Verifiable and forward private conjunctive keyword search from DIA tree

Laltu Sardar[1], Sushmita Ruj[2]
[1]Indian Statistical Institute, Kolkata, India [*]
[2]University of New South Wales, Sydney, Australia [†]

Abstract

In a *dynamic searchable encryption* (DSE) scheme, a cloud server can search on encrypted data that the client stores and updates from time to time. Due to information leakage during the search and update phase, DSE schemes are prone to file injection attacks. If during document addition, a DSE scheme does not leak any information about the previous search results, the scheme is said to be forward private. A DSE scheme that supports conjunctive keyword search should be forward private. There has been a fair deal of work on designing forward private DSE schemes in the presence of an honest-but-curious cloud server. However, a malicious cloud server might not run the protocol correctly and still want to be undetected. In a verifiable DSE, the cloud server not only returns the result of a search query but also provides proof that the result is computed correctly.

We design a forward private DSE scheme that supports conjunctive keyword search. At the heart of the construction is our proposed data structure called *dynamic interval accumulation tree* (DIA tree). It is an accumulator based authentication tree that efficiently returns both membership and non-membership proofs. Using the DIA tree, we can convert any single keyword forward private DSE scheme to a verifiable forward private DSE scheme that can support conjunctive query as well. Our proposed scheme has the same storage as the base DSE scheme and low computational overhead at the client-side. We have shown the efficiency of our design by comparing it with existing conjunctive DSE schemes. The comparison also shows that our scheme is suitable for practical use.

Keywords— searchable encryption, forward privacy, conjunctive search, verifiability, accumulator, DIA tree, authentication tree

1 Introduction

Sensitive data is often encrypted before storing it in outsourced servers (clouds). This makes searching difficult. In this paper we consider the problem of keyword search. A searchable encryption (SE) scheme allows a server to search on encrypted data and return the results of the search query to the client. In an SE scheme, a data owner outsources encrypted data, together with an encrypted search index, to a cloud. A search query includes an encrypted search trapdoor that allows the cloud to perform search and return the results, without leaking information of the database. A dynamic SE (DSE) scheme not only allows search but also supports updates on the database. DSE schemes are designed depending upon the type of data and query. When the data is a set of documents, each containing a set of keywords, some popular queries over them include single keyword search, conjunctive or boolean search on a set of keywords, etc. In a conjunctive keyword search scheme, given a set of keywords, the cloud returns the set of documents that contains all of them.

In a DSE scheme, updating the database may reveal the relation between the updated set of keywords and the previous search result. In such a scenario, a file injection attack ([1]) can be performed by a curious

[*] Email: laltu.sardar@tcgcrest.org, is presently with IAI, TCG CREST, Kolkata, India

[†] Email: sushmita.ruj@unsw.edu.au

server. A forward private DSE scheme does not leak any information about the previous search results when new documents are added [2].

Moreover, if a curious server becomes malicious, it may not return the actual search result for monetary or other benefits. For example, a verifiable DSE guarantees correctness and completeness of the search result even when the server is malicious. A server not only sends the search result but also a proof that the result is correct. Any conjunctive DSE scheme should be forward private and verifiable.

There are conjunctive SE schemes by Miao et al. [3], Wang et al. [4], Li et al. [5], Azraoui et al. [6] etc. Though the schemes are verifiable, they work only for the static databases. Dynamic conjunctive DSE schemes have been studied in [7], [8], [9]. However, they are either forward private or are verifiable but not both. In the presence of a malicious adversary, we need both the properties to be present in a conjunctive DSE.

In this paper, we study a conjunctive SE scheme with both forward privacy and verifiability. We present a generic scheme that converts any forward private DSE scheme to a verifiable conjunctive DSE scheme preserving its forward privacy. For verifiability, we propose a new cryptographic accumulator called *dynamic interval accumulation tree (DIA tree).*

Cryptographic accumulators are used for proving membership as well as non-membership of elements in a set. When the size of the set is large, proof generation and (or) proof size becomes expensive. Though the existing accumulator scheme like [10] can build an accumulation tree for a static database that can provide the proof of membership as well as non-membership, it is inefficient for a dynamic set. Papamanthou et al. [11] presented a scheme that dealt with the dynamic set that generates membership proofs efficiently. They extended their scheme with an additional authenticated tree that allows non-membership checks. However, this additional structure does not support updates.

In this chapter, we have proposed a **Dynamic Interval Accumulation tree** (DIA tree) that efficiently works for both membership and non-membership witnesses and returns proofs even on large dynamic dataset efficiently. We have used the DIA Tree in our conjunctive DSE scheme for verifiability. Please note that DIA trees are of important interest and can be applied to the other applications not just constructing DSE schemes.

Our contributions In this work, we make the following contributions.

- We propose an accumulator using a new data structure called Dynamic Interval Accumulation tree (DIA tree) that supports efficient proofs of membership and non-membership. To the best of our knowledge, there is no previous scheme in the literature that provides a single authentication data structure supporting both membership and non-membership proofs efficiently together with update support. We provide formal security proof for the accumulator.
- We propose a generic verifiable conjunctive search DSE scheme Blasu that converts any forward private DSE scheme conjunctive without losing its forward privacy property and without using any extra client storage for verifiability. To the best of our knowledge, our proposed scheme is *the first forward private as well as verifiable conjunctive SE scheme in a dynamic setting.* Moreover, we have given security proof of the scheme. We have shown that the proposed scheme Blasu is secure against adaptive chosen query attack.
- We compare our proposed scheme Blasu with the existing schemes and show that the scheme is practical.

1.1 Organization

We summarize our work in the paper as follows. We describe the literature related to our proposed scheme in Section 2. We briefly introduce the required cryptographic tools in Section 3. We propose an authenticated data structure DIA tree in Section 4, together with its security proof. Using the tree, in Section 5, we propose a DSE scheme Blasu that provides verifiability without extra client storage. In Section 6, we compare our scheme Blasu with a few of the existing similar schemes. Finally, in Section 7, we give the conclusion of our work.

2 Related Works

Though the term *Searchable Encryption* was first introduced in the year 2000 for static database, it got popularity when the first SE for dynamic database [12] was published in 2012. Thereafter, while researchers were focused on developing the literature of searchable encryption scheme and several works published for static as well as dynamic searchable encryption, file-injection attack by Zhang et al. [1] provided a new direction of research. [1] forced the researchers to think about dynamic SE schemes to be *forward private*.

Though ORAM based DSE schemes ([13] etc.) can achieve forward privacy easily, they are impractical since communication, computation, as well as storage costs, are too high in them. The first non-ORAM based forward private DSE scheme, presented by Raphael Bost [2], was for a single keyword search only. If we consider backward privacy, there are several works like [14], [15] etc., that have this property. However, we do not consider backward private DSE schemes as there are no formal attacks against schemes that do not have the property.

Again, in most of the works, the cloud service providers are considered semi-honest i.e., honest to follow the protocol but curious about the data and queries. That is why they become vulnerable when the cloud server behaves maliciously. Being verifiable, an SE or a DSE scheme becomes protected from such cloud servers.

While discussing privately verifiability and static database, using trie-like tree data structure, verifiable SE was first introduced by Chai and Gong [16]. Each node in the tree corresponds to some keywords and stored identifiers containing it. A verifiable SE scheme for static data, based on the secure indistinguishability obfuscation, was presented by Cheng et al. [17]. The scheme supports Boolean queries with public verifiability. A no-dictionary generic verifiable SE scheme was proposed by Ogata and Kurosawa [18]. The scheme was first to allow searching any binary sting as keyword still maintaining private verifiability using Cuckoo hash table. In the case of multiple owners, Miao et al. [19] presented a verifiable SE for a single keyword search. For static data, public verifiability was achieved by Soleimanian and Khazaei [20]. Their scheme was only for single keyword searches.

Moving to complex queries, works by Miao et al. [3], Miao et al. [21], Wang et al. [4], Li et al. [5] etc. supports private verifiability, whereas work by Azraoui et al. [6], Xu et al. [22] etc. were publicly verifiable. [22] is a blockchain-based scheme that supports Boolean range queries keeping the encrypted data index and queries on a blockchain having a good amount of monetary cost for each search whereas [6] only supports conjunctive search.

The above-mentioned schemes were for static databases only. There are few works on the literature of verifiable dynamic SE schemes. For example, the schemes by Yoneyama and Kimura [23], Sardar and Ruj [24] etc. are for single keyword search and provides verifiability as well. Also, the algebraic PRF based SE scheme by Yoneyama and Kimura [23] was privately verifiable. A publicly verifiable SE scheme is recently also proposed by Miao et al. [25].

On the other hand, if we consider complex queries, for dynamic data, a dynamic fuzzy keyword search scheme was proposed by Zhu et al. [8] which is privately verifiable. Again, publicly verifiable dynamic SE scheme by Jiang et al. [9] allows the query to be Boolean where Sun et al. [7] allows only conjunctive searches.

Discussing forward privacy on dynamic database, we can see schemes being forward private and verifiable [23] but not conjunctive, verifiable, and conjunctive but not forward private [9], conjunctive as well as forward private but not verifiable [26].

In this paper, we have proposed the first forward private conjunctive DSE scheme that is verifiable too. The scheme uses a forward private single keyword DSE scheme as the base. Moreover, our scheme does not use any extra client-storage for verifiability.

3 Preliminaries

3.1 System model

In our model of conjunctive verifiable DSE, there are three entities– client, cloud and auditor. The system model is shown in Figure 1. Here, we briefly describe the system model as follows.

Client: The *client* owns the database and requires outsourcing its data. It is assumed to be a *trusted* party. Before outsourcing the data, it builds a secure search index. Then, it encrypts and sends the data together with the index. It is the user of the database as well. For every query made, it generates an encrypted query token and sends it to the cloud. Finally it receives the result from the cloud.

Cloud: The entity *cloud* is assumed to be malicious. It provides both storage and computation services. It stores the encrypted data. When a search query is given, it computes over the data and returns result to the client. It also sends proof of its correct execution to the auditor.

Auditor: The entity *auditor* is an authority that tells, verifying the proof received from the client and the cloud, whether the returned result is correct or not. Any party, including the client, can be an auditor.

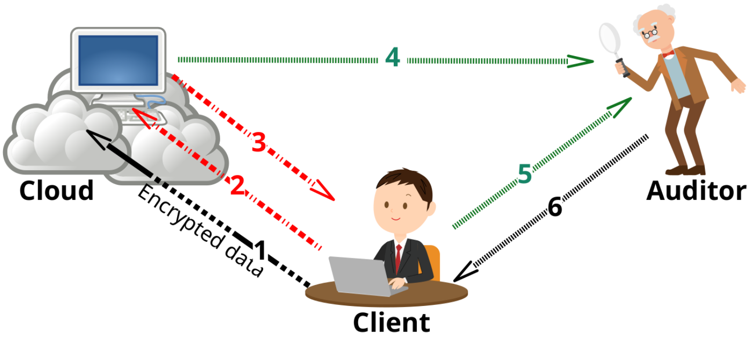


Figure 1: The system model
1. Client sends encrypted data, 2. Client sends query token to the Cloud, 3. After searching, Cloud sends Result, 4. Cloud sends proof to the Auditor, 5. Client sends proof of received result, 6. Auditor sends verification result.

3.2 Design Goals

While designing such a scheme, assuming the above system model and aiming to provide a solution toward a verifiable conjunctive search DSE scheme, with keeping it forward private, we achieve the following objectives.

Confidentiality: From the uploaded data and issued query token, the cloud should not get any information about the actual database or query. The auditor should not get the same from the received proofs.

Efficiency: We consider the client to be computationally weak, but the cloud has a large amount of storage and large computational power. Thus, in the designed scheme, computational and storage costs should be low for the client, while performing verifiability.

Scalability: It is desirable to scale the solution to support a large database.

Forward privacy: Since a DSE scheme, without forward privacy, is vulnerable to even honest-but-curious adversary, it is desirable the scheme to be forward private while achieving public verifiability for a conjunctive search result.

3.3 Cryptographic Tools

3.4 Multiset Hash

Definition 1 (Multiset Hash [27]). *Let by $M \sqsubset B$ we mean a multiset M of elements of a countable set B. Let multiset union of two multisets $M = \{m_1, m_2, \ldots, m_{|M|}\}$ and $M' = \{m'_1, m'_2, \ldots, m'_{|M'|}, \}$ be defined as $M \sqcup M' = \{m_1, m_2, \ldots, m_{|M|}, m'_1, m'_2, \ldots, m'_{|M'|}\}$.*

A triplet $(\mathsf{H}, +_{\mathsf{H}}, \equiv_{\mathsf{H}})$ of PPT algorithms is said to be a multiset hash *on B with security parameter λ when it satisfies the following properties:*

1. $\mathsf{H}(M) \in \{0,1\}^{\lambda}$, $\forall M \sqsubset B$ *(compression)*
2. $\mathsf{H}(M) \equiv_{\mathsf{H}} \mathsf{H}(M)$, $\forall M \sqsubset B$ *(comparability)*
3. $\mathsf{H}(M \sqcup M') \equiv_{\mathsf{H}} \mathsf{H}(M) +_{\mathsf{H}} \mathsf{H}(M')$, $\forall M, M' \sqsubset B$ *(incrementality)*

Clarke et al. [27] presented an incremental multiset hash function which is set-collision resistant.

3.5 Bilinear Map

Definition 2 (Bilinear Map [11]). *Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be three (multiplicative) cyclic groups of prime order p. Let $\mathbb{G}_1 =< g_1 >$ and $\mathbb{G}_2 =< g_2 >$. A map $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is said to be an* admissible non-degenerate bilinear map *if is satisfies the following properties.*

1. *$\exists$ Bilinearity i.e., $\hat{e}(u^a, v^b) = \hat{e}(u, v)^{ab}$, $\forall u \in \mathbb{G}_1$, $\forall v \in \mathbb{G}_2$ & $\forall a, b \in \mathbb{Z}_p$*
2. *$\exists$ Non-degeneracy i.e., $\hat{e}(g_1, g_2) \neq 1$, and*
3. *$\exists$ Efficiency i.e., $\hat{e}$ can be computed efficiently.*

In our case, we consider $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}$, and $\mathbb{G} =< g >$. For our scheme we require the group $\mathbb{G}$ to be a GDH group (see Definition 4). Let us consider the following bilinear map generating algorithms .

