b^\triangledown)$ are stored at random addresses $\mathsf{addr}^\triangle$ and $\mathsf{addr}^\triangledown$ in η respectively. The addresses $\mathsf{addr}^\triangle$ and $\mathsf{addr}^\triangledown$ are said to be *duals* of each other, and are registered in δ, *i.e.*, $\delta[\mathsf{addr}^\triangle] = \mathsf{addr}^\triangledown$ and $\delta[\mathsf{addr}^\triangledown] = \mathsf{addr}^\triangle$.

Globally, we describe how the nodes in η are connected to each other via the addresses stored in b. We collect all $\eta[\mathsf{addr}] = (a, b)$ corresponding to the edges in $(q, *, *)$ (or $(*, q, *)$). Let $n_q = |(q, *, *)|$ (or $|(*, q, *)|$). We pack these tuples into triangles of cascading heights $h_1 \le h_2 < h_3 < \ldots < h_k$, where $k \le \lceil \lg n_q \rceil + 1$. Note the possible equality between h_1 and h_2 but not the others. The ordering of the nodes is implicitly determined by the update algorithms, but does not matter here. Using the procedures described in the overview, given the size n_q, the heights $h_1, \ldots, h_k$ are uniquely determined. It is possible to represent the heights compactly by a trinary string $h \in \{0, 1, 2\}^{\lceil \lg n_q \rceil}$, such that the i-th trit (trinary digit) is set to t, if there are t triangles of height i. Due to the constraints on the heights, only the least significant non-zero trit can be set to 2. Finally, the addresses of the roots and the heights of these triangles are stored in $\gamma[q] = (\mathsf{addr}_1, \ldots, \mathsf{addr}_k, h)$.

Queries and Traversal. Traversing the sets $(q, *, *)$ and $(*, q, *)$ are straightforward with the above structure: First, retrieve the roots of the triangles from $\gamma[q] = (\mathsf{addr}_1, \ldots, \mathsf{addr}_k, h)$. Then, use parallel tree-traversal algorithms to traverse the trees from the root starting at each of these addresses. Notice that the neighbor query function Qry on x and y are supported by traversing the sets $\gamma[x] = (x, *, *)$ and $\gamma[y] = (*, y, *)$ respectively.

Add. To add a new edge (x, y, w), we first retrieve $\gamma[x] = (\mathsf{addr}_1^\triangle, \ldots, \mathsf{addr}_k^\triangle, h^\triangle)$ and check whether the triangles rooted at $\mathsf{addr}_1^\triangle$ and $\mathsf{addr}_2^\triangle$ have the same height.

To do so, we take a detour to describe the $+1$ *operation* in $h + 1$. Recall that only the least significant non-zero trit, say the i-th trit, in h can be set to 2. The operation $h + 1$ adds 1 to the i-th trit (instead of the least significant trit as in normal addition), which sets the i-th trit to 0 and carries 1 to the $(i + 1)$-trit. Denote this event by $\mathsf{Carry}(h + 1) = 1$. Otherwise, $h + 1$ simply adds 1 to the least significant trit as in normal addition, denoted by $\mathsf{Carry}(h + 1) = 0$. Later, for deletion, we would need the -1 *operation* which is the "reverse" of $+1$. Concretely, $h - 1$ subtracts 1 from the least significant non-zero trit, say the i-th trit, and set the $(i - 1)$-th trit to 2 if $i > 1$.

With the above procedures, checking the heights of the first two triangles can be done by simply checking whether the least significant non-zero trit in $h^\triangle$ equals 2, or equivalently whether $\mathsf{Carry}(h^\triangle + 1) = 1$. If that is

$(K, \mathsf{EDB}) \leftarrow \mathsf{Setup}(1^\lambda, |\mathcal{X}|, |\mathcal{Y}|, |\mathcal{W}|)$

$\gamma = \phi, \delta = \phi, \eta = \phi, \hat{G} = \phi$

return $K = \gamma, \mathsf{EDB} = (\delta, \eta, \hat{G})$

$\mathsf{Trav}(\mathsf{EDB}, \{\mathsf{addr}_j\}_{j=1}^k, \mathsf{ak}, \mathsf{bk})$

if $k > 1$ **then**
  $R = \phi, D = \phi$
  for $j = 1, \ldots, k$ **do**
    $(R', D') \leftarrow \mathsf{Trav}(\mathsf{EDB}, \mathsf{addr}_j, \mathsf{ak}, \mathsf{bk})$
    $R = R \cup R', D = D \cup D'$
  endfor
else
  $(c_a, c_b) \leftarrow \eta[\mathsf{addr}_1], \eta[\mathsf{addr}_1] \leftarrow \perp$
  $\overline{\mathsf{addr}}_1 \leftarrow \delta[\mathsf{addr}_1], \delta[\mathsf{addr}_1] \leftarrow \perp$
  if $\mathsf{ak} \neq \perp$ **then**

$\mathsf{Trav}(\mathsf{EDB}, \{\mathsf{addr}_j\}_{j=1}^k, \mathsf{ak}, \mathsf{bk})$ (cont.)

    $(x, y, w) \leftarrow \mathsf{NCE.Dec}(\mathsf{ak}, c_a)$
  else
    $(x, y, w) \leftarrow c_a$
  endif
  $(\mathsf{chd}_0, \mathsf{chd}_1) \leftarrow \mathsf{NCE.Dec}(\mathsf{bk}, c_b)$
  if $\mathsf{chd}_0 \neq \perp$ ($\vee$ $\mathsf{chd}_1 \neq \perp$) **then**
    $(R', D') \leftarrow \mathsf{Trav}(\mathsf{EDB}, \mathsf{chd}_0, \mathsf{ak}, \mathsf{bk})$
    $(R'', D'') \leftarrow \mathsf{Trav}(\mathsf{EDB}, \mathsf{chd}_1, \mathsf{ak}, \mathsf{bk})$
    $R = R' \cup R'' \cup \{(x, y, w)\}$
    $D = D' \cup D'' \cup \{\overline{\mathsf{addr}}_1\}$
  else
    $R = \{(x, y, w)\}, D = \{\overline{\mathsf{addr}}_1\}$
  endif
endif
return (R, D)

Fig. 3. Setup and Traverse algorithm of encrypted cascaded triangles

the case, we add $(a^\triangle, b^\triangle)$ where $a^\triangle = (x, y, w)$ and $b^\triangle = (\mathsf{addr}_1^\triangle, \mathsf{addr}_2^\triangle)$ to a random address $\mathsf{addr}^\triangle$ in η. We then update $\gamma[x] \leftarrow (\mathsf{addr}^\triangle, \mathsf{addr}_3^\triangle, \ldots, \mathsf{addr}_k^\triangle, h^\triangle + 1)$, where the two addresses $\mathsf{addr}_1^\triangle$ and $\mathsf{addr}_2^\triangle$ are replaced by the new $\mathsf{addr}^\triangle$. Otherwise, we add $(a^\triangle, b^\triangle)$ where $a^\triangle = (x, y, w)$ and $b^\triangle = (\perp, \perp)$ to a random address $\mathsf{addr}^\triangle$ in η, and update $\gamma[x] \leftarrow (\mathsf{addr}^\triangle, \mathsf{addr}_1^\triangle, \ldots, \mathsf{addr}_k^\triangle, h^\triangle + 1)$. The difference is highlighted in red.

Similarly, we retrieve $(\mathsf{addr}_1^\triangledown, \ldots, \mathsf{addr}_k^\triangledown, h^\triangledown) \leftarrow \gamma[y]$ and check whether $\mathsf{Carry}(h^\triangledown + 1) = 1$. If so, we add $((x, y, w), (\mathsf{addr}_1^\triangledown, \mathsf{addr}_2^\triangledown))$ to a random address $\mathsf{addr}^\triangledown$ in η, and update $\gamma[y] \leftarrow (\mathsf{addr}^\triangledown, \mathsf{addr}_3^\triangledown, \ldots, \mathsf{addr}_k^\triangledown, h^\triangledown + 1)$. Otherwise, we add $((x, y, w), (\perp, \perp))$ to $\mathsf{addr}^\triangledown$, and update $\gamma[y] \leftarrow (\mathsf{addr}^\triangledown, \mathsf{addr}_1^\triangledown, \ldots, \mathsf{addr}_k^\triangledown, h^\triangledown + 1)$.

Delete. To delete node x, or equivalently the set of edges $(x, *, *)$, we first traverse the set $(x, *, *)$ using the above traversal algorithm. We delete all the traversed nodes in η as well as the row $\gamma[x]$. It remains to delete the dual nodes of the traversed nodes. To do so, for each traversed address $\mathsf{addr}^\triangle$ with $a = (x, y, w)$, look up $\delta[\mathsf{addr}^\triangle] = \mathsf{addr}^\triangledown$ and $\gamma[y] = (\mathsf{addr}_1^\triangledown, \ldots, \mathsf{addr}_k^\triangledown, h^\triangledown)$. We wish to replace the content of $\eta[\mathsf{addr}^\triangledown] = (a^\triangledown, b^\triangledown)$ located in the middle of some triangle by the content of $\eta[\mathsf{addr}_1^\triangledown]$, the root of the smallest triangle, which splits the smallest triangle into two smaller ones. In this way, the heights of the resulting triangles still satisfy the required constraints.

Concretely, we perform the following steps. 1) Look up $\delta[\mathsf{addr}_1^\triangledown] = \mathsf{addr}_1^\triangle$. 2) Delete $\delta[\mathsf{addr}^\triangle]$ and $\delta[\mathsf{addr}_1^\triangledown]$. 3) Update $\delta[\mathsf{addr}_1^\triangle] \leftarrow \mathsf{addr}^\triangledown$ and $\delta[\mathsf{addr}^\triangledown] \leftarrow \mathsf{addr}_1^\triangle$. 4) Look up $\eta[\mathsf{addr}_1^\triangledown] = (a_1^\triangledown, b_1^\triangledown)$, where $b_1^\triangledown = (\mathsf{addr}_0^\blacktriangledown, \mathsf{addr}_1^\blacktriangledown)$. 5) Update $\eta[\mathsf{addr}^\triangledown] \leftarrow (a_1^\triangledown, b_1^\triangledown)$ and delete $\eta[\mathsf{addr}_1^\triangledown]$. 6) Update $\gamma[y] = (\mathsf{addr}_0^\blacktriangledown, \mathsf{addr}_1^\blacktriangledown, \mathsf{addr}_2^\triangledown, \ldots, \mathsf{addr}_k^\triangledown, h^\triangledown - 1)$, where $h^\triangledown - 1$ is the reverse of $h^\triangledown + 1$. We omit the deletion of $(*, y, *)$ which is similar to the above.

Efficiency. The storage cost of cascaded triangles is $O(|X|+|Y|+|G|) = O(|G|)$. The complexity of querying (or deleting) x and y are $O(|(x, *, *)|)$ and $O(|(*, y, *)|)$ respectively. Addition of an edge can be computed in constant time.

5.2 Our Construction: Encrypted Cascaded Triangles

We now transform the plaintext cascaded triangles into its encrypted version. Recall that the goal of the client is to encrypt a labeled bipartite graph G into an encrypted database EDB which still supports neighbor queries, edge additions, and node deletions. To do so, the client represents G using cascaded triangles, stores γ locally, and outsources δ and the encrypted η to the server. The encryption should be *non-committing*,

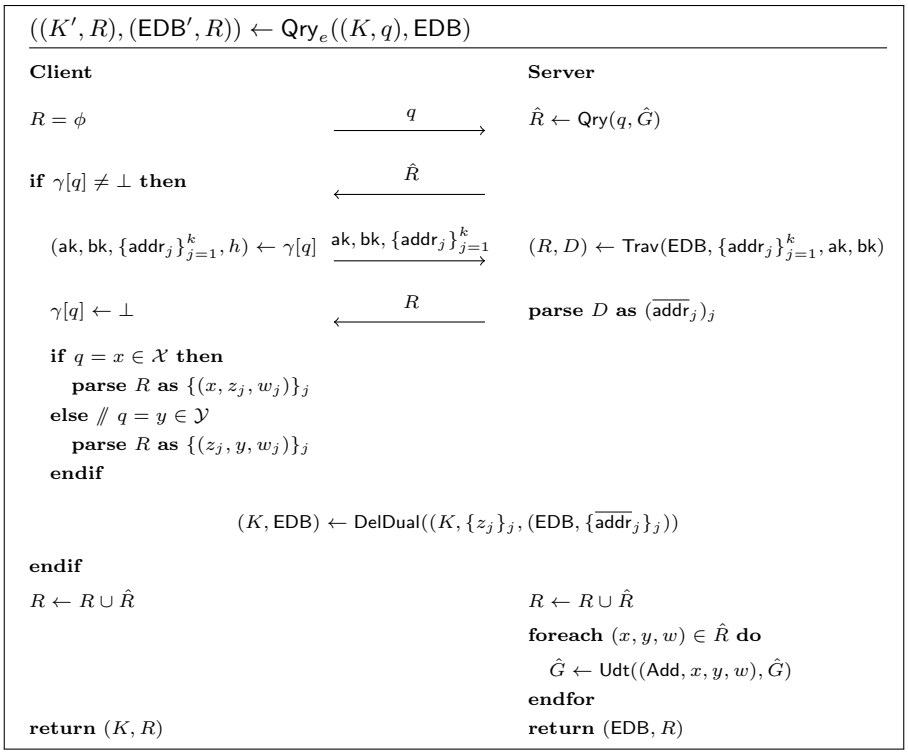

Fig. 4. Query protocol of encrypted cascaded triangles

such that there exists a simulator which can simulate the ciphertexts for new data without any leakage. When some data is to be returned upon queries or is deleted, the simulator is given enough leakage so that it can "explain" the dummy ciphertexts. Furthermore, when the sets $(x, *, *)$ and $(*, y, *)$ are leaked upon querying x and y respectively, there is no need to protect the sets by encryption any longer [HK14]. To speed up subsequent queries, the server should transfer the set from the encrypted η to a plaintext labeled bipartite graph $\hat{G}$. Thus, we can conceptually split G into two disjoint subgraphs $G = \tilde{G} \cup \hat{G}$, where $\tilde{G}$ is encrypted and $\hat{G}$ is in plaintext.

Encrypted cascaded triangles are similar to the plaintext counterparts. We highlight the differences and omit the identical parts.

Setup and Overview. Let $\mathsf{NCE}.(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key non-committing encryption scheme[5]. The correctness of our scheme will follow directly from that of NCE. For each edge (x, y, w) in the encrypted subgraph $\tilde{G}$, $\eta[\mathsf{addr}^{\triangle}]$ stores the tuple $(c_a^{\triangle}, c_b^{\triangle})$, where $c_a^{\triangle}$ and $c_b^{\triangle}$ are non-committing ciphertexts of $a^{\triangle} = (x, y, w)$ and $b^{\triangle} = (\mathsf{chd}_0, \mathsf{chd}_1)$ under the keys $\mathsf{ak}^{\triangle}$ and $\mathsf{bk}^{\triangle}$ respectively. The keys $\mathsf{ak}^{\triangle}$ and $\mathsf{bk}^{\triangle}$ are independently generated for each x. Similarly, $\eta[\mathsf{addr}^{\triangledown}]$ stores the tuple $(c_a^{\triangledown}, c_b^{\triangledown})$ encrypted under $\mathsf{ak}^{\triangledown}$ and $\mathsf{bk}^{\triangledown}$ respectively. The keys $\mathsf{ak}^{\triangledown}$ and $\mathsf{bk}^{\triangledown}$ are independently generated for each y. For each x, $\gamma[x]$ additionally stores secret keys $\mathsf{ak}^{\triangle}$ and $\mathsf{bk}^{\triangle}$ associated to x. Similarly, for each y, $\gamma[y]$ additionally stores secret keys $\mathsf{ak}^{\triangledown}$ and $\mathsf{bk}^{\triangledown}$ associated to y. Formally, the setup protocol in Figure 3 shows the initialization of these dictionaries.