$(p, \mathbb{G}, \mathbb{G}_T, g, \hat{e}) \leftarrow \texttt{BMGen}(1^\lambda)$: It is a PPT algorithm (bilinear map generator) that takes a security parameter λ as input and outputs a uniquely random tuples $(p, \mathbb{G}, \mathbb{G}_T, g, \hat{e})$ of bilinear pairing parameters.

$(p, \mathbb{G}, \mathbb{G}_T, g, \hat{e}) \leftarrow \texttt{BMGGen}(1^\lambda)$: It is a PPT algorithm (bilinear map generator) that takes a security parameter λ as input and outputs a uniquely random tuples $(p, \mathbb{G}, \mathbb{G}_T, g, \hat{e})$ of bilinear pairing parameters where $\mathbb{G}$ *is a GDH group.*

3.6 q-Strong Diffie-Hellman Assumption

Definition 3 (q-Strong Diffie-Hellman Assumption [11]). *Let λ be a security parameter and $(p, \mathbb{G}, \mathbb{G}_T, \hat{e}, g)$ be a uniformly randomly generated tuple of bilinear pairing parameters. Given an upper bound q, an element $s \xleftarrow{\$} \mathbb{Z}_p^*$ and the set $\{g, g^{s^1}, g^{s^2}, \ldots, g^{s^q}\}$, it is said to satisfy q-*strong Diffie-Hellman *(q-SDH) assumption if, any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ can find a pair $(c, g^{\frac{1}{x+c}})$ only with negligible probability, namely*

$$Adv_{\mathcal{A}}^{q\text{-}SDH} = \Pr\left[\mathcal{A}(g, g^s, g^{s^2}, \cdots, g^{s^q}) \rightarrow (s, g^{\frac{1}{s+c}})\right] \leq neg(\lambda),$$

where $c \in \mathbb{Z}_p$.

3.7 Gap Diffie-Hellman (GDH) group

Definition 4 (GDH group [28]). *Let $\mathbb{G}$ be a multiplicative cyclic group with prime order p. For $a, b, c, \in \mathbb{Z}_p$, given $g, g^a, g^b, g^c \in \mathbb{G}$, deciding whether $c = ab$ is called* Decisional Diffie-Hellman (DDH) *problem in $\mathbb{G}$. Again, For $a, b, \in \mathbb{Z}_p$, given $g, g^a, g^b \in \mathbb{G}$, computing $g^{ab} \in \mathbb{G}$ is called* Computational Diffie-Hellman (CDH) *problem in $\mathbb{G}$. The group $\mathbb{G}$ is said to be a* Gap Diffie-Hellman (GDH) group *if, the CDH problem is hard, but the DDH problem is easy in $\mathbb{G}$.*

Definition 5 ((τ, t, ϵ)-GDH group [28]). *The group $\mathbb{G}$ is said to be (τ, t, ϵ)-GDH group if, the DDH problem on $\mathbb{G}$ can be solved in at most time τ and no algorithm which runs in time at most t can break CDH on $\mathbb{G}$ with probability $\geq \epsilon$.*

3.7.1 Signature Based on GDH Groups

Boneh et al. [28] first presented a signature scheme based on bilinear map over GDH Group. The scheme can be described as follows.

Let $\hat{e} : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ be a bilinear map where $|\mathbb{G}| = |\mathbb{G}_T| = p$, a prime and $\mathbb{G} =< g >$. A *BLS* signature scheme $\mathcal{S}$=(`KeyGen`, `Sign`, `Verify`) is given as a tuple of three algorithms as follows.

- $(sk, pk) \leftarrow \texttt{KeyGen}$: It selects $\alpha \xleftarrow{\$} \mathbb{Z}_p$. It keeps the private key $sk = \alpha$ and publishes the public key $pk = g^\alpha$.
- $\sigma \leftarrow \texttt{Sign}(sk, m)$: Given $sk = \alpha$, and some message m, it outputs the signature $\sigma = (\mathcal{H}(m))^\alpha$ where, $\mathcal{H} : \{0, 1\}^* \rightarrow \mathbb{G} \setminus \{1\}$ is a full-domain one-way hash function.
- $\{0/1\} \leftarrow \texttt{Verify}(pk, m, \sigma)$: For a message m, signature σ with public key pk, it checks whether $(g, pk, \mathcal{H}, \sigma)$ is a Diffie-Hellman tuple by verifying equality between $\hat{e}(\sigma, g)$ and $\hat{e}(\mathcal{H}(m), pk)$.

3.7.2 Dynamic universal accumulator

A dynamic universal accumulator (DUA) allows one to outsource a set of elements, with the ability to query the existence of an element in a set. It also allows the elements to be added/deleted to/from the set, together with functionality to verify the result. Let us consider a DUA proposed by Au et al. [10]. Let `AC` = (`Init`, `Gen`, `Update`, `MemWitGen`, `MemWitVer`) be such a DUA described as follows.

Initialization. $(s, tup) \leftarrow \texttt{AC.Init}(\lambda)$:
Given a security parameter λ, let us consider a uniformly generated tuple $tup = (p, \mathbb{G}, \mathbb{G}_T, g, \hat{e})$ of bilinear pairing parameters generated with `BMGGen` (see Appendix 3.5). Then $\hat{e} : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ be a bilinear pairing such that $|\mathbb{G}| = |\mathbb{G}_T| = p$ for some λ-bit prime p and $\mathbb{G} =< g >$. Let q be the maximum number of elements to be accumulated. Then a uniformly random element s from Z_p^* is selected. s is treated as secret key.

Accumulator Generation. $Acc(Y) \leftarrow \texttt{AC.Gen}(Y, s)$: Given a k-size set $Y = \{y_1, \ldots, y_k\}$, where $y_i \in Z_p^*$, let $v(s) = \prod_{y \in Y}(y + s) \mod p$ be a polynomial of degree $k \leq q$. Then the accumulator is $Acc(Y) = g^{v(s)}$, which can be computed efficiently.

Membership witness generation $wt(\bar{y}) \leftarrow \texttt{AC.MemWitGen}(PG, Y, \bar{y})$:
For a set of elements $Y = \{y_1, \ldots, y_k\} \in Z_p^*$, a membership witness $wt(\bar{y})$ for the element $\bar{y} \in Y$ is given by

$$wt(\bar{y}) = \left[g^{\prod_{i=1}^{k}(y_i+s)}\right]^{\frac{1}{\bar{y}+s}} = [Acc(Y)]^{\frac{1}{\bar{y}+s}} .$$

Membership witness verification $b_v \leftarrow \texttt{AC.MemWitVer}(Acc(Y), \bar{y},\ wt(\bar{y}))$: Membership witness is verified by checking if $\hat{e}(Acc(Y), g) = \hat{e}(wt(\bar{y}), g^{\bar{y}+s})$. Finally, a bit b_v is returned where $b_v = 1$ if equality holds and $b_v = 0$ otherwise.

Correctness: of membership verification follows from

$$\hat{e}(wt(\bar{y}), g^{\bar{y}+s}) = \hat{e}(g^{\prod_{y \in Y, y \neq \bar{y}}(y+s)}, g^{\bar{y}+s}) = \hat{e}(g^{\prod_{y \in Y}(y+s)}, g) = \hat{e}(Acc(Y), g)$$

Updating accumulator $Acc(Y') \leftarrow \texttt{AC.Update}(Acc(Y), s, \bar{y}, op)$: Let $Acc(Y)$ be the accumulator value for a set of elements $Y = \{y_1, \ldots, y_k\} \in \mathbb{G}_p$. If $\bar{y}$ is added, i.e., op = add, the new accumulator value will be $Acc(Y') = Acc(Y)^{\bar{y}+s}$, where $Y' = Y \cup \{\bar{y}\}$. Similarly, if an element $\bar{y}$ is deleted, i.e., op = delete, the accumulator changes to $Acc(Y') = Acc(Y)^{\frac{1}{\bar{y}+s}}$, where $Y' = Y \setminus \{\bar{y}\}$. For both case, the secret value s is needed to compute updated value.

3.8 Definitions and Terminologies

3.8.1 Notations

Let identifiers of the documents belong to the space of document identifiers $\mathcal{D}$. Let $\mathcal{DB} \subseteq \mathcal{D}$. Let each document contains some keywords belonging to a keyword space $\mathcal{W}$. For each keyword $w \in \mathcal{W}$, let $DB(w) = \{id_1^w, id_2^w, \ldots, id_{n_w}^w\}$ be the set of document identifiers that contains w, where $n_w = |DB(w)|$ and $id_i^w \in \mathcal{DB}$ is the ith identifier. n_w is also called the *frequency of the keyword* $w \in W$. Thus, $\bigcup_{w \in \mathcal{W}} DB(w) \subseteq \mathcal{DB}$.

Let $EDB = \{c_{id} : id \in \mathcal{D}\}$ where by c_{id} we mean the encrypted document that has id as identifier. Let us consider the existence of a one-way function which maps every document identifier to a random number. However, when we say cloud returns documents to the client, we assume the cloud performs the function on every identifier before returning them.

Let, $R : \{0,1\}^* \rightarrow \{0,1\}^*$ be a PRNG and $F : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ be a PRP. A *stateful algorithm* stores its previous states and use them to compute the current state. In Table 1, we have shown some notations used in this paper.

Table 1: Notations

Symbol	Meaning	Symbol	Meaning
S	Set of elements from $\{0,1\}^*$	DT	DIA tree
B_i	ith bucket	d	root of DT
S_i	sorted set of elements in B_i	s	secret key of client for DT
l_i	lower bound of B_i	a_i	ith accumulators
r_i	upper bound of B_i	$\mathcal{W}$	the keyword set
S_i^o	$S_i \cup \{l_i, r_i\}$	w_i	ith keyword in $\mathcal{W}$
$\mathbb{G}$	a bilinear group	x, x', x''	elements of $\mathbb{G}$
$\hat{e}$	a bilinear map	$wt(x)$	membership witness of x
e	an elements from $\{0,1\}^*$	$wtn(x)$	non-membership witness of x
g	a generator of $\mathbb{G}$	id_j^w	jth file that contains w
$\mathcal{H}$	a one-way hash $\{0,1\}^* \rightarrow \mathbb{G} \setminus \{1\}$	$\phi(v)$	membership proof of v
		$SL(v)$	set of siblings of v

3.8.2 Dynamic Searchable Encryption (DSE) scheme

A dynamic searchable encryption (DSE) scheme Σ consists of algorithms (`KeyGen`, `Build`, `SrchTkn`, `Search`, `UptdTkn`, `Update`), between a client and a server, briefly described as follows.

$K_\Sigma \leftarrow \texttt{KeyGen}(1^\lambda)$: is a PPT algorithm run by the client that takes a security parameter 1^λ and outputs the secret key K_Σ.

$(\xi, EDB) \leftarrow \texttt{Build}(\mathcal{DB}, K_\Sigma)$: is a client-side PPT algorithm that takes the dataset and the secret key as input and outputs a pair (ξ, EDB) where EDB the encrypted database, and ξ an encrypted index.

$\tau_w^\Sigma \leftarrow \texttt{SrchTkn}(w, K_\Sigma)$: is also a client-side PPT algorithm that generates an encrypted search trapdoor τ_w^Σ for a keyword w with the help of K_Σ.

$R_w \leftarrow \texttt{Search}(\xi, \tau_w^\Sigma)$: with this PPT algorithm, the server perform search over ξ for τ_w^Σ and returns the search result R_w to the client.

$\tau_u^\Sigma \leftarrow \texttt{UptdTkn}(K_\Sigma, w, id)$: Given a keyword-document pair (w, id) the client generates an token, encrypted with K_Σ, for update with the help of this PPT algorithm.

$\xi' \leftarrow \texttt{Update}(\xi, \tau_u^\Sigma, op)$ is a cloud-side algorithm that updates ξ according to the op for the update token τ_u^Σ, and keeps the updated index ξ'.

Confidentiality of a DSE

Definition 6 (CKA2-security of a DSE scheme). *[12] Let Σ =(*`KeyGen`*,* `Build`*,* `SrchTkn`*,* `Search`*,* `UptdTkn`*,* `Update`*) be a DSE scheme. Let $\mathcal{A}$ be a stateful adversary, $\mathcal{C}$ be a challenger, $\mathcal{S}$ be a stateful simulator and $\mathcal{L}_\Sigma = (\mathcal{L}_\Sigma^{bld}, \mathcal{L}_\Sigma^{srch}, \mathcal{L}_\Sigma^{updt})$ be a stateful leakage algorithm. Let us consider the following two games.*

$\boldsymbol{Real}_{\mathcal{A}}^{\Sigma}(\lambda)$*: At first $\mathcal{C}$ generates a key $K_\Sigma \leftarrow \texttt{KeyGen}(1^\lambda)$. In the same time, $\mathcal{A}$ chooses a set of documents $\mathcal{DB}$ and sends it to $\mathcal{C}$. Then, $\mathcal{C}$ computes $(\xi, EDB) \leftarrow \texttt{Build}(\mathcal{DB}, K_\Sigma)$ and sends (ξ, EDB) to $\mathcal{A}$. During search phase, $\mathcal{A}$ makes a polynomial number of adaptive queries. In each query $\mathcal{A}$ sends either a search query for a keyword w or an update query for (id, op) for a document with identifier id operation op. Depending on the query, $\mathcal{C}$ returns either the search token $t_w^\Sigma \leftarrow \texttt{SrchTkn}(w, K_\Sigma)$ or the update token $t_u^\Sigma \leftarrow \texttt{UptdTkn}(K_\Sigma, w, id)$ to $\mathcal{A}$. Finally $\mathcal{A}$ returns a bit b that is output by the experiment.*

$\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}^{\Sigma}(\lambda)$*: At first, $\mathcal{A}$ generates $\mathcal{DB}$ and gives it to $\mathcal{S}$ together with and $\mathcal{L}_\Sigma^{bld}(\mathcal{DB})$. On receiving $\mathcal{L}_\Sigma^{bld}(\mathcal{DB})$, $\mathcal{S}$ generates (ξ, EDB) and sends it to $\mathcal{A}$. $\mathcal{A}$ makes a polynomial number of adaptive queries $q \in \{w, (id, op)\}$. For each query, $\mathcal{S}$ is given either $\mathcal{L}_\Sigma^{srch}(w)$ or $\mathcal{L}_\Sigma^{updt}(id, op)$. Depending on the query q, $\mathcal{S}$ returns to $\mathcal{A}$ either search token t_w^Σ or update token t_u^Σ. Finally $\mathcal{A}$ returns a bit b that is output by the experiment.*

We say Σ is $\mathcal{L}_\Sigma$-secure against adaptive dynamic chosen-keyword attacks if for any PPT (probabilistic polynomial-time) adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that

$$|Pr[\boldsymbol{Real}_{\mathcal{A}}^{\Sigma}(\lambda) = 1] - Pr[\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}^{\Sigma}(\lambda) = 1]| \leq \mu(\lambda) \tag{1}$$

Correctness: The correctness of a DSE scheme ensures that the every search protocol must return the correct result for every query, except with negligible probability.