Queries. Queries are similar to the plaintext case. Apart from the addresses, the client also sends ak and bk retrieved from $\gamma[q]$ to the server. Using bk, the server decrypts all $b = (\mathsf{chd}_0, \mathsf{chd}_1)$ and traverses the

[5] For example, a ciphertext for message m with randomness r can be computed as $c = (r, \mathsf{PRF}(K, r) \oplus m)$, where PRF, modeling a random oracle, is a pseudorandom function with secret key K. In practice, one may substitute PRF with an HMAC.

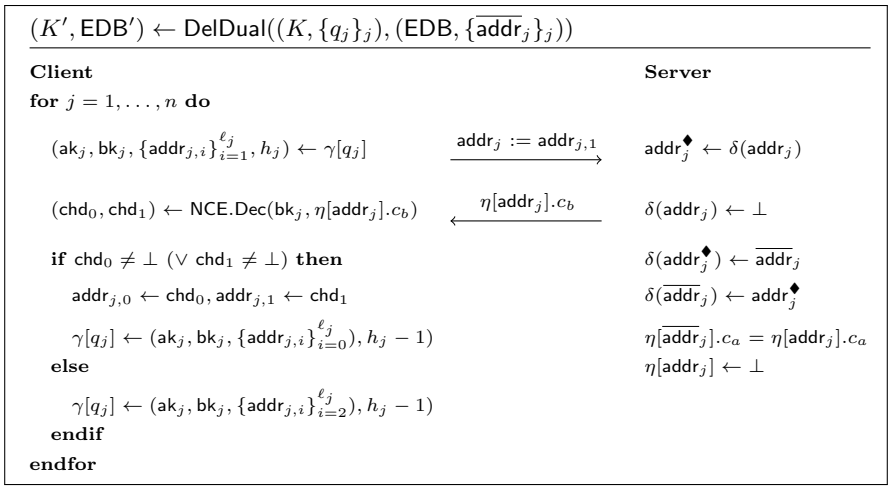

Fig. 5. Dual deletion protocol of encrypted cascaded triangles

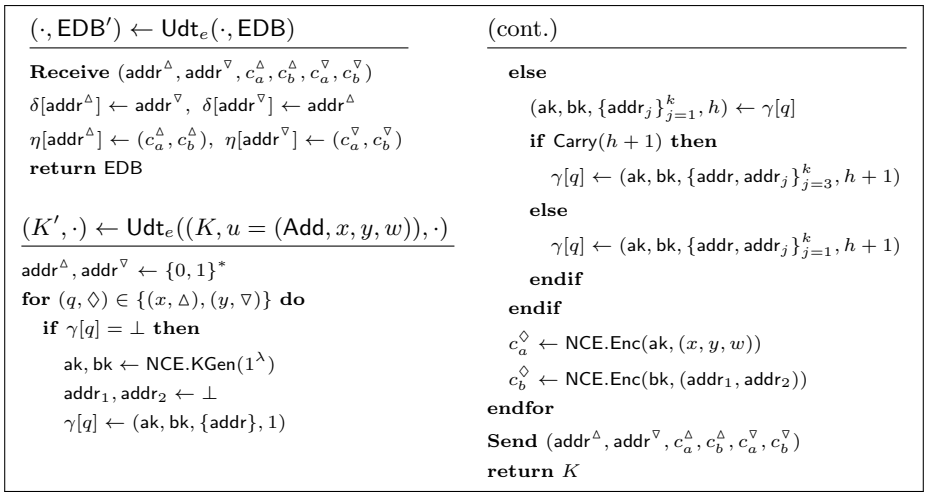

Fig. 6. Addition protocol of encrypted cascaded triangles

sub-trees. Using ak, it decrypts all $a = (x, y, w)$ which are returned as query results. The server also returns the previous query results $\hat{R}$ stored in $\hat{G}$.

As mentioned before, for the efficiency of subsequent queries, the server should remove revealed entries from η and add them to the plaintext subgraph $\hat{G}$. Thus, the client and the server cooperates to delete the set $(q, *, *)$ or $(*, q, *)$ from η. This conceptually removes the set from the encrypted subgraph $\tilde{G}$. Finally, the server adds the set to $\hat{G}$. Formally, the query protocol is shown in Figure 4, which utilizes the subroutine Trav and subprotocol $\mathsf{DelDual}$ shown in Figure 3 and 5 respectively.

Add. Instead of sending $(a^{\triangle}, b^{\triangle})$ in the clear, the client sends their ciphertexts $(c_a^{\triangle}, c_b^{\triangle})$, encrypted under $\mathsf{ak}^{\triangle}$ and $\mathsf{bk}^{\triangle}$ retrieved from $\gamma[x]$, to the server respectively. In the case where $\gamma[x] = \bot$, the client generates new secret keys $\mathsf{ak}^{\triangle}$ and $\mathsf{bk}^{\triangle}$ using the key generation algorithm of the non-committing encryption scheme. Sending $(a^{\triangledown}, b^{\triangledown})$ requires a similar treatment. Figure 6 formally describes the addition protocol.

Delete. Deletion of $(x, *, *)$ is almost identical to querying x, except that the client does not send out $\mathsf{ak}^{\triangle}$. Instead, the server returns all $c_a^{\triangle}$ so that the client can decrypt them locally. After obtaining all the

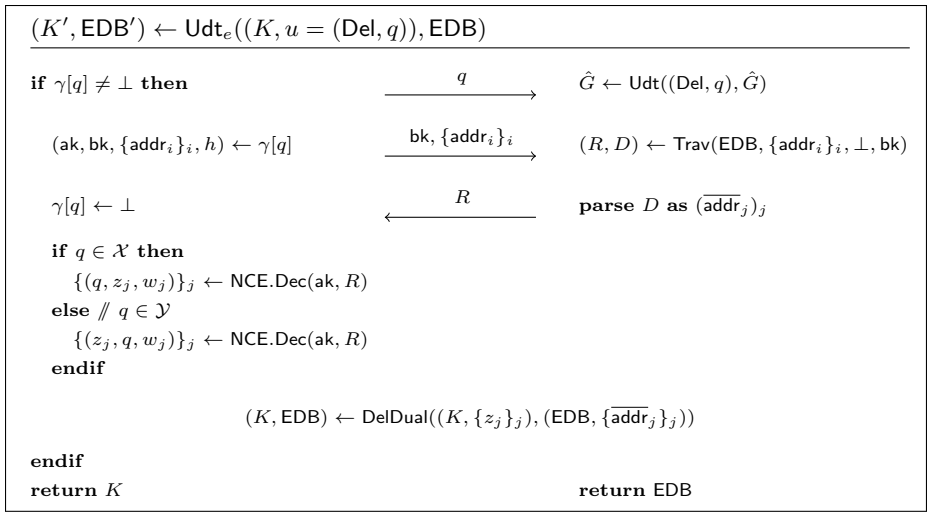

Fig. 7. Deletion protocol of encrypted cascaded triangles

edges (x, y, w), the client and the server cooperate to delete $(x, *, *)$ from η as in the query algorithm. This conceptually removes $(x, *, *)$ from the encrypted subgraph $\tilde{G}$. Finally, the server also removes $(x, *, *)$ from the plaintext subgraph $\hat{G}$. Deletion of $(*, y, *)$ is done similarly. Figure 7 shows the deletion protocol, which also utilizes the subroutines Trav and $\mathsf{DelDual}$ in Figure 3 and 5 respectively.

5.3 Analysis

Storage Cost. For each x, if $n_x = |(x, *, *) \cap \tilde{G}| > 0$, the client stores two λ-bit keys of non-committing encryption, one $\lceil \lg n_x \rceil$-trit string, and at most $\lceil \lg n_x \rceil + 1$ λ-bit addresses. Similar storage is required for each y. In the extreme case where the client adds all possible data and never queries or deletes, the storage cost is $O(\mathsf{poly}(\lambda) \cdot (|\mathcal{X}| \lg |\mathcal{Y}| + |\mathcal{Y}| \lg |\mathcal{X}|))$. However, querying and deleting x (or y) removes $(x, *, *)$ (or $(*, y, *)$) from $\tilde{G}$, which waives the client local storage for x (or y). Thus, the storage of a reasonable client would be much smaller.

The storage cost of the server is linear in the number of edges (x, y, w) added to the server, which is optimal. Furthermore, if an edge (x, y, w) is revealed due to a previous query, it is stored in plaintext instead of ciphertext, where the former is much better in terms of locality [CT14].

Computation and Communication Cost. Both the client and the server perform essentially no work during setup: They just initialize empty dictionaries.

During a query on q, the client first looks up its dictionary $\gamma[q]$, which consists of two λ-bit keys of NCE, one $\lceil \lg n_q \rceil$-trit string, and at most $(\lceil \lg n_q \rceil + 1)$ λ-bit addresses. The query q, the keys, and the addresses are sent to the server. The server traverses the set $(x, *, *)$ if $q = x \in \mathcal{X}$, or the set $(*, y, *)$ if $q = y \in \mathcal{Y}$. In the former, the server needs to perform $O(|(x, *, *) \cap \tilde{G}|)$ decryption, and execute $\mathsf{Qry}(q, \hat{G})$ which takes time $|(x, *, *) \cap \hat{G}|$. It then returns the query result of size $|(x, *, *)| = |(x, *, *) \cap \tilde{G}| + |(x, *, *) \cap \hat{G}|$ to the client. The client looks up and sends $|(x, *, *) \cap \tilde{G}|$ addresses to the server, which performs the same amount of I/O tasks, and returns $O(|(x, *, *) \cap \tilde{G}|)$ ciphertexts to the client. The client finalizes by performing $O(|(x, *, *) \cap \tilde{G}|)$ decryption and the same amount of I/O tasks. Overall, both computation and communication complexities for both the client and the server are in the order of $O(\mathsf{poly}(\lambda) \cdot |(x, *, *)|)$, which is optimal for the server. In contrast to the presentation in the formal construction, the number of rounds can be compressed into 4.

Since deletion is almost identical to querying except for local decryption, their overall computation and communication complexities are identical.

For addition, the client performs a constant amount of I/O tasks to update γ, while sending 2 λ-bit addresses and 4 ciphertexts to the server. The server simply writes the ciphertexts to the specified addresses.

Therefore, the overall computation, round, and communication complexities for both the client and the server are constant, which is again optimal.

Note that our scheme supports batch operations (querying, addition, and deletion) straightforwardly. In such case, the computation and communication complexities increase linearly while the round complexity remains unchanged.

Security. We prove that our scheme is secure against adaptive chosen query attack with very minimal leakage. In particular, our scheme achieves $(\mathcal{U}_1, 1)$-forward privacy, where $\mathcal{U}_1$ is defined in Table 1. We begin by defining the leakage functions. For setup, $\mathcal{L}_u$ only leaks the sizes of the spaces, *i.e.*, $|\mathcal{X}|$, $|\mathcal{Y}|$, and $|\mathcal{W}|$. For addition, $\mathcal{L}_u$ only leaks the update type and the time $\mathsf{Time}(x, y)$ of the addition as the new addresses are truly random and the ciphertexts can be simulated by the simulator of the non-committing encryption scheme. For deletion of $(x, *, *)$, $\mathcal{L}_u$ leaks the update type, x, and the time $\mathsf{Time}(x, y)$ when each of the edges $(x, y, w) \in (x, *, *)$ is added. It also leaks, for each y such that $(x, y, w) \in (x, *, *)$, the time $\mathsf{Time}(*, y)$ when the last edge in $(*, y, *)$ is added. Leakage for deleting $(*, y, *)$ is defined similarly. Finally, for queries on q, $\mathcal{L}_q$ leaks all information leaked by $\mathcal{L}_u$ upon deletion of $(x, *, *)$ if $q = x \in \mathcal{X}$, or $(*, y, *)$ if $q = y \in \mathcal{Y}$, and the access patterns $\mathsf{AP}_t(q)$ of q, assuming it is sorted by the time each response is added. Formally, we define:

- $\mathcal{L}_e(\mathcal{G}) = (|\mathcal{X}|, |\mathcal{Y}|, |\mathcal{W}|)$
- $\mathcal{L}_u(\mathsf{Add}, x, y, w) = (\mathsf{Add}, \mathsf{Time}(x, y))$
- $\mathcal{L}_u(\mathsf{Del}, x) = (\mathsf{Del}, x, \{(\mathsf{Time}(x, y), \mathsf{Time}(*, y)) : (x, y, \cdot) \in (x, *, *)\})$
- $\mathcal{L}_u(\mathsf{Del}, y) = (\mathsf{Del}, y, \{(\mathsf{Time}(x, y), \mathsf{Time}(x, *)) : (x, y, \cdot) \in (*, y, *)\})$
- $\mathcal{L}_q(x) = (x, \mathsf{AP}_t(x), \{(\mathsf{Time}(x, y), \mathsf{Time}(*, y)) : (x, y, \cdot) \in (x, *, *)\})$
- $\mathcal{L}_q(y) = (y, \mathsf{AP}_t(y), \{(\mathsf{Time}(x, y), \mathsf{Time}(x, *)) : (x, y, \cdot) \in (*, y, *)\})$

The simulation is sketched as follows. The simulator first initializes empty dictionaries δ and η, and the empty plaintext graph $\hat{G}$. It maintains a table T mapping time t to time-address tuples $((t_0, \mathsf{addr}), (t_1, \mathsf{addr}^\nabla)$.

For addition at time t, it samples random addresses addr and addr^∇, and registers them in δ. It sets $T[t] \leftarrow ((t, \mathsf{addr}), (t, \mathsf{addr}^\nabla))$. It simulates the ciphertexts in $\eta[\mathsf{addr}]$ and $\eta[\mathsf{addr}^\nabla]$ using the simulator of NCE.

For deletion of $(x, *, *)$, it is given x, which allows it to delete $(x, *, *)$ from $\hat{G}$. It is also given the time $\mathsf{Time}(x, y)$ when each of the edges $(x, y, w) \in (x, *, *)$ is added, and for each y such that $(x, y, w) \in (x, *, *)$ the time $\mathsf{Time}(*, y)$ when the last edge in $(*, y, *)$ is added. It recalls from the table T all addr and addr^∇ pairs which are created at time $\mathsf{Time}(x, y)$ and $\mathsf{Time}(*, y)$. With the knowledge of these addresses, the simulator can maintain δ and η as in the real scheme. It must also output simulated bk and explain the ciphertexts c_b encrypting the addresses. To achieve this, the simulator passes the ciphertexts and the corresponding addresses to the simulator of the non-committing encryption scheme, where the latter outputs the simulated bk. Deletion of $(*, y, *)$ is simulated similarly.

Queries are simulated almost identically as in the simulation of deletions, except that the simulator must now also output simulated ak and explain the ciphertexts c_a encrypting the query result. Similar to the above, this can be done by calling the simulator of the non-committing encryption scheme.

Theorem 2. *Assume that* $\mathsf{NCE}.(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *is a symmetric-key non-committing encryption scheme with message space* $\{0,1\}^{\max(\lg|\mathcal{X}|+\lg|\mathcal{Y}|+\lg|\mathcal{W}|, 2\lambda)}$*, the above construction is* $(\mathcal{L}_e, \mathcal{L}_q, \mathcal{L}_u)$*-CQA2 secure. Furthermore, the above construction is* $(\mathcal{U}_1, 1)$*-forward private, where* $\mathcal{U}_1$ *is defined in Table 1.*

Appendix B.2 gives the proof of Theorem 2.

References

[BHJP14] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. A Survey of Provably Secure Searchable Encryption. *ACM Comput. Surv.*, 47(2):18:1–18:51, August 2014.

[Bos16] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1143–1154, 2016.

[CCC+16] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Cryptography for parallel RAM from indistinguishability obfuscation. In Madhu Sudan, editor, *ITCS 2016: 7th Innovations in Theoretical Computer Science*, pages 179–190, Cambridge, MA, USA, January 14–16, 2016. Association for Computing Machinery.