A DSE scheme Σ that does not leak any information about the previous search results, is said to be **forward private**.

The schemes [29], [2], etc., are good examples of a forward private DSE schemes. In our proposed scheme, we use any forward private DSE scheme Σ as a black box. We assume the black box scheme Σ is correct and $\mathcal{L}_\Sigma$-secure against adaptive dynamic chosen-keyword attacks.

3.8.3 Verifiable Dynamic Conjunctive Searchable Encryption (VDCSE)

A *dynamic conjunctive SE (DCSE)* scheme supports conjunctive keyword search in dynamic database. In the presence of a malicious adversary, a *verifiable dynamic conjunctive SE* scheme provides the ability to verify whether the returned result is consistent with the updated database. We define a VDCSE scheme formally as follows.

Definition 7 (Verifiable Dynamic Conjunctive Searchable Encryption). *A verifiable dynamic conjunctive searchable encryption (VDCSE) scheme Ψ is a tuple* (`VCKeyGen`*,* `VCBuild`*,* `VCSrchTkn`*,* `VCSearchCD`*,*`VCSearchCT`*,* `VCUpdtTkn`*,* `VCUpdate`*) of algorithms defined as follows.*

- $K_\Psi \leftarrow$ $\texttt{VCKeyGen}(\lambda)$*: Given a security parameter λ, this PPT algorithm, run by the client, outputs a key K_Ψ.*
- $(\texttt{st}, EDB, \gamma, \mathcal{I}\ ) \leftarrow \texttt{VCBuild}(\mathcal{DB}, K_\Psi)$*: This is PPT algorithm run by the client. Given K_Ψ and a set of documents $\mathcal{DB}$, it outputs the encrypted set of documents EDB together with an encrypted search index γ and an auxiliary data structure $\mathcal{I}$ for verifiability. It also outputs state* $\texttt{st}$ *of the database.*
- $\tau_s^\Psi \leftarrow \texttt{VCSrchTkn}(K_\Psi, \hat{w}, \texttt{st})$*: Given K_Ψ,* $\texttt{st}$ *and a set of keywords $\hat{w}$, the client runs this PPT algorithm and outputs a search token τ_s^Ψ.*
- $(pf_c, \hat{R}_{\hat{w}}, X_{\hat{w}}) \leftarrow \texttt{VCSearchCD}(\gamma, \mathcal{I}, t_s^\Psi)$*: Given γ, $\mathcal{I}$ and τ_s^Ψ, in this cloud-run PPT algorithm, the cloud returns result set $\hat{R}_{\hat{w}}$ of document ids, a proof pf_c.*
- $(pf_u) \leftarrow \texttt{VCSearchCT}(\hat{R}_{\hat{w}}, t_s^\Psi)$*: Given $\hat{R}_{\hat{w}}$, in this client-run PPT algorithm, the client returns a proof pf_u.*
- $\nu_{\hat{w}} \leftarrow \texttt{VCVerify}(d, pf_u, pf_c, \hat{R}_{\hat{w}})$ *This is a PPT algorithm that takes the proofs pf_u, pf_c together with the result $\hat{R}_{\hat{w}}$ and outputs the verification bit $\nu_{\hat{w}}$*
- $(\tau_u^\Psi, \texttt{st}') \leftarrow \texttt{VCUpdtTkn}(K, id, \texttt{st})$*: Taking a key K_Ψ, a document identifier id and the present state* $\texttt{st}$*, the cloud runs this PPT algorithm and outputs an update token τ_u^Ψ and a new state* $\texttt{st}'$*.*
- $(EDB', \gamma', \mathcal{I}') \leftarrow \texttt{VCUpdate}(\gamma, EDB, \mathcal{I}, \tau_u^\Psi, op)$*: It is a cloud-run PPT algorithm which takes an update token τ_u^Ψ, operation bit op, EDB, the index γ and the auxiliary information $\mathcal{I}$ and outputs updated $(EDB', \gamma', \mathcal{I}')$.*

Correctness A VDCSE scheme Ψ is said to be *correct* if $\forall \lambda \in \mathbb{N}$, $\forall K_\Psi$ generated using $\texttt{KeyGen}(\lambda)$ and all sequences of search and update operations, every search outputs the correct set of identifiers, except with a probability $neg(\lambda)$.

Verifiability By verification, we mean verification of a search result. We verify whether the search is performed on the current state of the database. We do not include verification of updates on the cloud-side. If the cloud cheats, and updates incorrectly, it will fail verification test when a search result includes such updated information.

3.8.4 Security Definitions

We follow security definition of [20]. Security of a VDCSE scheme is divided into two parts– confidentiality and soundness, described as follows.

Confidentiality: This property protects the client to leak only allowed amount of information, not more than that. We define the confidentiality of a VDCSE as follow.

Definition 8 (CKA2-Confidentiality)**.** *Let Ψ = (*$\texttt{VCKeyGen}$*,* $\texttt{VCBuild}$*,* $\texttt{VCSrchTkn}$*,* $\texttt{VCSearchCD}$*,* $\texttt{VCSearchCT}$*,* $\texttt{VCUpdtTkn}$*,* $\texttt{VCUpdate}$*) be a verifiable dynamic conjunctive searchable Encryption scheme. Let $\mathcal{A}$, $\mathcal{C}$ and $\mathcal{S}$ be a stateful adversary, a challenger and a stateful simulator respectively. Let $\mathcal{L}_\Psi = (\mathcal{L}_\Psi^{bld}, \mathcal{L}_\Psi^{srch}, \mathcal{L}_\Psi^{updt})$ be a stateful leakage algorithm. Let us consider the following two games.*

$\boldsymbol{Real}_{\mathcal{A}}^{\Psi}(\lambda)$*: At first, a key $K_\Psi \leftarrow$* $\texttt{VCKeyGen}(\lambda)$ *is generated by the challenger $\mathcal{C}$. Then a database $\mathcal{DB}$ is chosen by the adversary $\mathcal{A}$ and sent to $\mathcal{C}$. The encrypted database EDB is built and an encrypted search index is generated by $\mathcal{C}$ as $(\texttt{st}, EDB, \gamma, \mathcal{I}) \leftarrow \texttt{VCBuild}(\mathcal{DB}, K_\Psi)$ and then $(EDB, \gamma, \mathcal{I})$ is sent to $\mathcal{A}$. In the next phase a polynomial number of adaptive queries are made by $\mathcal{A}$. In each of them, either a search query for a keyword set $\hat{w}$ or an update query for a keyword-document pair (w, id) and operation bit op is sent to $\mathcal{C}$ by it. In sequence, $\mathcal{C}$ returns either a search token $\tau_s^\Psi \leftarrow \texttt{VCSrchTkn}(K_\Psi, \hat{w}, \texttt{st})$ or an update token $\tau_u^\Psi \leftarrow \texttt{VCUpdtTkn}(K, id, \texttt{st})$ to $\mathcal{A}$. Finally, a bit b, that is output of the experiment, is returned by $\mathcal{A}$.*

$\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}^{\Psi}(\lambda)$*: At first, a database $\mathcal{DB}$ is chosen by $\mathcal{A}$ and is given to $\mathcal{S}$ together with $\mathcal{L}_\Psi^{bld}(\mathcal{DB})$. Then, a simulated database and index (EDB, γ) is generated by $\mathcal{S}$ and sent to $\mathcal{A}$. Then a polynomial number of adaptive queries are made by $\mathcal{A}$. For each query, either $\mathcal{L}_\Psi^{srch}(\hat{w}, \mathcal{DB})$ or $\mathcal{L}_\Psi^{updt}(op, w, id)$, depending on the query, is given to $\mathcal{S}$. Accordingly, $\mathcal{S}$ returns either search token τ_s^Ψ or update token τ_u^Ψ to $\mathcal{A}$. Finally, a bit b', that is output of the experiment, is returned by $\mathcal{A}$.*

We say Ψ is $\mathcal{L}_\Psi$-secure against adaptive dynamic chosen-query attacks if $\forall$ PPT adversary $\mathcal{A}$, $\exists$ a simulator $\mathcal{S}$ such that

$$|Pr[\boldsymbol{Real}_{\mathcal{A}}^{\Psi}(\lambda) = 1] - Pr[\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}^{\Psi}(\lambda) = 1]| \leq \mu(\lambda) \tag{2}$$

where $\mu(\lambda)$ is negligible in λ.

Soundness The soundness property ensures the client gets complete result with respect to the present state of database. The game-based definition of soundness property of a VDCSE scheme is given as follow.

Definition 9 (Soundness). *Let Ψ be a VDCSE scheme with $\Psi =$ (`VCKeyGen`, `VCBuild`, `VCSrchTkn`, `VCSearchCD`, `VCSearchCT`, `VCUpdtTkn`, `VCUpdate`). Let us consider the following game.*

sound$_{\mathcal{A}}^{\Psi}(\lambda)$*: At first, a key $K_\Psi \leftarrow$ `VCKeyGen`(λ) is generated by the challenger $\mathcal{C}$. Then, a database $\mathcal{DB}$ is chosen by the adversary and sent to $\mathcal{C}$. The encrypted database is computed as $(\mathtt{st}, EDB, \gamma, \mathcal{I}) \leftarrow$ `VCBuild`$(\mathcal{DB}, K_\Psi)$ by $\mathcal{C}$ and then $(EDB, \gamma, \mathcal{I})$ is sent to $\mathcal{A}$. A polynomial number of adaptive queries are made by $\mathcal{A}$. In each of them, either a search query, for a keyword set $\hat{w}$, or an update query, for a keyword-document pair (w, id) and operation bit op, is sent to $\mathcal{C}$. In response, depending on the query, either a search token $\tau_s^\Psi \leftarrow$ `VCSrchTkn`$(K_\Psi, \hat{w}, \mathtt{st})$ or an update token $\tau_u^\Psi \leftarrow$ `VCUpdtTkn`$(K, id, \mathtt{st})$ is returned to $\mathcal{A}$.*

In the challenge phase, a target keyword set $\hat{w}$ is chosen by $\mathcal{A}$ and a search query for $\hat{w}$ is sent to $\mathcal{C}$. In response, a search token τ_s^Ψ is returned from which $(R_{\hat{w}}, \nu_{\hat{w}})$ is searched by $\mathcal{A}$, where $\nu_{\hat{w}} =$ accept is verification bit from $\mathcal{C}$. Finally, a pair $(R^*_{\hat{w}}, \nu^*_w)$ for a keyword set $\hat{w}$ is generated by $\mathcal{A}$. If $\nu^*_{\hat{w}} =$ accept even when $R^*_{\hat{w}} \neq \bigcap_{w \in \hat{w}} DB(w)$, $\mathcal{A}$ returns 1 as output of the game, otherwise returns 0.

We say that Ψ is sound *if $\forall$ PPT adversaries $\mathcal{A}$, $Pr[$**sound**$_{\mathcal{A}}^{\Psi}(\lambda) = 1] \leq \mu(\lambda)$.*

4 Dynamic Interval Accumulation tree (DIA tree)

If we consider membership witnesses, they can not be updated without the secret key. However, the client computes them initially by itself and stores them in the cloud. It fetches and updates them each time a new element is added or deleted. For a given set of elements, if only one accumulator is generated for the set, then the generating membership witness for a new element x becomes inefficient. This is because the membership witness generation considers all elements belong to the set in the computation i.e., computational complexity grows with the size of the set. If the set is too large, the computation of the witness becomes impractical.

This is why, instead of generating a single accumulator, the set is divided into buckets and a separate accumulator is generated for every bucket. In the next level, this set of accumulators becomes input set and another set of accumulators is generated for them. The process continues until only one element, i.e., the root is left. The generated tree is an accumulation tree.

Papamanthou et al. [11] studied the above approach previously in their work of authenticated hash table. In spite of the work is good for membership-proof and supports updates, it has a serious issue. For non-membership poof, they take an additional accumulation tree which is based on intervals. This tree does not support deletion. This makes the tree an append-only tree.

We take a different interval approach and construct DIA tree. The tree gives the ability to perform both membership and non-membership in a single tree, even in case the set is dynamic and large.

4.1 DIA Tree construction

For a given set S, in our proposed DIA tree construction, the set is stored separately. A tree is constructed to give proof of whether an element exists in the given set S. We describe a DIA tree scheme `DIAT` as a tuple (`Init`, `BuildTree`, `Search`, `Update`) of algorithms as follows.

Initialization $[(s, tup) \leftarrow \mathtt{DIAT.Init}(1^\lambda)]$: Given a security parameter λ, a tuple $(p, \mathbb{G}, \mathbb{G}, \hat{e}, g) \leftarrow \mathtt{BMGen}(1^\lambda)$ is generated. Let $tup = (p, \mathbb{G}, \mathbb{G}, \hat{e}, g)$. An element s is chosen randomly from $\mathbb{Z}_p^*$ and finally (tup, s) is returned.

Building the Tree $[(DT, d) \leftarrow \mathtt{DIAT.BuildTree}(tup, s, S)]$: We consider in the given set $S = \{e_1, e_2, \ldots, e_n\}$, each e_i is of fixed length and sorted. Let the complete range of elements be $[0, 2^\lambda)$. We divide the range into b half-open intervals, each of size 2^η. Then the number of intervals will be $b = 2^{\lambda - \eta}$. Let the ith intervals be $[2^{i-1}, 2^i)$ and it corresponds to the ith bucket B_i. Finally, we take range of bucket B_i as closed intervals $[2^{i-1} - 1, 2^i]$. Let $S_i = \{e_{i,1}, e_{i,2}, \ldots, e_{i,n_i}\}$ be the sorted set of elements from S, falls into the bucket B_i. We consider $S_i^o = S_i \cup \{l_i, r_i\}, i = 1, \ldots, b$, where $l_i = 2^{i-1} - 1$, $r_i = 2^i$, $\forall i = 1, \ldots, b$, and $l_0 = -\infty$ and $r_{n_i} = +\infty$. Now, we treat each S_i^o separately as follow.

Let $I_{i,j} = (e_{i,j}, e_{i,j+1}), \forall j = 0, \ldots, n_i$, where $e_{i,0} = l_i$ and $e_{i,n_i} = r_i$. Then, we map each of the intervals in $\mathbb{G}$ as $x_{i,j} = \mathcal{H}(I_{i,j})$, where $\mathcal{H} : \{0,1\}^* \rightarrow \mathbb{G}$ that brings each interval to an element in $\mathbb{G}$. Let $\bar{S}_i = \{(x_{i,j}) : j = 0, \ldots, n_i\}$. Thus, the set of elements belongs to S_i transfers to the set of $\bar{S}_i$.

Finally, we get the sets $\bar{S}_1, \bar{S}_2, \ldots, \bar{S}_b$ and make accumulators for each of them. The elements of the set $\bar{S}_i$ are kept in the leaves, say at Level-h, where h is the height of the tree. For each set $\bar{S}_i$, the accumulator is generated as $a_i \leftarrow \texttt{AC.Gen}(\bar{S}_i, s)$ where $a_i = g^{\prod_{x \in \bar{S}_i} (\mathcal{H}'(x)+s)} \in \mathbb{G}$ and $\mathcal{H}' : \mathbb{G} \to \mathbb{Z}_p^*$.

Next, we start from the set $\{a_1, a_2, \ldots, a_b\}$, recursively make an m-ary tree, above the leaves, until we reach only one accumulator, say d, the digest of the tree. If h is the height of the tree then $m^{h-1} = b$. Thus, every internal node of the tree DT stores the accumulator corresponding to its set of children. Moreover, every node, including leaf nodes and excluding the root, contains membership proof with respect to its parent. Thus if v is a node, it keeps membership proof as $\phi(v) = g^{\prod_{x \in SL(v)} (\mathcal{H}'(x)+s)} \in \mathbb{G}$ where $SL(v)$ is the set of siblings of v. We can see that the each internal node has same number of children, whereas Level-$(h-1)$ nodes stores random number of children. Finally, the accumulation tree DT is returned to the cloud, and the client stores d.

Search $[(b_x, w_e) \leftarrow \texttt{DIAT.Search}(DT, e)]$: Given an element e, this algorithm tells whether it exists together with a witness. This is done as follows.

Let B_k be the bucket for e. If $e \notin S_k$, it finds other two elements $e', e'' \in S_k^o$ such that $e' < e < e''$ in the bucket corresponding to e. Then it computes $x \leftarrow \mathcal{H}(I)$ where $I = (e', e'')$. Then it gives membership witness $wt(x) = \pi_I$ for x in DT. Let $v_0, v_1, \ldots, v_h$ nodes in the path corresponding to the node x where v_h is the root of the tree. Then, $wt(x)$ is of the form (e', e'', π_I) where $\pi_I = (\pi_1, \pi_2, \ldots, \pi_h)$ and each π_i is a pair (α_i, β_i) defined as

$$\alpha_i = \Phi(v_{i-1}) \text{ and } \beta_i = g^{\prod_{u \in SL(v_{i-1})} (\mathcal{H}(\Phi(u))+s)}, i = 1, 2, \ldots, h \tag{3}$$

Note that, in our proposed scheme, we pre-compute both α_i and β_i.

Now, if $e \in S_k$, it finds another two elements $e', e'' \in S_k^o$ such that $e' < e < e''$. Then the witness of non-existence of x is given by $wtn(x) = (e', e'', \pi_{I'}, \pi_{I''})$ where $I' = (e', e)$ and $I'' = (e, e'')$.

$b_e = 1$ indicates existence and $w_e = wt(x)$ is set whereas $b_x = 0$ indicates the opposite and $w_e = wtn(x)$ is set.

Verify Search $[b \leftarrow \texttt{DIAT.VerifySearch}(d, b_e, w_e, e)]$: If $b_e = 1$, verifier verifies whether $\beta_i^{\mathcal{H}(\alpha_{i-1})+s} = \alpha_i$, $\forall i = 1, 2, \ldots, h$. It recomputes the element $x = \mathcal{H}'(e', e'')$ and computes $\alpha_i = x$. Then, it verifies

$$\hat{e}(\alpha_i, g) = \hat{e}(\beta_{i-1}, g^{(\mathcal{H}(\alpha_{i-1})+s)}), i = 1, 2, \ldots, h,$$

Additionally, it is checked if $\hat{e}(d, g) = \hat{e}(\beta_l, g^{(\mathcal{H}(\alpha_h)+s)})$ where d is the root digest. The result is accepted if all are verified correctly.

If $b_e = 0$, it recomputes the element $x_1 = \mathcal{H}'(e', e)$ and $x_2 = \mathcal{H}'(e, e'')$. verifies the same for both intervals. It returns accept if, witnesses are verified for both intervals. verification is done similarly as above.

Update $[d' \leftarrow \texttt{DIAT.Update}(DT, T, s, d, e, op)]$: Given an element e let B_k be its bucket. Then it finds two elements $e', e'' \in S_k^o$, such that $e' < e < e''$. Then $x' \leftarrow \mathcal{H}'(e', e)$, $x'' \leftarrow \mathcal{H}'(e, e'')$ and $x \leftarrow \mathcal{H}'(e', e'')$ are computed. Let v_1, v_2, v_h be the path above them. Let $\phi(v), wt(v)$ denotes accumulator and witness stored in v resp. Now, for $op =$ add, we do the following.

1. At level h, x', x'' are inserted and x is removed. The client can calculate and upload their witnesses as $wt(x') \leftarrow \{\phi(v_1)\}^{\frac{(\mathcal{H}(x')+s)}{(\mathcal{H}(x)+s)}}$ and $wt(x'') \leftarrow \{\phi(v_1)\}^{\frac{(\mathcal{H}(x'')+s)}{(\mathcal{H}(x)+s)}}$

2. For v_1, client computes $\phi_1(v_1) \leftarrow \{\phi(v_1)\}^{\mathcal{H}(x')+s}$, $\phi_2(v_1) \leftarrow \{\phi_1(v_1)\}^{\mathcal{H}(x'')+s}$ and $\phi_0(v_1) \leftarrow \{\phi_2(v_1)\}^{\frac{1}{\mathcal{H}(x)+s}}$. It computes
$\phi_1(v_i) \leftarrow (\phi(v_i))^{(\mathcal{H}(\phi_0(v_{i-1}))+s)}$ and $\phi_0(v_i) \leftarrow (\phi_1(v_i))^{\frac{1}{(\mathcal{H}(\phi(v_{i-1}))+s)}}$, for other v_is. Thus, the new accumulator values along the path are $\phi_0(v_1)$, $\phi_0(v_2)$, …, $\phi_0(v_h)$.

3. For each child u of v_1, server computes updated witness $wt_0(u)$ without using s directly as follow.

   (a) Compute $wt_1(u) \leftarrow \phi(v_1).(wt(u))^{(\mathcal{H}(x')-\mathcal{H}(u))}$

   (b) Compute $wt_2(u) \leftarrow \phi_1(v_1).(wt_1(u))^{(\mathcal{H}(x'')-\mathcal{H}(u))}$

   (c) compute $wt_0(u) \leftarrow \left(\dfrac{wt_2(u)}{\phi_0(v_1)}\right)^{\frac{1}{\mathcal{H}(x)-\mathcal{H}(u)}}$

4. Finally for any other child u of v_i new witnesses computed by the server are

$$wt_1(u) \leftarrow \phi(v_1).(wt(u))^{\mathcal{H}(\phi_0(v_i))-\mathcal{H}(u)} \text{ and}$$

$$wt_0(u) \leftarrow \left(\frac{wt_1(u)}{\phi_0(v_1)}\right)^{\frac{1}{\mathcal{H}(\phi(v_i))-\mathcal{H}(u)}}$$

5. The client keeps $d' = \phi(v_h)$ as the new digest of the root

The client needs to keep the new digest only. Verification of the update is not required. If the server changes something, no search result will be verified correctly.

If $op =$ delete, the tree can be updated in a similar way. The only changes in the algorithm are membership witnesses update of the leaf nodes (of the same bucket) and updating $\phi(v_1)$. During deletion, at Level-h, x is inserted and x', x'' are removed and the tree is updated accordingly as follows.

1. For v_1, client computes updated witness $\phi_0(v)$ of $\phi(v)$ as

$$\begin{aligned} \phi_1(v_1) &\leftarrow \{\phi(v_1)\}^{(\mathcal{H}(x)+s)}, \\ \phi_2(v_1) &\leftarrow \{\phi_1(v_1)\}^{\frac{1}{\mathcal{H}(x)+s}}, \text{ and} \\ \phi_0(v_1) &\leftarrow \{\phi_2(v_1)\}^{\frac{1}{\mathcal{H}(x'')+s}}. \end{aligned}$$

   For $1 < i <= h$, similarly as in delete, it computes the new accumulator values $\phi_0(v_1), \phi_0(v_2), \ldots, \phi_0(v_h)$.

2. For each child u of v_1, server computes updated witness $wt_0(u)$ without using s directly as follow.

   (a) Compute $wt_1(u) \leftarrow \phi(v_1).(wt(u))^{(\mathcal{H}(x)-\mathcal{H}(u))}$

   (b) Compute $wt_2(u) \leftarrow \left(\frac{wt_1(u)}{\phi_2(v_1)}\right)^{\frac{1}{\mathcal{H}(x')-\mathcal{H}(u)}}$

   (c) compute $wt_0(u) \leftarrow \left(\frac{wt_2(u)}{\phi_0(v_1)}\right)^{\frac{1}{\mathcal{H}(x'')-\mathcal{H}(u)}}$

3. Additionally, at Level-h, x is inserted and x', x'' are removed. Client computes the witnesses

$$wt(x) \leftarrow \{\phi(v_1)\}^{\frac{1`}{(\mathcal{H}(x')+s)(\mathcal{H}(x'')+s)}}.$$

4. Finally, for any other child u of v_i $(i > 1)$ new witness $wt_0(u)$ computed similarly as

$$wt_0(u) \leftarrow \left(\frac{wt_1(u)}{\phi_0(v_1)}\right)^{\frac{1}{\mathcal{H}(\phi(v_i))-\mathcal{H}(u)}} \text{ where, } wt_1(u) = \phi(v_1).(wt(u))^{(\mathcal{H}(\phi_0(v_i))-\mathcal{H}(u))}$$

5. The client keeps $d_0 = \phi(v_h)$ as the new digest of the root

4.2 Example of a DIA tree

Let us consider a 3-ary tree with height $h = 3$. Then there are $h + 1$ levels where Level-0 is the root. Then Level-2 has 9 elements. Each node at Level-2 can hold at most 5 elements. Then, all possible elements can be mapped in [0,44]. The ith element at Level-2 corresponds to the bucket $[5i, 5(i + 1) - 1]$. However, for construction, we want them to be in some open interval which allows any operation to effect one bucket only. So, we take ith interval as $I_i = (l_i, r_i) = (5i - 1, 5(i + 1))$ (see Figure 2).

Now, given a set $S = \{6, 7, 9, 13, 21, 24\}$, we consider the following 15 (open) intervals, $(-1, 5)$, $(4, 6)$, $(6, 7)$, $(7, 9)$, $(9, 10)$, $(9, 13)$, $(13, 15)$, $(14, 20)$, $(19, 21)$, $(21, 24)$, $(24, 25)$, $(24, 30)$, $(29, 35)$, $(34, 40)$, $(39, 45)$. Let I_i be the ith interval. Then we take hash $x_i = \mathcal{H}(I_i)$, for each i, to map them as an element of G. Then the 2nd Level-2 node stores $\{x_2, x_3, x_4, x_5\}$, 5th Level-2 node stores $\{x_9, x_{10}, x_{11}\}$, 6th stores x_{12} etc.

In addition, any node, except leaves, stores accumulator for its set of children. Moreover, any node, except the root, stores witness of membership in the parent node.

Search: To search an element $e = 21$, at first, its corresponding interval is searched. Let its boundaries be $l = 19$ and $r = 25$. And then it searches two elements $e' = 13$ and $e'' = 24$ in S such that $e' < e < e''$. Finally, it sets $e' = max\{e', l\}$ and $e'' = min\{e'', r\}$. This is equivalent to say that we are choosing two elements e' and e'' in the left and the right of e respectively from the bucket corresponding to e (see Figure 3).