[CFGN96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *28th Annual ACM Symposium on Theory of Computing*, pages 639–648, Philadephia, PA, USA, May 22–24, 1996. ACM Press.

[CGKO06] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 06: 13th Conference on Computer and Communications Security*, pages 79–88, Alexandria, Virginia, USA, October 30 – November 3, 2006. ACM Press.

[CGKO11] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[CK10] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In Masayuki Abe, editor, *Advances in Cryptology – ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594, Singapore, December 5–9, 2010. Springer, Heidelberg, Germany.

[CT14] David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 351–368, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.

[GMP16] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part III*, volume 9816 of *Lecture Notes in Computer Science*, pages 563–592, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany.

[HK14] Florian Hahn and Florian Kerschbaum. Searchable encryption with secure and efficient updates. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14: 21st Conference on Computer and Communications Security*, pages 310–320, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *ISOC Network and Distributed System Security Symposium – NDSS 2012*, San Diego, CA, USA, February 5–8, 2012. The Internet Society.

[KP13] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In Ahmad-Reza Sadeghi, editor, *FC 2013: 17th International Conference on Financial Cryptography and Data Security*, volume 7859 of *Lecture Notes in Computer Science*, pages 258–274, Okinawa, Japan, April 1–5, 2013. Springer, Heidelberg, Germany.

[KPR12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12: 19th Conference on Computer and Communications Security*, pages 965–976, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

[LC15] Russell W. F. Lai and Sherman S. M. Chow. Structured encryption with non-interactive updates and parallel traversal. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015*, pages 776–777, 2015.

[LC16] Russell W. F. Lai and Sherman S. M. Chow. Parallel and dynamic structured encryption. In *SECURECOMM 2016*, 2016. to appear.

[LZWT14] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 265:176–188, 2014.

[RG15] Panagiotis Rizomiliotis and Stefanos Gritzalis. ORAM based forward privacy preserving dynamic searchable symmetric encryption schemes. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop, CCSW 2015, Denver, Colorado, USA, October 16, 2015*, pages 65–76, 2015.

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *ISOC Network and Distributed System Security Symposium – NDSS 2014*, San Diego, CA, USA, February 23–26, 2014. The Internet Society.

[SWP00] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55, Oakland, CA, USA, May 2000. IEEE Computer Society Press.

[ZKP16] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 707–720, 2016.

A Preliminary

A.1 Symmetric Key Non-Committing Encryption

Non-committing encryption schemes are originally considered in the public-key setting [CFGN96]. The literature of SSE used the following simplest symmetric key encryption of the message m, *i.e.*, $c = (F(K, r) \oplus m, r)$, where F is a pseudorandom function modeled as a random oracle, K is the secret key, and r is the encryption randomness. It is non-committing in the sense that, a simulator can simulate a ciphertext c by truly random strings (s, r). It can later open the ciphertext to an arbitrary message m encrypted under key K by programming the random oracle F such that $F(K, r) = s \oplus m$. To keep our construction and security proof clean, we abstract the property out as a building block.

Definition 4. *A symmetric key encryption scheme* $\mathsf{NCE}.(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *is said to be non-committing, if there exists a simulator* $\mathcal{S}$ *such that the following properties hold:*

- *Efficiency:* KGen*,* Enc*,* Dec*, and* $\mathcal{S}$ *are all PPT algorithms.*
- *Correctness: For any message* $m \in \mathcal{M}$*,*

$$\Pr[\mathsf{Dec}(\mathsf{sk}, c) = m : \mathsf{sk} \leftarrow \mathsf{KGen}(1^\lambda); c \leftarrow \mathsf{Enc}(\mathsf{sk}, m)] = 1.$$

- *Simulatability: For all* $\ell \in \mathbb{N}$*, the following distributions defined by the simulator* $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$ *are computationally indistinguishable:*

$$\{(\mathsf{sk}, (m_i)_{i=1}^{\ell}, (r_i)_{i=0}^{\ell}) : \mathsf{tk} \leftarrow \mathcal{S}_1(1^\lambda); c_i \leftarrow \mathcal{S}_2(\mathsf{tk}) \ \forall \ i \in [\ell]; \\ (\mathsf{sk}, (r_i)_{i=0}^{\ell}) \leftarrow \mathcal{S}_3(\mathsf{tk}, (m_i)_{i=1}^{\ell})\}$$

and

$$\{(\mathsf{sk}, (m_i)_{i=1}^{\ell}, (r_i)_{i=0}^{\ell}) : \mathsf{sk} \leftarrow \mathsf{KGen}(1^\lambda; r_0); c_i \leftarrow \mathsf{Enc}(\mathsf{sk}, m_i; r_i) \ \forall \ i \in [\ell]\}$$

We remark that simulatability implies semantic security.

A.2 Discussion on Our Data Representation

Here we give a more detailed explanation of our choice of data representation. We consider labeled bipartite graphs G defined by the spaces $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{W}$, where $\mathcal{X}$ and $\mathcal{Y}$ are disjoint, *i.e.*, $\mathcal{X} \cap \mathcal{Y} = \phi$. We denote by $\mathcal{G} = \mathcal{G}(\mathcal{X}, \mathcal{Y}, \mathcal{W})$ a set of labeled bipartite graphs specified by these sets. Labeled bipartite graphs are graphs in which nodes can be partitioned into disjoint subsets $X \subseteq \mathcal{X}$ and $Y \subseteq \mathcal{Y}$, and edges with labels from the subset $W \subseteq \mathcal{W}$ can never connect two nodes from the same partition. Let $G \in \mathcal{G}$ be such a graph. An edge in G can be uniquely represented by the tuple $(x, y, w) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{W}$. Thus, G can formally be defined as the collection of all these tuples.

Consider the neighbor query function Qry which maps a node $q = x$ (or $q = y$) and a graph G to a set of all edges connecting node x, denoted by $(x, *, *)$ (or a set of all edges connecting node y, denoted by $(*, y, *)$). Similarly, we also denote as $(x, y, *)$ set of edges in the form of $(x, y, \cdot)$, which should be a singleton. One can also consider an edge query function Qry' which maps a tuple (x, y) and a graph G to a bit b, such that $b = 1$ if and only if $(x, y, *) \subseteq G$.

Neighbor queries over one of the partitions already naturally capture a wide range of applications such as (conjunctive) keyword search, neighbor, and adjacency queries over graphs, any queries over binary relations. Using techniques involving the semi-private data [CK10], compositions of these queries are also supported. More generally, neighbor queries over the entire bipartite graph (both $\mathcal{X}$ and $\mathcal{Y}$) enable interesting bi-directional searches.

The update function Udt maps an update u and a graph G to a new graph G'. The update $u = (\mathsf{Op}, \cdot, \cdot, \cdot)$ where $\mathsf{Op} = \mathsf{Add}$ or $\mathsf{Op} = \mathsf{Del}$ takes one of the following forms:

1. Edge Addition: $u = (\mathsf{Add}, x, y, w)$ adds the edge (x, y, w) to G.

2. Node Addition: $u = (\mathsf{Add}, \mathbf{x}, y, \mathbf{w})$ (or $u = (\mathsf{Add}, x, \mathbf{y}, \mathbf{w})$) adds the edges (x_i, y, w_i) where $x_i \in \mathbf{x}$ (or (x, y_i, w_i) where $y_i \in \mathbf{y}$) and $w_i \in \mathbf{w}$ to G, where node y (or x) originally does not exists in G.
3. Edge Deletion: $u = (\mathsf{Del}, x, y)$ deletes the edge of the form $(x, y, \cdot)$ from G.
4. Node Deletion: $u = (\mathsf{Del}, q)$ deletes the set of edges $(q, *, *)$ (or $(*, q, *)$) from G. Since $\mathcal{X}$ and $\mathcal{Y}$ are disjoint, there is no ambiguity.

In this work, we focus on DSSE constructions which support neighbor query function Qry, edge additions, and node deletions defined above.