Since, 21 is in S, it considers the intervals as $I' = (e', e) = (19, 21)$ and $I'' = (e, e'') = (21, 24)$. Then calculate the hashes $x' = \mathcal{H}(I')$ and $x'' = \mathcal{H}(I'')$. We see that both x' and x'' are in the tree. So, for each of

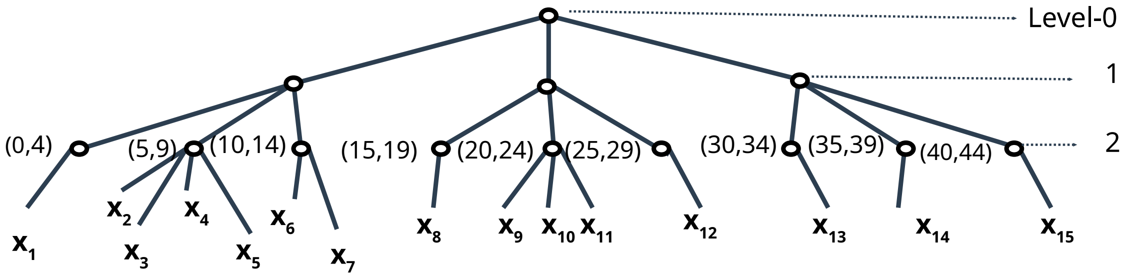


Figure 2: DIA tree for S

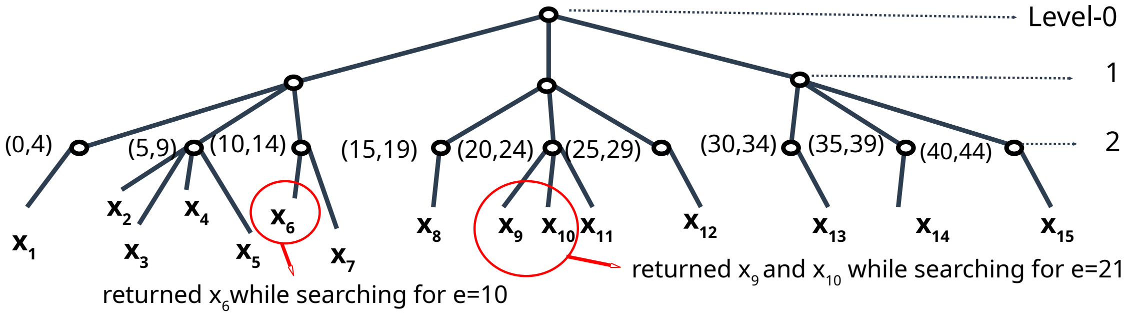


Figure 3: search for $e = 21$ and $e = 10$

them, with the result, cloud returns proof of their existence. The proof contains accumulators and witnesses stored in every node from leaf to root in the path of the bucket.

Similarly, if 10 is searched, the proof for the interval $(9, 13)$ is returned. This is because if some element belongs to S, it appears in two intervals– in one as the right boundary and in another as the left boundary. When it is not in S, there exists an interval that contains the searched element.

Update: When we want to add $e = 11$, we find similarly $e' = 9$ and $e'' = 13$ such that $e' < e < e''$, in the bucket corresponding to 3 (see Fig. 4a). Then we just remove x_6 corresponding to the interval $I = (9, 13)$ and then add two intervals $I' = (9, 11)$ and $I'' = (11, 13)$. For that we remove $x_6 = \mathcal{H}(I)$ and add both $x' = \mathcal{H}(I')$ and $x'' = \mathcal{H}(I'')$. After doing the same, accumulators in the path of the bucket from leaf to root and and witnesses for each of their children are updated.

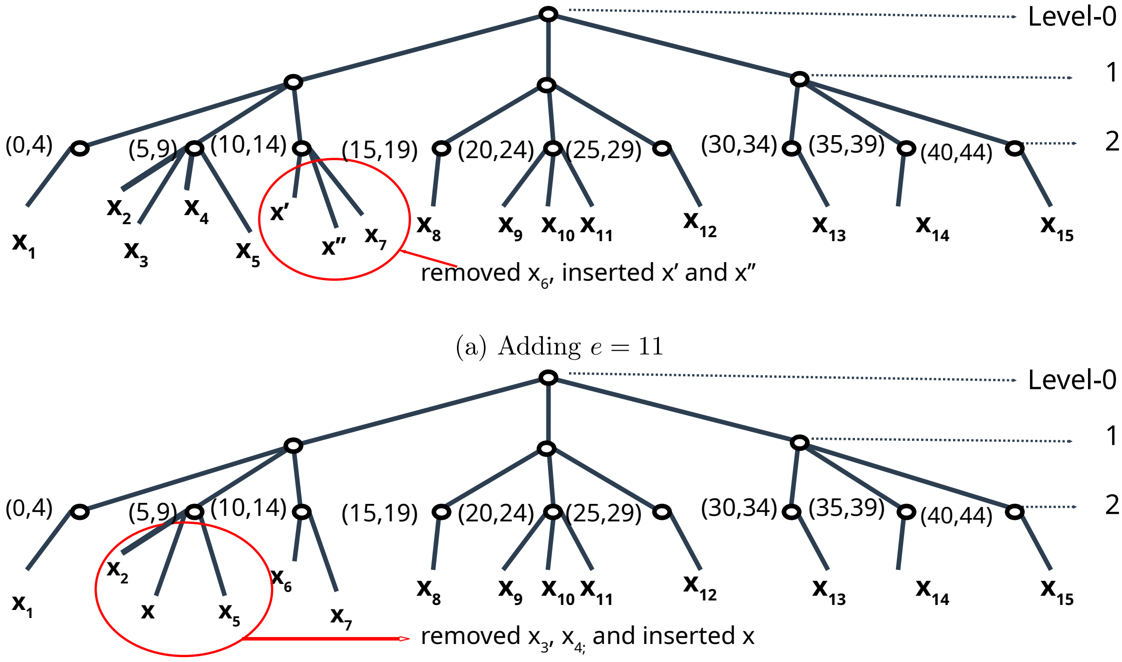
Again, we delete an element only if it exists in S. So, in that case, given an element $e = 7$, we can find two intervals I' and I'' where e is left and right bound i.e., $I' = (6, 7)$ $I'' = (7, 9)$. So, $e' = 6$ and $e'' = 9$ (see Fig. 4b). Let $I = (e', e'') = (6, 9)$. To delete the element $e = 7$, we remove both $x' = \mathcal{H}(I')$ and $x'' = \mathcal{H}(I'')$, and then add $x = \mathcal{H}(I)$ and update the tree accordingly.

4.3 Complexity of DIA tree

Let h be the height of the tree. Since, the leaves store elements in G corresponding to the intervals, parents of the leaves store different numbers of elements. However, the tree is a complete m-ary tree from Level-0 to Level-$(h-1)$, i.e., without leaves.

Storage Cost: Since, the tree is an m-ary tree without the leaves, it can hold upto $b = m^{(h-1)}$ elements in Level-$(h-1)$ and each node at Level-$(h-1)$ can hold at most $\frac{2^\lambda}{m^{(h-1)}}$ elements. However, there may be some nodes at Level-$(h-1)$ that may not contain only one element. If the size of the set is n, then the number of leaves is $n+1+m^{(h-1)}$. Now, each node stores an accumulator of its children and a witness of its parent. The root only keeps an accumulator and every leaf keeps an element in $\mathbb{G}$ corresponding to its interval. Number of internal nodes, from root to Level-$(h-1)$ is $m^{(h-1)} + m^{(h-2)} + \ldots + 1 \ = (m^{(h)} - 1)/(m-1)$. Thus, the number of elements the DIA tree store is $2(\frac{m^h-1}{m-1} + m^{(h-1)} + 1 + n) - 1$. This is stored at the cloud-side. The client keeps only the root and the secret key.

Building Cost: The numbers of accumulators at internal nodes is $\frac{m^h-1}{m-1}$. Among them, $\frac{m^{(h-1)}-1}{m-1}$ accumulators, in the Level above $h-1$, are for set of size m each. The accumulators at Level-$(h-1)$ are for the set



(a) Adding $e = 11$

(b) deleting $e = 7$

Figure 4: Updating the tree

of average size $\frac{n-1}{m^h}$. Besides, the number of witnesses to be generated is $(\frac{m^h-1}{m-1} + m^{(h-1)} + 1) + (n-1) = \frac{m^h-1}{m-1} + m^{(h-1)} + n$ The above cost is a one time client-side cost.

Search Cost: During a search, the cloud has to return the accumulators and witnesses in the paths corresponding to the given intervals. The cloud can retrieve them $O(2(h+1))$ or $O(h+1)$ time depending on whether the search element exists or not. Thereafter, the cloud returns $4(h+1)$ group elements if the searched element exists, else returns $2(h+1)$ group elements.

Verification Cost: To verify the result, the verifier needs to compute $4(h+1)$ and $2h$ powers in $\mathbb{G}$. The cost is half if the searched element does not exist.

Update Cost: During an update, an interval, the client retrieves all nodes in the path corresponding to the interval and all witnesses that are affected due to this change. The client only retrieves and updates $2(m(h-1)+1)$ accumulators and sends them back to the cloud. The cloud stores them and updates the witnesses to their children. The number of such witnesses is $m(h-1)$ for the nodes above Level-h. Other than that, there can have $2^\lambda/b$ witnesses at the bottom at most whereas the number is $|S|/b$ on average . The cost is double when the searched interval exists in the database.

4.4 Advantages of DIA tree

We have seen how the system works in a DIA tree, using on interval-approach. For a given set S, in both of [11] and our proposed DIA tree construction, the set is stored separately. In [11], leaf nodes stores $\mathcal{H}(x), \forall x \in S$ where $\mathcal{H}$ maps each element in a bilinear group $\mathbb{G}$. However, in our case, we store maps of the open intervals as $\mathcal{H}((x,y)); x, y \in S$.

Papamanthou et al. [11] used an interval-based approach for non-membership proof only. They store the given set S, and an accumulation tree corresponding to the open intervals. However, *there is no formal description of how it works.* Secondly, they have to maintain two trees, one for membership proof and another for non-membership proof where the second one works only when the given set S is static.

Our proposed accumulation tree, DIA tree, gives proof of membership as well as non-membership even the set S is dynamic. Moreover, it uses a single tree, resulting reduction of cloud storage. We achieve those at the cost of computation. In DIA tree, the update time is thrice, and the search time is at most twice than that in [11]. However, the required time is asymptotically the same for both. We give computational complexity of DIA tree in Appendix 4.3.

There is another basic difference between the two constructions. In [11], the computation of witnesses is done by the server when verification is required. This is useful when the frequency of the search is very low. However, if a frequent search is there, we have to return only proofs fast. So, in the DIA tree we pre-compute all witnesses which enable the frequent search. For the same, the client has to update the $O(mh)$ witnesses during each interval update.

Moreover, we can see that the sorted Merkle tree can solve the problem of both membership and non-membership proof with very efficiently. However, one downside of sorted Merkle hash trees is that even if a single element in the data set S is changed, that element may need to move to a different leaf, and the entire hash tree will need to be recomputed from scratch. This can take $O(|S|)$ hash computations. DIA tree provide the same functionality as Merkle trees, but also support an efficient update, requiring at most $O(log|S|)$ calculations to update an element. This is the advantage of any accumulation tree over Merkle tree including our proposed one.

4.5 Security of a DIA tree

We see that $\Phi(v)$ gives an accumulator corresponding to the subset of the set S rooted at v. Thus, $\Phi(v)$ is also called the *bilinear digest* of the tree rooted at v.

Theorem 1 (Acc Tree Security)**.** *Given a security parameter λ and a set $U = \{x_1, x_2, \ldots, x_n\}$, $x_i \in \mathbb{G}$, let DT be the accumulation tree constructed with* `AC.Gen()` *as above. Under the q-strong Diffie-Hellman assumption, the probability that a PPT adversary $\mathcal{A}$, knowing only the bilinear pairings parameters $(p, \mathbb{G}, \mathbb{G}, \hat{e}, g)$ and the elements $\{g, g^{s^1}, g^{s^2}, \ldots, g^{s^q}\}$, of $\mathbb{G}$, for some randomly chosen s from $\mathbb{Z}_p^*$ and $n \leq q$, can find another set V, with elements from $\mathbb{G}$, such that $V \neq U$ and $\Phi(V) = \Phi(U)$ is neg(k).*

Proof. Follows from the proof of Papamanthou et al. [11]. □

Theorem 2 (Security of our construction)**.** *Given a security parameter λ and a set $S = \{e_1, e_2, \ldots, e_n\}$, where $e_i \in \{0,1\}^*$, let* `DIAT` *be the accumulation tree constructed as above. Under the q-strong Diffie-Hellman assumption, the probability that a PPT adversary $\mathcal{A}$, knowing only the bilinear pairings parameters $(p, \mathbb{G}, \mathbb{G}, \hat{e}, g)$ and the elements $\{g, g^{s^1}, g^{s^2}, \ldots, g^{s^q}\}$ of $\mathbb{G}$, for some randomly chosen s from $\{0,1\}^*$ and $n \leq q$, can find another set S', with elements from $\mathbb{G}$, such that $S' \neq S$ and $\Phi(S') = \Phi(S)$ is $neg(\lambda)$.*

Proof. Here, we use Theorem 1 with reduction method. We show that if Theorem 2 is false, then so is Theorem 1. But, since Theorem 1 is true, it implies Theorem 2 is true.

The main difference between our `DIAT` and Papamanthou et al. [11] is that our scheme supports efficient updates.

Since $\mathcal{H}$ is public, if $S = \{e_1, e_2, \ldots, e_n\}$ is given then so is $\bar{S} = \{x_1, x_2, \ldots, x_n\}$. Let us consider Theorem 2 does not hold, then there exists a PPT algorithm $\mathcal{A}$, which finds another set $S' = \{e'_1, e'_2, \ldots, e'_{n'}\}$ such that $\Phi(S') = \Phi(S)$ with probability $\geq neg(\lambda)$.

Let $U = \bar{S}$ and $V = \bar{S}'$ where $\bar{S}' = \{\mathcal{H}(I'_0), \mathcal{H}(I'_1), \ldots, \mathcal{H}(I'_{n'})\}$ and $I'_i = (e'_i, e'_{i+1}), \forall i$. Thus, given U, we have found a PPT adversary $\mathcal{A}$ that finds another set V with probability $\geq neg(\lambda)$. This contradicts Theorem 1. Thus our assumption that Theorem 2 does not hold is false. Hence Theorem 2 is true. □

5 Our proposed VDCSE scheme

In this section, we propose our scheme which is forward secure and verifiable. Our scheme does not use any extra storage for verification. We see in the next that verifiability with $O(|W|)$ client storage is very easy for any single keyword search scheme.

5.1 Single keyword search DSE with $O(|W|)$ extra storage for verifiability

For *single keyword searches* [24] shows that when there is client storage of $O(|W|)$, verifiability can be achieved with any hash function for static data. Whereas, the same can be achieved with multiset hash ([27]) when the data is dynamic. These schemes are for single keyword search only. Besides, for dynamic data, when forward privacy is concerned, the solution [23, 24] shows how forward privacy can be achieved without extra client storage and still with keeping them verifiable. We see that if for every keyword $w \in W$, the client is able to store a digest of the set of identifiers $DB(w)$, then any multiset hash H solves the problem of verifiability for a single keyword search. The client can compute an aggregated hash using multiset hashing, which can be updated with every update done by the client. The client can recompute the aggregated hash when receives search results for the keyword and can match with the stored one. In such a scenario, since all computations are done by the client and nothing is outsourced to the cloud, there is no forgery. Thus, with $O(|W|)$ client storage, the client can verify the result, using multiset hashing, in any single keyword search DSE scheme, without affecting the forward or backward privacy.

A *conjunctive forward private keyword* search scheme can be either static or dynamic. A dynamic conjunctive search may have forward privacy. With non-trivial solution[1], [9] deals with verifiability when data is static and [7, 23] deal when they are dynamic. However, when forward privacy is concerned, the above solutions are not applicable. Also, they used at least $O(|W|)$ client storage as well. In a conjunctive dynamic SE scheme, if the client is able to store the accumulator corresponding to each keyword $w \in W$, then the client is able to verify the received result. It can verify whether all resulted identifiers are present in a keyword or not. Since this requires extra computation the client can outsource this computation to a proxy server too. Thus, the extra $O(|W|)$ client storage makes the scheme easier to verify the search result for any conjunctive dynamic SE scheme without effecting its forward or backward privacy if there is any.

DSE without $O(|W|)$ extra storage: We see that $O(|W|)$ client storage can make any conjunctive as well as single and Boolean keyword search scheme verifiable. Trivially, if the client issues a single keyword search token for each keyword in the conjunctive query and server returns the search result for each of them, Then it can compute the intersections of them to get the final result. This can also be done using [24] without extra client storage. However, the trivial approach has two issues. Firstly, it leaks the complete result for each keyword instead of the required. Secondly, searching for identifiers containing each keyword requires extra computation power. Thus, it is inefficient for a conjunctive search.

There are conjunctive DSE schemes that are either verifiable without forward privacy ([4]) or are forward private without verifiability ([26]). There are other conjunctive schemes which are neither forward private nor conjunctive. In the next, we have shown it is difficult to extend them to a verifiable DCSE scheme without that extra storage.

5.2 Difficulty in Extending existing schemes to a VDCSE scheme

It is an important question that whether existing conjunctive DSE without forward privacy [26] or without verifiability [4] can be extended with having both properties.

The key point of [26] is that modification of documents is not allowed here and the files are always unchanged. So, if we keep the accumulators at the leaf node of VBTree corresponding to every document, the accumulators will be always unchanged. However, the solution is not complete. If we keep the accumulators in the leaf then, for membership or non-membership proof, it must reveal what are the elements in the set. Thus the tree structure will be revealed and the scheme can not be forward private anymore. The cloud sends which tuple is not present in a node in the path. So we have to keep the accumulators in the nodes of the tree. However, if some new file is added then the complete path of the file may be revealed to add new elements in the nodes. *Thus, it is hard to extend [26] to be verifiable without extra client storage.*

Besides, [4] is for static data. So forward privacy is not applicable to it. If we extend the scheme to be dynamic then we can only try to make it forward private. We can see that it keeps the accumulator corresponding to every keyword on the cloud-side. When an update happens, the cloud has to update the accumulators too, and updating it reveals whether the keyword was searched previously. So, its extension to forward private is not possible in this way.

5.3 Overview of our proposed scheme Blasu

In this section, we present a generic forward private conjunctive DSE scheme with verifiability that makes any forward private single keyword search scheme to conjunctive one. However, we want to reduce this extra client storage for verifiability. Our proposed forward private conjunctive DSE scheme with verifiability does not use any extra client storage for verifiability. Here, we give a short overview of our scheme Blasu.

In most of the conjunctive schemes, including ours, the least-frequency method is considered (exception [26]). In this method, the least frequent keyword is taken and its result is found. Then for each resulting document, the presence of all other keywords is checked. For example, given a query $\hat{w} = \{w_1, w_2, \ldots, w_k\}$, let $R_{w_1} = \{id_1^{w_1}, id_2^{w_1}, \ldots, id_{n_{w_1}}^{w_1}\}$ be the single keyword query result for the lowest frequent keyword w_1. The

[1] A trivial solution is downloading search results for all keywords present in a conjunctive query and taking the intersection of them at client-side

frequency of a keyword is the number of documents that contain it. The server computes R_{w_1} and checks if id^1_i, for $1 \leq i \leq n_{w_1}$, contains all the keywords in $\hat{w} \setminus \{w_1\}$ and includes it in the search result $R_{\hat{w}}$, in the case does.

Our proposed scheme Blasu uses a forward private DSE scheme Σ as a black box.At the time of building two data structures, a hash table, and a DIA tree are built in addition to the encrypted index. For each keyword-document pair, a unique element is created. The element stores a signature generated using the corresponding keyword and the document identifier. It is kept in the hash table that gives it efficient access. After all the elements are generated, a DIA tree is built on them.

To search, we use the least-frequency approach. The client first generates a search token and sends it to the cloud. The cloud performs a search using Σ for a minimal frequent keyword. Then, for each document identifier in the result, the cloud checks its existence of other keywords. It returns the documents, each of which contains all searched keywords together with proof of its correct execution. The elements and DIA tree help the cloud to return the proof. To update a new keyword-document pair, the client generates the corresponding element and sends it to the cloud, which then updates both the table and the tree accordingly.

5.4 Technical Details

There are three phases in our proposed VDCSE scheme Ψ which is an algorithm tuple (`VCKeyGen`, `VCBuildIndex`, `VCSrchTknGen`, `VCSearchCD`, `VCSearchCT`, `VCUpdtTknGen`, `VCUpdate`)– initialization, search and update. The interaction between the entities, during those phases, are shown in Fig 5. The phases are described as follows.

Initialization: It is divided into two parts– key generation and building an encrypted search index, given as below.

Key Generation: is given Algo. 1. Let $\mathcal{E} = (\texttt{Enc}, \texttt{Dec})$ be a CPA-secure symmetric encryption scheme with key-space $\{0,1\}^\lambda$. Given some security parameter λ, the key K_Σ is generated for Σ. Moreover, three λ-bit stings K_s, K_t and $K_{\bar{s}}$ are picked at random to use them as secret.

Algorithm 1: Ψ.`VCKeyGen`(1^λ)

```
$K_\Sigma \leftarrow \Sigma.\texttt{KeyGen}(1^\lambda)$ ; $(sk, pk) \leftarrow \mathcal{S}.\texttt{KeyGen}(1^\lambda)$ ;
$K_s, K_t, K_{\bar{s}} \leftarrow \{0,1\}^\lambda$; $s \leftarrow \{0,1\}^\lambda$ /*for DIAT*/;
return $K_\Psi = (K_s, K_t, K_{\bar{s}}, sk, pk, K_\Sigma, s)$;
```

For each keyword, K_s is used as a key, together with the keyword, for generating a seed. This seed is used to generate a sequence of random numbers. Similarly, for each keyword, K_t helps to generate a tag that helps to find some random positions in a table (T_{sig}). Whereas $K_{\bar{s}}$ generates a unique key for the symmetric encryption scheme $\mathcal{E}$. A BLS signature scheme $\mathcal{S} = (\texttt{KeyGen}, \texttt{Sign}, \texttt{Verify})$ is generated together with a tuple $tup = (p, \mathbb{G}, \mathbb{G}, \hat{e}, g)$ and a key pair (sk, pk). A λ-bit secret key s is chosen for DIA tree DT. Finally, $K_\Psi = (K_s, K_{\bar{s}}, K_t, sk, pk, K_\Sigma)$ is returned

Encrypted Index Building: is given in Algo 2. Instead of $DB(w)$, for each $w \in W$, we consider $DB(w) \cup \{id^w_0\}$, where $\{id^w_0\}$ is a random unassigned identifier. Doing so prevents the server to return an empty set of identifiers. Whenever the cloud returns the actual file it neglects the first identifier. Without loss of generality, we take $DB(w) = DB(w) \cup \{id^w_0\}$ and $\mathcal{DB} = \{DB(w) \cup \{id^w_0\} : w \in W\}$. In rest of the paper, we consider the same.

To generate an encrypted search index, the client takes an empty hash table T_{sig} where it keeps a key-value pair $(pos^w_i, (\sigma^w_i, v^w_i\ ))$ for each keyword-doc pair (w, id^w_i). The key pos^w_i indicates the position in the table where value $(\sigma^w_i, v^w_i\ )$ keeps two things– a signature σ^w_i for the pair and encrypted file-sequence-number v^w_i for the keywords. A symmetric key encryption scheme Enc can be taken to get v^w_i. Finally, the client builds a DIA tree DT for the set P of all such positions. The tree DT is constructed with `DIAT.BuildTree`(tup, s, P) as described in Section 4. The root of the tree is kept at client-side. Moreover, the documents are kept encrypted. The encrypted index ξ for them is generated using Σ. Here $\gamma = \{\xi, T_{Sig}\}$ and $\mathcal{I} = DT$.

Search Phase: The search phase consists of three steps. At first, the client generates a search token to search for a set of keywords and sends it to the cloud server. Then, the cloud server performs a search on the encrypted database and generates a proof of the search result. The client also generates a proof of the received result and gives it to the auditor. Finally, an auditor verifies the result and the proofs.

Search token generation: Given a query $\hat{w} = \{w_1, w_2, \ldots, w_{|\hat{w}|}\}$, the client first generates search token $\tau^\Sigma_{w_1}$ for w_1 (the lowest frequent keyword), according to the base searchable encryption scheme Σ (Algo. 3). This

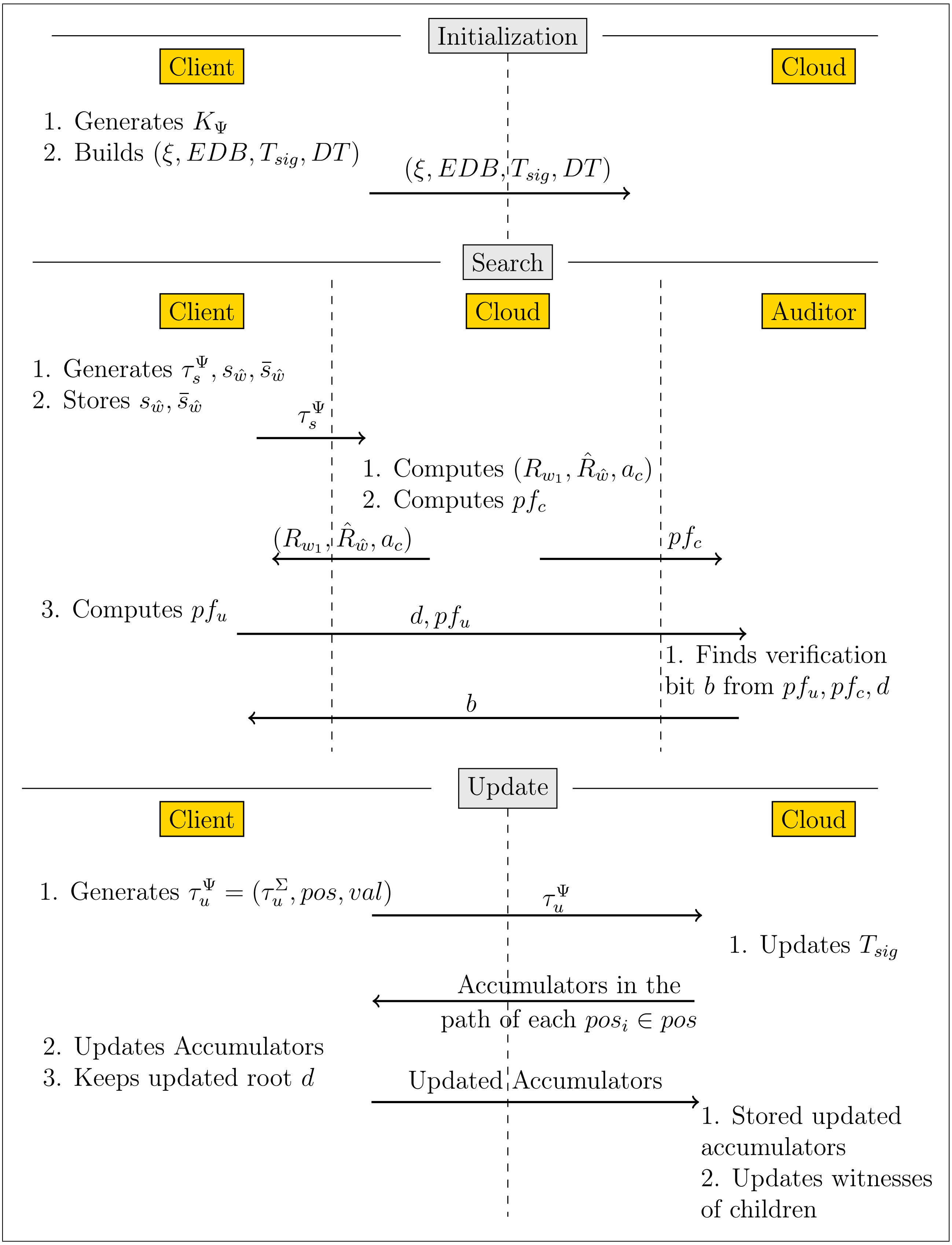

Initialization
Client
Cloud
1. Generates K_Ψ
2. Builds (ξ, EDB, T_{sig}, DT)
(ξ, EDB, T_{sig}, DT)
Search
Client
Cloud
Auditor
1. Generates $\tau_s^\Psi, s_{\hat{w}}, \bar{s}_{\hat{w}}$
2. Stores $s_{\hat{w}}, \bar{s}_{\hat{w}}$
τ_s^Ψ
1. Computes $(R_{w_1}, \hat{R}_{\hat{w}}, a_c)$
2. Computes pf_c
$(R_{w_1}, \hat{R}_{\hat{w}}, a_c)$
pf_c
3. Computes pf_u
d, pf_u
1. Finds verification bit b from pf_u, pf_c, d
b
Update
Client
Cloud
1. Generates $\tau_u^\Psi = (\tau_u^\Sigma, pos, val)$
τ_u^Ψ
1. Updates T_{sig}
Accumulators in the path of each $pos_i \in pos$
2. Updates Accumulators
3. Keeps updated root d
Updated Accumulators
1. Stored updated accumulators
2. Updates witnesses of children