Remarks on the different forms of updates are in order. First, for additions, notice that edge additions enable fine-grained modification of existing nodes x and y. Node additions are however weaker, or easier to support, than edge additions. To see why this is the case, consider a DSSE scheme supporting edge additions: Intuitively, the server needs to somehow link the newly added edge (x, y, w) to some previously added edge (x, y', w') for the same node x. Then when the user queries node x, the server can retrieve all such linkage and return all edges in the set $(x, *, *)$. To provide such linkage, the user might need to reveal some secret information (such as x itself) to the server during the addition of edges. The user can, however, prepare such linkage locally for node additions in one shot, and thus reveal less information to the server.

Next, for the deletions, notice that edge deletions require the user to know the existence of the edge connecting some nodes x and y in the graph. On one hand, they enable fine-grained modification similar to edge additions. On the other hand, they are not too useful when the user wishes to delete all edges connected to some node x (or y) while not remembering them. Recall that node deletions allow the user to delete all edges in the set $(x, *, *)$ (or $(*, y, *)$), essentially removing the node x (or y) entirely from the graph. Similar to the discussion above, this intuitively requires the user to provide sufficient secret information during the additions, such that the server can somehow link (x, y, w) with (x, y', w') to intelligently delete them later.

A.3 Why Forward Privacy

The CQA2 definition itself does not say much about the actual security level of the DSSE scheme due to the flexibility in defining the leakage functions. In particular, consider the following attack scenario of DSSE: The client cooperates with the server for querying on x. The former later adds a new edge (x, y, w). Using the knowledge learnt from the previous query on x, the server might be able to tell that this new edge is connecting a previously queried node to some node y (although the server may not know the exact value of x and y). This affects the privacy of newly added data since the client only wanted to delegate the query power to the server at an earlier time. Indeed, there are practical attacks which exploit such information [ZKP16]. The security property which prevents the above privacy breach is known as forward privacy. Forward privacy for DSSE was advocated by Stefanov *et al.* [SPS14] without an explicit formal definition. Rizomiliotis and Gritzalis [RG15] gave another construction but still did not formally define the notion. A formal definition was recently given by Bost [Bos16], yet it is specific to a certain type of constructions, and is weaker in the sense that there might exist schemes which satisfy the definition, yet updates to the same file might be linkable. (See the discussion in Section 3.2 for details.)

Earlier works [SPS14, RG15] claimed that forward privacy was captured by their respective leakage function $\mathcal{L}_q(q)$ on a query q occurred at time t, which leaks only data added and deleted in the past, *i.e.*, $\mathsf{AP}_t(q)$. However, capturing forward privacy in this way is problematic: The simulator is given $\mathcal{L}_q(q)$ when a query q is chosen by the environment at some time t. It is sufficient for $\mathcal{L}_q(q)$ to leak access pattern of q up to time t such that the simulator of a non-forward-private DSSE can simulate the query results. Indeed, most if not all existing (both forward and non-forward-private) DSSE schemes have the leakage $\mathcal{L}_q(q)$ defined as $(q, \mathsf{AP}_t(q))$. The correct way is to capture forward privacy in $\mathcal{L}_u$, which is also the case in the definition by Bost [Bos16].

B Security Proofs

B.1 Proof of Theorem 1 (Security for Generic Construction)

Proof. Since $\mathcal{E}$ is $(\tilde{\mathcal{L}}_q, \tilde{\mathcal{L}}_u)$-CQA2 secure, there exists a simulator $\mathcal{S}_{\mathcal{E}}$ which simulates queries and updates of $\mathcal{E}$ when given the leakage defined by $\tilde{\mathcal{L}}_q$ and $\tilde{\mathcal{L}}_u$. Using $\mathcal{S}_{\mathcal{E}}$, we construct a simulator $\mathcal{S}$ as follows: $\mathcal{S}$ first runs $\mathcal{S}_{\mathcal{E}}$ to simulate $\tilde{\mathsf{EDB}}$. It also initiates empty dictionary γ and empty graph $\hat{G}$.

Upon update $u = (\mathsf{Add}, x, y, w)$, the simulator $\mathcal{S}$ receives

$$\mathcal{L}_u(\mathsf{Add}, x, y, w) = (\tilde{x}, \tilde{\mathcal{L}}_u(\mathsf{Add}, \tilde{x}, y, w))$$

for a uniformly random $\tilde{x}$, which is computationally indistinguishable from the PRF output $F(K_x, i)$ computed in the real scheme by the pseudorandomness of F. Thus, $\mathcal{S}$ passes $\tilde{\mathcal{L}}_u(\mathsf{Add}, \tilde{x}, y, w)$ to $\mathcal{S}_{\mathcal{E}}$ which outputs a simulated τ_u^+.

Upon update $u = (\mathsf{Del}, x)$, the simulator $\mathcal{S}$ is given

$$\mathcal{L}_u(\mathsf{Del}, x) = (x, \{\tilde{\mathcal{L}}_u(\mathsf{Del}, \tilde{x}_i)\}_{i=1}^{c_x}).$$

It passes $\tilde{\mathcal{L}}_u(\mathsf{Del}, \tilde{x}_i)$ to $\mathcal{S}_{\mathcal{E}}$ which outputs a simulated $\tilde{\tau}_i^-$ for each i. Lastly, for update $u = (\mathsf{Del}, y)$, the simulator $\mathcal{S}$ is given

$$\mathcal{L}_u(\mathsf{Del}, y) = (y, \tilde{\mathcal{L}}_u(\mathsf{Del}, y)).$$

It then passes $\tilde{\mathcal{L}}_u(\mathsf{Del}, y)$ to $\mathcal{S}_{\mathcal{E}}$ which outputs a simulated $\tilde{\tau}^-$.

To simulate the query on x, the simulator $\mathcal{S}$ is given

$$\mathcal{L}_q(x) = (x, \mathsf{AP}_t(x), \{\tilde{\mathcal{L}}_q(\tilde{x}_i), \tilde{\mathcal{L}}_u(\mathsf{Del}, \tilde{x}_i)\}_{i=1}^{c_x}).$$

It passes for each i the leakages $\tilde{\mathcal{L}}_q(\tilde{x}_i)$ and $\tilde{\mathcal{L}}_u(\mathsf{Del}, \tilde{x}_i)$ to $\mathcal{S}_{\mathcal{E}}$, which returns the corresponding simulated query token $\tilde{\tau}_i$ and simulated update token $\tilde{\tau}_i^-$.

Suppose there exists an environment $\mathcal{Z}$ which distinguishes the simulation from the real scheme, then we can construct another environment which acts as $\mathcal{S}$ and distinguish the simulated base scheme from the real base scheme.

Furthermore, since for any $x_0, x_1 \in \mathcal{X}$, $\mathcal{L}_u(\mathsf{Add}, x_0, y, w)$ and $\mathcal{L}_u(\mathsf{Add}, x_1, y, w)$ have exactly the same distribution, we conclude that the above construction is $(\mathcal{U}_1, p'_1)$-forward private. □

B.2 Proof of Theorem 2 (Security for Concrete Construction)

Proof. We prove by constructing a simulator $\mathcal{S}$ which simulates a real-world adversary when given leakage functions $\mathcal{L}_q$ and $\mathcal{L}_u$. By the simulatability of NCE, there exists a PPT simulator $\mathcal{S}^{\mathsf{NCE}} = (\mathcal{S}_1^{\mathsf{NCE}}, \mathcal{S}_2^{\mathsf{NCE}}, \mathcal{S}_3^{\mathsf{NCE}})$.

At the start, $\mathcal{S}$ is given the size bounds $|\mathcal{X}|$, $|\mathcal{Y}|$, and $|\mathcal{W}|$. $\mathcal{S}$ simulates setup by releasing empty dictionaries δ and η, and empty graph $\hat{G}$. Internally, it maintains a table T which maps time t to addresses $((t_0, \mathsf{addr}), (t_1, \mathsf{addr}^\nabla))$, and initializes the simulator $\mathcal{S}^{\mathsf{NCE}}$ of the non-committing encryption by $\mathsf{tk} \leftarrow \mathcal{S}_1^{\mathsf{NCE}}(\lambda)$.