Figure 5: Interaction between entities in different phases

```
Algorithm 2: $\Psi.\texttt{VCBuild}(\mathcal{DB}, K_\Psi)$
$T_{sig} \leftarrow$ An empty list of size $|\mathcal{W}|$ ;
for $w \in \mathcal{W}$ do
    $s_w \leftarrow F(K_s, w)$; $tag_w \leftarrow F(K_t, w)$; $\bar{s}_w \leftarrow F(K_{\bar{s}}, w)$;
    for $i = 0$ to $c_w(= |DB(w)|)$ do
        $r_i^w \leftarrow R(s_w||i)$;
        $m_i^w \leftarrow r_i^w.id_i^w \mod q$; $pos_i^w \leftarrow F(tag_w, id_i^w||i)$;
        $\sigma_i^w \leftarrow \mathcal{S}.\texttt{Sign}(sk, m_i^w)$ ; $v_i^w \leftarrow \mathcal{E}.\texttt{Enc}(\bar{s}_w, i)$ ;
        $T_{sig}[pos_i^w] \leftarrow (\sigma_i^w, v_i^w)$;
    end
end
$P = \{pos_i^w : w \in \mathcal{W}$ and $i = 0, 1, \ldots, n_w\}$;
$(DT, d) \leftarrow \texttt{DIAT.BuildTree}(tup, s, P)$;
$\mathcal{DB} = \{DB(w) : w \in W\}$;
$(\xi, EDB) \leftarrow \Sigma.Build(\mathcal{DB}, K_\Sigma)$ ;
Client keeps the root digest $d$;
return $(\xi, EDB, T_{sig}, DT)$ to cloud;
```

helps to find file identifiers that contain w_1. Then, it generates corresponding set of tags $\{tag_{w_1}, tag_{w_2}, \ldots, tag_{w_{|\hat{w}|}}\}$ for all keywords. These tags help to find whether the keyword-document pairs exist without revealing the actual keyword. Additionally, $s_{\hat{w}} = \{s_{w_1}, s_{w_2}, \ldots, s_{w_{|\hat{w}|}}\}$ and $\bar{s}_{\hat{w}} = \{\bar{s}_{w_1}, \bar{s}_{w_2}, \ldots, \bar{s}_{w_{|\hat{w}|}}\}$ are generated by the client. Finally, $\tau_s^\Psi = (\tau_{w_1}^\Sigma, tag_{w_1}, tag_{w_2}, \ldots, tag_{w_{|\hat{w}|}})$ is issued as a search token for the cloud and $s_{\hat{w}}, \bar{s}_{\hat{w}}$ are stored at client-side.

```
Algorithm 3: $\Psi.\texttt{VCSrchTknGen}(\hat{w}, K_\Psi)$
$\{w_1, w_2, \ldots, w_{|\hat{w}|}\} \leftarrow \hat{w}$, where $w_1$ is minimal frequent ;
$(K_s, K_t, K_{\bar{s}}, sk, pk, K_\Sigma, s) \leftarrow K_\Psi$;
$\tau_{w_1}^\Sigma \leftarrow \Sigma.\texttt{SearchToken}(w_1, K_\Sigma)$;
for $i = 1$ to $i = |\hat{w}|$ do
    $tag_{w_i} \leftarrow F(K_t, w_i)$; $s_{w_i} \leftarrow F(K_s, w_i)$; $\bar{s}_{w_i} \leftarrow F(K_{\bar{s}}, w_i)$;
end
$\tau_s^\Psi \leftarrow (\tau_{w_1}^\Sigma, tag_{w_1}, \ldots, tag_{w_{|\hat{w}|}})$;
$s_{\hat{w}} \leftarrow (s_{w_1}, s_{w_2}, \ldots, s_{w_{|\hat{w}|}})$; $\bar{s}_{\hat{w}} \leftarrow (\bar{s}_{w_1}, \bar{s}_{w_2}, \ldots, \bar{s}_{w_{|\hat{w}|}})$;
return $\tau_s^\Psi$ for cloud and $(s_{\hat{w}}, \bar{s}_{\hat{w}})$ only for client;
```

Search and proof generation: After receiving the search token τ_s^Ψ from the client, at first, the cloud finds single keyword search result R_{w_1}=$\{id_0^{w_1}$, $id_1^{w_1}$, $\ldots$, $id_{n_{w_1}}^{w_1}$ $\}$ for the keyword w_1 using Σ. Then from R_{w_1} and the tag tag_{w_1}, it finds the position of the keyword-file pairs corresponding to w_1 and retrieves signatures of them from the table T_{sig}. After that, it multiplies them as an aggregate signature for w_1 which is treated proof $pf_c^{(0)}$ for w_1.

Then, for each file in R_{w_1}, it checks whether other keywords are present in the file by verifying the existence of the keyword-file pairs. To verify them, corresponding positions are regenerated and checked whether the table T_{sig} contains them. If for some jth file id, the position does not exist, the cloud computes non-membership proof $pf_c^{(i)}$ for that positions. If all keywords are contained in jth file, then the product of their signatures is returned as proof corresponding to the keyword, and the set $a_c^{(j)}$ of v_i^js are returned to the client. $\hat{R}_{\hat{w}}$ keeps the identifiers that contain all keywords.

Thus, for each file in R_{w_1}, if it is in $\hat{R}_{\hat{w}}$, then the cloud returns the product of the signatures corresponding to the keyword file pairs and the set of v_i^js. In case a file in R_{w_1} is not present in $\hat{R}_{\hat{w}}$, it returns a non-membership proof for the position corresponding to a non-existing keyword-file pair. Finally, the cloud server returns its part of the proof pf_c and $(\hat{R}_{\hat{w}}, X_{\hat{w}})$ to the client where $X_{\hat{w}} = (R_{w_1}, a_c)$ and a_c is the auxiliary information from cloud.

```
Algorithm 4: $\Psi$.VCSearchCD($\gamma, \tau_s^\Psi$)
Cloud Receives $\tau_s^\Psi$ from client ;
$(\tau_{w_1}^\Sigma, tag_{w_1}, tag_{w_2}, \ldots, tag_{w_{|\hat{w}|}}) \leftarrow \tau_s^\Psi$;
$R_{w_1} \leftarrow \Sigma.\mathtt{Search}(\xi, \tau_{w_1}^\Sigma)$;
$\{id_0^{w_1}, id_1^{w_1}, \ldots, id_{n_{w_1}}^{w_1}\} = R_{w_1}$;
for $i = 0$ to $n_{w_1}$ do
    $pos_i^{w_1} \leftarrow F(tag_{w_1}, id_i^{w_1}||i)$;
    $\sigma_i' \leftarrow T_{sig}[pos_i^{w_1}][0]$; ;
end
$pf_c^{(0)} = \sigma' \leftarrow \prod_{i=0}^{n_w} \sigma_i'$; $\hat{R}_{\hat{w}} \leftarrow \Phi$;
if $|\hat{w}| = 1$ return $(R_{w_1}, pf_c^{(1)})$
for $j = 1$ to $n_{w_1}$ do
    $flag = 0$;
    for $i = 2$ to $|\hat{w}|$ do
        $pos_j^{w_i} \leftarrow F(tag_{w_i}, id_j^{w_1})$;
        if $[T_{sig}[pos_j^{w_i}]] = \perp$ then
            $pf_c^{(j)} \leftarrow \mathtt{DIAT.Search}(DT, pos_j^{w_i})$;
            $a_c^{(j)} \leftarrow pos_j^{w_i}$; $flag = 1$ ;
            break;
        end
        $(\sigma_j^i, v_j^i) \leftarrow T_{sig}[pos_j^{w_i}]$;
    end
    if $flag = 0$ then
        $pf_c^{(j)} \leftarrow \prod_{i=2}^{|\hat{w}|} \sigma_j^i$; $a_c^{(j)} \leftarrow (v_j^2, \ldots, v_j^{|\hat{w}|})$;
        $\hat{R}_{\hat{w}} \leftarrow \hat{R}_{\hat{w}} \cup \{id_j^{w_1}\}$;
    end
end
$pf_c = (pf_c^{(0)}, pf_c^{(1)}, \ldots, pf_c^{(n_{w_1})})$;
$a_c = (a_c^{(1)}, a_c^{(2)}, \ldots, a_c^{(n_{w_1})})$ ; $X_{\hat{w}} = (R_{w_1}, a_c)$;
return $pf_c$ and $(\hat{R}_{\hat{w}}, X_{\hat{w}})$ ;
```

After receiving$(\hat{R}_{\hat{w}}, X_{\hat{w}} = (R_{w_1}, a_c))$, the client generates its part of the proof pf_u. For w_1, it regenerates all the random numbers $m_i^{w_1}$ for each of the files in R_{w_1}. Then it generates the product of them as $m_0 = \sum_{i=0}^{n_{w_1}} m_i^{w_1} \mod p$ (see step 3 to step 10 in in Algo. 5).

For each file $id \in R_{w_1} \setminus \{w_1\}$, if $id \in \hat{R}_{\hat{w}}$, the client decrypts the encrypted numbers v_i^js, generates random numbers corresponding to each keyword and calculates the product m_i of them as $pf_u^{(i)}$. This acts as membership proof of all the keywords in the file. So, we do not have to generate a separate proof for all keyword-file pairs. In case, $id \notin \hat{R}_{\hat{w}}$, the client keeps $pf_u^{(i)}$ as null. This is because the cloud already keeps non-membership proof for them.

The auditor (or any third party) verifies the search result by taking pf_c from the cloud, and pf_u, $\hat{R}_{\hat{w}}$ and d form the client. The algorithm is given in Algo. 5.

Verification: The auditor verifies for w_1 as well as for each files in R_{w_1}. There are two cases in verification. For the identifiers $\in \hat{R}_{\hat{w}}$, containing all keywords, it verifies $\mathcal{S}.\mathtt{Verify}(pk, pf_u[k][1], pf_c[k])$. For the identifiers $\notin \hat{R}_{\hat{w}}$ that does not contains some keyword, non-membership proof, for corresponding *pos*, is verified with `DIAT.VerifySearch`. auditor returns *accept* only when all get success (see Algo. 6).

```
Algorithm 5: $\Psi$.VCSearchCT$(\gamma, \tau_s^{\Psi})$
Client Receives $(\hat{R}_{\hat{w}}, X_{\hat{w}} = (R_{w_1}, a_c))$ ;
$(a_c^{(1)}, a_c^{(2)}, \dots, a_c^{(n_{w_1})}) \leftarrow a_c$;
$\{id_0^{w_1}, id_1^{w_1}, \dots, id_{n'_{w_1}}^{w_1}\} \leftarrow R_{w_1}$; $n_{w_1} \leftarrow C[w_1]$ ;
if $n_{w_1} \neq n'_{w_1}$ then return reject;
$\{s_{w_1}, s_{w_2}, \dots, s_{w_{|\hat{w}|}}\} \leftarrow s_{\hat{w}}$ (see Algo. 3);
for $i = 0$ to $n_{w_1}$ do
    $r_i^{w_1} \leftarrow R(s_{w_1}||i)$;
    $m_i^{w_1} \leftarrow id_i^{w_1}.r_i^{w_1} \mod p$;
end
$pf_u^{(0)} = m_0 = \sum_{i=0}^{n_{w_1}} m_i^{w_1} \mod p$ ;
for $j = 1$ to $n_{w_1}$ do
    if $id_j^{w_1} \notin \hat{R}_{\hat{w}}$ then $pf_u^{(i)} = (0, a_c^{(j)})$ ;
    else
        $(v_j^2, v_j^3, \dots, v_j^{|\hat{w}|}) \leftarrow a_c^{(j)}$;
        for $i = 2$ to $|\hat{w}|$ do
            $k_i \leftarrow \mathcal{E}$.Dec$(\bar{s}_{w_i}, v_j^i)$;
            $r_i^j \leftarrow R(s_{w_i}||k_i)$;
            $m_i^j \leftarrow \hat{R}_{\hat{w}}[i].r_i^j \mod p$;
        end
        $m_j = \sum_{i=2}^{|\hat{w}|} m_i^j \mod p$;
        $pf_u^{(j)} = (1, m_j)$
    end
end
return $pf_u = \{pf_u^{(0)}, pf_u^{(1)}, \dots, pf_u^{(n_{w_1})}\}$;
```

```
Algo. 6: $\Psi$.Verify$(d, pf_u, pf_c, \hat{R}_{\hat{w}})$
Receives $pf_u$ from client and $pf_c$ from cloud;
for $k = 0$ to $n_{w_1}$ do
    if $pf_u[k][0] = 0$ then
        $b_v =$ DIAT. VerifySearch$(d, pf_c[k][0], pf_c[k][1], pf_u[k][1])$
    else
        $b_v \leftarrow \mathcal{S}$.Verify$(pk,\ pf_u[k][1], pf_c[k])$
    end
    if $b_v = failure$ return reject;
end
return accept ;
```

Updating the database: Given a new file f with a new identifier id, the client first generates an update token. From f, it extracts the set of keywords $\{w_1, w_2, \dots, w_{n_{id}}\}$, where n_{id} is the number of keywords present in f. It computes update token τ_u^{Σ} of the file according to Σ. For each keyword-doc pair, during update, corresponding entries in T_{sig} and the DIA tree DT are updated. Since, the client stores the frequencies of the keywords as the state, it retrieves them to compute key-value pairs for the table T_{sig}.