At time $\mathsf{Time}(x, y)$, when the client submits an update $u = (\mathsf{Add}, x, y, w)$ to the ideal functionality $\mathcal{F}$, $\mathcal{S}$ is notified of

$$\mathcal{L}_u(\mathsf{Add}, x, y, w) = (\mathsf{Add}, \mathsf{Time}(x, y)).$$

It simulates c_a, c_a^∇, c_b and c_b^∇ by calling $\mathcal{S}_2^{\mathsf{NCE}}(\mathsf{tk})$. It samples $\mathsf{addr}, \mathsf{addr}^\nabla \leftarrow \{0,1\}^\lambda$, and set $\delta[\mathsf{addr}] \leftarrow \mathsf{addr}^\nabla$, $\delta[\mathsf{addr}^\nabla] \leftarrow \mathsf{addr}$, $\eta[\mathsf{addr}] = (c_a, c_b)$, and $\eta[\mathsf{addr}^\nabla] = (c_a^\nabla, c_b^\nabla)$. Internally, it sets $T[t] \leftarrow ((t, \mathsf{addr}), (t, \mathsf{addr}^\nabla))$.

When the client submits an update $u = (\mathsf{Del}, x)$ to $\mathcal{F}$, $\mathcal{S}$ is notified of

$$\mathcal{L}_u(\mathsf{Del}, x) = (\mathsf{Del}, x, \{(\mathsf{Time}(x, y), \mathsf{Time}(*, y)) : (x, y, \cdot) \in (x, *, *)\}).$$

$\mathcal{S}$ performs the following:

- It removes $(x, *, *)$ from $\hat{G}$.

– For each $t \in \{\mathsf{Time}(x,y) : (x,y,\cdot) \in (x,*,*)\}$, it retrieves from $T[t]$ the tuples $((t_0,\mathsf{addr}),(t_1,\mathsf{addr}^{\triangledown}))$. Denote the set of all addr by $(\mathsf{addr}_j)_j$.
– It arranges the addresses $(\mathsf{addr}_j)_j$ in a unique ordering of cascaded triangles according to the times t_0.
– It explains the ciphertexts c_b in $\eta[\mathsf{addr}]$ according to the structure of the cascaded triangles by $(\mathsf{bk},(r_j)_j) \leftarrow \mathcal{S}_3^{\mathsf{NCE}}(\mathsf{tk},(\mathsf{addr}_j)_j)$.
– For each $(t,t') \in \{(\mathsf{Time}(x,y),\mathsf{Time}(*,y)) : (x,y,\cdot) \in (x,*,*)\}$, it performs the following:
  • It retrieves $((t_0,\mathsf{addr}),(t_1,\mathsf{addr}^{\triangledown})) \leftarrow T[t]$ and $((t'_0,\mathsf{addr}'),(t'_1,\mathsf{addr}^{\blacktriangledown})) \leftarrow T[t']$.
  • It sets $T[t] \leftarrow \perp$ and $T[t'] \leftarrow ((t'_0,\mathsf{addr}'),(t_1,\mathsf{addr}^{\triangledown}))$.
  • It sets $\delta[\mathsf{addr}] \leftarrow \perp$ and $\eta[\mathsf{addr}] \leftarrow \perp$ (as in Trav).
  • It sets $\delta[\mathsf{addr}^{\blacktriangledown}] \leftarrow \perp$, $\delta[\mathsf{addr}'] \leftarrow \mathsf{addr}^{\triangledown}$, $\delta[\mathsf{addr}^{\triangledown}] \leftarrow \mathsf{addr}'$, $\eta[\mathsf{addr}^{\triangledown}].c_a \leftarrow \eta[\mathsf{addr}^{\blacktriangledown}].c_a$, and $\eta[\mathsf{addr}^{\blacktriangledown}] \leftarrow \perp$ (as in $\mathsf{DelDual}$).

The update $u = (\mathsf{Del},y)$ is simulated similarly. When the client submits an update $u = (\mathsf{Del},y)$ to $\mathcal{F}$, $\mathcal{S}$ is notified of

$$\mathcal{L}_u(\mathsf{Del},y) = (\mathsf{Del},y,\{(\mathsf{Time}(x,y),\mathsf{Time}(x,*)) : (x,y,\cdot) \in (*,y,*)\}).$$

$\mathcal{S}$ performs the following:

– It removes $(*,y,*)$ from $\hat{G}$.
– For each $t \in \{\mathsf{Time}(x,y) : (x,y,\cdot) \in (*,y,*)\}$, it retrieves from $T[t]$ the tuples $((t_0,\mathsf{addr}),(t_1,\mathsf{addr}^{\triangledown}))$. Denote the set of all $\mathsf{addr}^{\triangledown}$ by $(\mathsf{addr}_i^{\triangledown})_i$.
– It arranges the addresses $(\mathsf{addr}_i^{\triangledown})_i$ in a unique ordering of cascaded triangles according to the times t_1.
– It explains the ciphertexts c_b in $\eta[\mathsf{addr}^{\triangledown}]$ according to the structure of the cascaded triangles by $(\mathsf{bk}^{\triangledown},(r_i)_i) \leftarrow \mathcal{S}_3^{\mathsf{NCE}}(\mathsf{tk},(\mathsf{addr}_i^{\triangledown})_i)$.
– For each $(t,t') \in \{(\mathsf{Time}(x,y),\mathsf{Time}(*,y)) : (x,y,\cdot) \in (*,y,*)\}$, it performs the following:
  • It retrieves $((t_0,\mathsf{addr}),(t_1,\mathsf{addr}^{\triangledown})) \leftarrow T[t]$ and $((t'_0,\mathsf{addr}'),(t'_1,\mathsf{addr}^{\blacktriangledown})) \leftarrow T[t']$.
  • It sets $T[t] \leftarrow \perp$ and $T[t'] \leftarrow ((t_0,\mathsf{addr}),(t'_1,\mathsf{addr}^{\blacktriangledown}))$.
  • It sets $\delta[\mathsf{addr}^{\triangledown}] \leftarrow \perp$ and $\eta[\mathsf{addr}^{\triangledown}] \leftarrow \perp$ (as in Trav).
  • It sets $\delta[\mathsf{addr}'] \leftarrow \perp$, $\delta[\mathsf{addr}^{\blacktriangledown}] \leftarrow \mathsf{addr}$, $\delta[\mathsf{addr}] \leftarrow \mathsf{addr}^{\blacktriangledown}$, $\eta[\mathsf{addr}].c_a \leftarrow \eta[\mathsf{addr}'].c_a$, and $\eta[\mathsf{addr}'] \leftarrow \perp$ (as in $\mathsf{DelDual}$).

When the client submits a query x to $\mathcal{F}$, $\mathcal{S}$ is notified of

$$\mathcal{L}_q(x) = (x,\mathsf{AP}_t(x),\{(\mathsf{Time}(x,y),\mathsf{Time}(*,y)) : (x,y,\cdot) \in (x,*,*)\}).$$

$\mathcal{S}$ performs the following:

– It splits $\mathsf{AP}_t(x)$ into $\tilde{R}$ and $\hat{R}$, where $\hat{R} = \mathsf{AP}_t(x) \cap \hat{G}$.
– It merges $\tilde{R}$ to $\hat{G}$.
– For each $t \in \{\mathsf{Time}(x,y) : (x,y,\cdot) \in (x,*,*)\}$, it retrieves from $T[t]$ the tuples $((t_0,\mathsf{addr}),(t_1,\mathsf{addr}^{\triangledown}))$. Denote the set of all addr by $(\mathsf{addr}_j)_j$.
– It arranges the $\tilde{R}$ in a unique ordering of cascaded triangles according to the times t_0.
– It arranges the addresses $(\mathsf{addr}_j)_j$ in a unique ordering of cascaded triangles according to the time t_0.
– It explains the ciphertexts c_a and c_b in $\eta[\mathsf{addr}]$ according to the structure of the cascaded triangles by $(\mathsf{ak},(r_j)_j) \leftarrow \mathcal{S}_3^{\mathsf{NCE}}(\mathsf{tk},\tilde{R})$ and $(\mathsf{bk},(r'_j)_j) \leftarrow \mathcal{S}_3^{\mathsf{NCE}}(\mathsf{tk},(\mathsf{addr}_j)_j)$.
– For each $(t,t') \in \{(\mathsf{Time}(x,y),\mathsf{Time}(*,y)) : (x,y,\cdot) \in (x,*,*)\}$, it performs the following:
  • It retrieves $((t_0,\mathsf{addr}),(t_1,\mathsf{addr}^{\triangledown})) \leftarrow T[t]$ and $((t'_0,\mathsf{addr}'),(t'_1,\mathsf{addr}^{\blacktriangledown})) \leftarrow T[t']$.
  • It sets $T[t] \leftarrow \perp$ and $T[t'] \leftarrow ((t'_0,\mathsf{addr}'),(t_1,\mathsf{addr}^{\triangledown}))$.
  • It sets $\delta[\mathsf{addr}] \leftarrow \perp$ and $\eta[\mathsf{addr}] \leftarrow \perp$ (as in Trav).
  • It sets $\delta[\mathsf{addr}^{\blacktriangledown}] \leftarrow \perp$, $\delta[\mathsf{addr}'] \leftarrow \mathsf{addr}^{\triangledown}$, $\delta[\mathsf{addr}^{\triangledown}] \leftarrow \mathsf{addr}'$, $\eta[\mathsf{addr}^{\triangledown}].c_a \leftarrow \eta[\mathsf{addr}^{\blacktriangledown}].c_a$, and $\eta[\mathsf{addr}^{\blacktriangledown}] \leftarrow \perp$ (as in $\mathsf{DelDual}$).

Similarly, when the client submits a query y to $\mathcal{F}$, $\mathcal{S}$ is notified of

$$\mathcal{L}_q(y) = (y,\mathsf{AP}_t(y),\{(\mathsf{Time}(x,y),\mathsf{Time}(*,y)) : (x,y,\cdot) \in (*,y,*)\}).$$

$\mathcal{S}$ performs the following:

– It splits $\mathsf{AP}_t(y)$ into $\tilde{R}$ and $\hat{R}$, where $\hat{R} = \mathsf{AP}_t(y) \cap \hat{G}$.
– It merges $\tilde{R}$ to $\hat{G}$.
– For each $t \in \{\mathsf{Time}(x, y) : (x, y, \cdot) \in (*, y, *)\}$, it retrieves from $T[t]$ the tuples $((t_0, \mathsf{addr}), (t_1, \mathsf{addr}^{\triangledown}))$. Denote the set of all $\mathsf{addr}^{\triangledown}$ by $(\mathsf{addr}_j^{\triangledown})_j$.
– It arranges the $\tilde{R}$ in a unique ordering of cascaded triangles according to the times t_1.
– It arranges the addresses $(\mathsf{addr}_j^{\triangledown})_j$ in a unique ordering of cascaded triangles according to the time t_1.
– It explains the ciphertexts $c_a^{\triangledown}$ and $c_b^{\triangledown}$ in $\eta[\mathsf{addr}^{\triangledown}]$ according to the structure of the cascaded triangles by $(\mathsf{ak}^{\triangledown}, (r_j)_j) \leftarrow \mathcal{S}_3^{\mathsf{NCE}}(\mathsf{tk}, \tilde{R})$ and $(\mathsf{bk}^{\triangledown}, (r'_j)_j) \leftarrow \mathcal{S}_3^{\mathsf{NCE}}(\mathsf{tk}, (\mathsf{addr}_j^{\triangledown})_j)$.
– For each $(t, t') \in \{(\mathsf{Time}(x, y), \mathsf{Time}(*, y)) : (x, y, \cdot) \in (*, y, *)\}$, it performs the following:
  • It retrieves $((t_0, \mathsf{addr}), (t_1, \mathsf{addr}^{\triangledown})) \leftarrow T[t]$ and $((t'_0, \mathsf{addr}'), (t'_1, \mathsf{addr}^{\blacktriangledown})) \leftarrow T[t']$.
  • It sets $T[t] \leftarrow \perp$ and $T[t'] \leftarrow ((t_0, \mathsf{addr}), (t'_1, \mathsf{addr}^{\blacktriangledown}))$.
  • It sets $\delta[\mathsf{addr}^{\triangledown}] \leftarrow \perp$ and $\eta[\mathsf{addr}^{\triangledown}] \leftarrow \perp$ (as in Trav).
  • It sets $\delta[\mathsf{addr}'] \leftarrow \perp$, $\delta[\mathsf{addr}^{\blacktriangledown}] \leftarrow \mathsf{addr}$, $\delta[\mathsf{addr}] \leftarrow \mathsf{addr}^{\blacktriangledown}$, $\eta[\mathsf{addr}].c_a \leftarrow \eta[\mathsf{addr}'].c_a$, and $\eta[\mathsf{addr}'] \leftarrow \perp$ (as in $\mathsf{DelDual}$).

By the simulatability of NCE, the simulation above is computationally indistinguishable from the real-world scheme. Furthermore, since $\mathcal{L}_u(\mathsf{Add}, x_0, y_0, w_0)$ and $\mathcal{L}_u(\mathsf{Add}, x_1, y_1, w_1)$ have identical distributions, we conclude that the scheme is $(\mathcal{U}_1, 1)$-forward privacy, where $\mathcal{U}_1$ is defined in Table 1. □

C More on Related Work

The DSSE by Kamara *et al.* [KPR12] is non-interactive and is the first to achieve optimal sequential query and update efficiency. Unfortunately, their scheme uses linked list as the set representation for which traversal and update algorithms are not parallelizable. Moreover, updates in their scheme leak a considerable amount of information.

The first parallel DSSE scheme was proposed by Kamara and Papamanthou [KP13], which uses red-black trees as the set representation. A bit array as long as the keyword space (or the space X in our context) is stored in each tree node to indicate whether any of its children contains a given keyword. Thus, the actual data is only stored at the leaf node of the red-black tree, resulting in a $\log N$ overhead in the query complexity, *i.e.*, the query complexity is $O((m \log N)/p)$ where m and p are the number of matches and CPU respectively.

The DSSE scheme proposed by Stefanov *et al.* [SPS14] uses a novel data structure, consisting of multiple levels of hash chains. As deletion in their scheme is done by inserting dummy "delete node" instead of actually deleting the data, amortized rebuilding are done during queries, which makes their query complexity suffers from an overhead of order roughly $\log^3 N$, *i.e.*, the query complexity is $O((m \log^3 N)/p)$. The crux of their scheme is a complicated interactive database rebuild algorithm which maintains the compactness of the encrypted database while providing strong security. In particular, their scheme is the first providing (half-)forward privacy.

Hahn and Kerschbaum [HK14] does not follow the inverted-index approach. Instead, they encrypt the forward index and create an inverted index for a keyword only after the keyword is searched. This results in a first-time search complexity linear in the size of the database (*i.e.*, $O(N/p)$), while the asymptotic search complexity of issued queries will become optimal. On the other hand, the complexities for adding and removing a file are linear in the size of the inverted index (*i.e.*, $O(|I|/p)$ where I is the inverted index). Thus, as times goes by and as the inverted index grows larger, the efficiency of updates decreases.

Rizomiliotis and Gritzalis [RG15] used ORAM to give the second solution to (half-)forward private DSSE. Their main idea is to refresh the pseudorandom function key for the keyword w each time after it is being queried on.

Bost [Bos16] recently gave the first formal definition of forward security, and proposed a simple construction to achieve it. The main idea is to use a trapdoor permutation to invert the pseudonym of a keyword to give a new pseudonym for each addition. The proposed scheme was constructed in two phases. First, he constructed a scheme which supports only additions but not deletions. Support for deletions is then

added by letting the server maintain two copies of the base scheme: To delete, similar to the scheme by Stefanov *et al.* [SPS14], a dummy "delete node" is added. The query efficiency thus decreases as the number of deletions increases. Moreover, due to the use of trapdoor permutation, the query algorithm is inherently sequential.

Lastly, we note that while the schemes by Stefanov *et al.* [SPS14] and Bost [Bos16] support edge additions and deletions, others [KPR12, KP13, HK14, RG15] support node additions and deletions.