For each keyword w_i, it generates tag tag_{w_i}, s_{w_i} and $\bar{s}_{w_i}$ with the secret key. Then it generates key-value pair (pos_i, val_i) for every keywords w_i as given in Algo. 8. Finally, it returns $\tau_u^{\Psi} = (\tau_u^{\Sigma}, pos, val)$ to the cloud.

Algo. 7: Ψ.VCUpdtTkn(K_{Ψ}, st, f)

```
{w_1, w_2, ..., w_{n_id}} ∈ f;
(K_t, K_s, sk, pk, K_Σ) ← K_Ψ ;
τ_u^Σ ← Σ.UpdateToken(K_Σ, w_i, id) ∀i ∈ [n_id];
for i = 1 to n_id do
    s_{w_i} ← F(K_s, w_i); tag_{w_i} ← F(K_t, w_i);
    s̄_{w_i} ← F(K_s̄, w_i); n_{w_i} ← C[w_i];
    r_i ← R(s_{w_i}||(n_{w_i} + 1)); C[w_i] = C[w_i] + 1;
    m_i ← r_i.id  mod p; v_i ← ℰ.Enc(s̄_w, c_w + 1);
    σ_i ← 𝒮.Sign(sk, m_i^{w_i});
    pos_i ← F(tag_{w_i}, id); val_i = (σ_i, v_i)
end
pos ← {pos_1, pos_2, ..., pos_{n_id}};
val ← {val_1, val_2, ..., val_{n_id}};
return τ_u^Ψ = (τ_u^Σ, pos, val)
```

During the update phase, the cloud updates the file f according to Σ. Then it inserts key-value pairs in the table T_{sig}. Finally, after updating them in the database, it updates DT for each pos_i and returns corresponding proof of update for each position.

Algo. 8: Ψ.VCUpdate(T_{tag}, γ, op, f)

```
(τ_u^Σ, pos, val) = τ_u^Ψ;
Σ.Update(ξ, τ_u^Σ, op) ;
{pos_1, pos_2, ..., pos_n} ← pos; {val_1, val_2, ..., val_n} ← val;
for  i = 1to i = n do
    if (op=add) then T_sig[pos_i] ← val_i;
    else remove T_sig[pos_i];
    inpt ← (DT, s, pos_i, op, d) ;
    d' ← DIAT.Update(inpt)
end
Client keeps updated d';
return
```

Extra cost for verifiability: Building the index requires $O(N)$ key-value pairs computation and a DIA tree for a set of size N. During the search, the server has to compute $O((|\hat{w}| + 1).|R_{w_1}|)$ key-value pair, $O(|\hat{R}_{\hat{w}}|.|R_{w_1}|)$ multiplications in $\mathbb{G}$. It also has to compute $O(|\hat{w}|.(|R_{w_1}| - |\hat{R}_{\hat{w}}|))$ key-value pairs together with proofs of their non-membership. To generate proof at the client-side, the client only generates random numbers and computes the product of them which makes them very efficient for lightweight clients.

5.5 Security of our proposed scheme

5.5.1 Confidentiality

We see that the DIA tree is just an additional data structure that is get searched (updated) when the key of a key-value pair is searched (updated). So, it does not give any extra information about the encrypted database. (At the time of simulation, the simulator can also keep a similar tree based on the simulated database. The simulator only gives existential proof. So, in our security proof, we have not taken the accumulator part.) Else, suppose we have stored with a list of entries, then we build a simulator corresponding to that simulated database. In either case, the simulator must be there and so is the DIA tree. Since a verification phase is there, we can not return the random element in that case of the DIA tree. So, we can eliminate DT from leakage, but we should keep it with valid proof.

Leakage function: Let $\mathcal{L}_\Sigma = (\mathcal{L}_\Sigma^{bld}, \mathcal{L}_\Sigma^{srch}, \mathcal{L}_\Sigma^{updt})$ be the leakage function of Σ, then the leakage function $\mathcal{L}_\Psi$ = $(\mathcal{L}_\Psi^{bld}$, $\mathcal{L}_\Psi^{srch}$, $\mathcal{L}_\Psi^{updt})$ of Ψ is given as follows.

$$\begin{aligned}
\mathcal{L}_\Psi^{bld}(\mathcal{DB}) &= \{\mathcal{L}_\Sigma^{bld}(\mathcal{DB}), |T_{sig}|\} \\
\mathcal{L}_\Psi^{srch}(\hat{w}) &= \{\mathcal{L}_\Sigma^{srch}(w_1), \{(id_i^{w_1}, pos_i^{w_1}, \sigma_i^{w_1}) : i = 1, 2, \ldots, n_{w_1}\}, \\
&\quad \{(pos_j^i, \sigma_j^i) : \forall id_j \in R_{w_i}, w_i \in \hat{w}, i \neq 1\}\} \\
\mathcal{L}_\Psi^{updt}(w, id) &= \{id, \mathcal{L}_\Sigma^{updt}(w, id), pos^w, \sigma^w\}
\end{aligned}$$

Since we consider any forward private DSE scheme Σ which $\mathcal{L}_\Sigma$-secure against adaptive chosen keyword attack, we have the following theorem.

Theorem 3. *Let Σ = (KeyGen, Build, SearchToken, Search, UpdateToken, Update) be the forward private correct DSE scheme with leakage function $\mathcal{L}_\Sigma = (\mathcal{L}_\Sigma^{bld}, \mathcal{L}_\Sigma^{srch}, \mathcal{L}_\Sigma^{updt})$. If Σ is $\mathcal{L}_\Sigma$-secure against adaptive chosen keyword attack, under random oracle model, then for any adversary $\mathcal{A}_\Sigma$, there exists a simulator $\mathcal{S}_\Sigma$ which simulates Σ.*

The proof of the above theorem depends on the scheme Σ and can be seen in the corresponding paper (for example; [2]).

However, assuming the theorem we will proof confidentiality of Ψ. We show that Ψ is $\mathcal{L}_\Psi$-secure against adaptive dynamic chosen-query attacks in the random oracle model. The proof of confidentiality is given as follows.

Theorem 4. *If F is a PRF, R is a PRG and Σ is $\mathcal{L}_\Sigma$-secure against adaptive dynamic chosen-query attacks in the random oracle model, then Ψ is $\mathcal{L}_\Psi$-secure against adaptive dynamic chosen-query attacks, under q-SDH assumption, in random oracle model.*

Proof. We give the proof of the above theorem, according to Definition 8. It is sufficient to show that, for any PPT adversary $\mathcal{A}_\Psi$, there exists a simulator $\mathcal{S}_\Psi$, for which, the output of $\mathbf{Real}^\Psi_{\mathcal{A}_\Psi}(\lambda)$ and $\mathbf{Ideal}^\Psi_{\mathcal{A}_\Psi,\mathcal{S}_\Psi}(\lambda)$ are computationally indistinguishable.

Let $\mathcal{A}_\Sigma$ be the part of $\mathcal{A}_\Psi$ for Σ, then by Theorem 3, there exists a simulator $\mathcal{S}_\Sigma$ that simulates Σ. Therefore, it is to remain to construct a simulator $\mathcal{S}_\Psi$ to simulated extra data structure T_{sig} and query tokens (both search and update). Then, $\mathcal{S}_\Psi$ simulates as follows.

Simulating F: Simulation of the PRF F is done using a table T_F in random oracle model. For a given pair (x, y) of elements in $\mathtt{G}$, if $T_F[(x, y)] = \perp$, i.e. the corresponding entry does not exists, a random entry is kept as $T_F[(x, y)] \leftarrow \{0, 1\}^\lambda$ and finally $T_F[(x, y)]$ is returned.

Simulating Build: Given the leakage $\mathcal{L}_\Psi^{bld}(\mathcal{DB}) = \{\mathcal{L}_\Sigma^{bld}(\mathcal{DB}), |T_{sig}|\}$, $\mathcal{S}$ simulates two data structure EDB and the table T_{sig}. The DIA tree always accumulates the keys of the key-value pairs of T_{sig}. Again, T_{sig} is simulated with a table $\widetilde{T}_{sig}$. While simulating, let $\mathcal{S}_\Sigma$ returns $\widetilde{DB}$ while simulating EDB.

To keep the tags, a table $\widetilde{T}_{tag}$ taken by $\mathtt{Sim}_\Psi$. It stores a random λ-bit string for every keyword w. It acts as random oracle and returns $\widetilde{tag}_w \leftarrow \widetilde{T}_{tag}[w]$. A table $\widetilde{T}'_{sig}$ is also kept by $\mathtt{Sim}_\Psi$ to indicate whether an entry $\widetilde{T}_{sig}$ is queried or not.

The building of the data structures are simulated as follows.

1. $\widetilde{T}_{sig} \leftarrow \Phi$ and $\widetilde{T}'_{sig} \leftarrow \Phi$
2. For $i = 1$ to $i = |T_{sig}|$ do

```
        (a) $pos_i \xleftarrow{\$} \{0,1\}^\lambda$; $val_i \xleftarrow{\$} \{0,1\}^\lambda$
        (b) $\widetilde{T}_{sig}[pos_i] \leftarrow val_i$; $\widetilde{T}'_{sig}[pos_i] \leftarrow 0$
    3. $\widetilde{DB} \leftarrow \mathcal{S}_\Sigma(\mathcal{L}_\Sigma^{bld}(\mathcal{DB}))$
    4. $tup = (p, \mathbb{G}, \mathbb{G}, \hat{e}, g) \leftarrow \mathtt{BMGen}(1^\lambda)$ is generated for $\widetilde{\mathtt{DT}}$.
    5. $s \leftarrow \{0,1\}^\lambda$
    6. $\widetilde{\mathtt{DIAT}}.build(tup, s, \{pos_i : i = 1, \ldots, |T_{sig}|\})$
    7. return $(\widetilde{DB}, \widetilde{T}_{sig}, \widetilde{\mathtt{DT}})$ and keeps $(\widetilde{T}'_{sig}, s, p)$
```

Simulating search token: Let the search leakage $\mathcal{L}_\Psi^{srch}(\hat{w})$ is given.

A table T_F is taken to keep the positions for each keyword-file pair. Given a tuple $(\widetilde{tag}_w, id, i)$ it returns a position in the table. If the position is searched before, then it returns the previous one, else it allocate a new and return that. These table is kept at $\mathcal{S}_\Psi$. Let $\{w_1, w_2, \ldots, w_n\} \in \hat{w}$ where w_1 has least frequency. The complete simulation of search token is done by $\mathcal{S}_\Psi$ as follows.

```
    1. Receives $\tau_s^\Psi$ from client ;
    2. $\tau_{w_1}^\Sigma \leftarrow \mathtt{Sim}_\Sigma.\mathtt{SimSearch}(\mathcal{L}_{srch}^\Sigma(w_1))$;
    3. For $i = 1$ to $n'_{w_1}$
        (a) $\widetilde{tag}_{w_i} \xleftarrow{o} \widetilde{T}_{tag}[w_i]$;
        (b) $pos_i^{w_1} \xleftarrow{o} T_F[(\widetilde{tag}_{w_1}, id_i^{w_1}||i)]$;
        (c) $\sigma'_i \xleftarrow{o} \widetilde{T}_{sig}[\widetilde{pos}_i^{w_1}]$;
    4. $pf_c^{(1)} = \sigma' \leftarrow \prod_{i=1}^{n_w} \sigma'_i$;
    5. For $j = 1$ to $n'_{w_1}$
        (a) For $i = 2$ to $n_q$
            i. $\widetilde{pos}_j^{w_i} \xleftarrow{o} T_F[(\widetilde{tag}_{w_i}, id_j^{w_1})]$;
            ii. If $(T_{sig}[pos_j^{w_i}] = \bot)$
                • $\widetilde{T}_{sig}[\widetilde{pos}_j^{w_i}] \leftarrow \bot$ ; $a_c^{(j)} \leftarrow null$;
                • $pf_c^{(j)} \leftarrow \mathtt{DIAT.Search}$ $(\widetilde{DT}, pos_j^{w_i})$
                • continue for next $i$;
            iii. $(\sigma_j^i, v_j^i) \leftarrow \widetilde{T}_{sig}[pos_j^{w_i}]$;
        (b) $pf_c^{(j)} = \sigma_j \leftarrow \prod_{i=1}^{|\hat{w}|} \sigma_j^i$; $a_c^{(j)} \leftarrow (v_j^1, v_j^2, \ldots, v_j^{|\hat{w}|})$;
    6. $pf_c = (pf_c^{(1)}, pf_c^{(2)}, \ldots, pf_c^{(n_{w_1})})$;
    7. $a_c = (a_c^{(1)}, a_c^{(2)}, \ldots, a_c^{(n_{w_1})})$;
    8. Return $pf_c$ and $(R_{w_1}, \hat{R}_{\hat{w}}, a_c)$ ;
    9. return $\widetilde{\tau}_s^\Psi = (\widetilde{\tau}_\Sigma, \widetilde{tag}_w)$
```

Here, oracle access is indicated by "$\xleftarrow{o}$", if the elements is not empty, then it is returned, else a random element is allocated and then returned.

Simulating Update token Leakage function to add a document f, with identifier id, containing $\{w_1, w_2, \ldots, w_{n_w}\}$, is given by

$\mathcal{L}^{\Psi}_{updt}(f) = \{H'(id), \{(\mathcal{L}^{\Sigma}_{updt}(w_i, id)) : i = 1, 2, \ldots, n_{id}\}\}$.

1. For each keyword $w_i \in f$
   (a) $\widetilde{\tau}^i_u \leftarrow \mathtt{Sim}_{\Sigma}(\mathcal{L}^{\Sigma}_{updt}(w, id))$
   (b) $\widetilde{tag}_{w_i} \xleftarrow{o} \widetilde{T}_{tag}[w_i]$
   (c) $n_{w_i} \leftarrow C[w_i] + 1$
   (d) If $T_F[(\widetilde{tag}_{w_i}, id||(n_{w_i} + 1))]$ is not null,
      i. $\widetilde{pos}_i \leftarrow T_F[(\widetilde{tag}_{w_i}, id||(c_v + 1))]$
      Else
      i. $\widetilde{pos}_i \leftarrow$ a random pos_i such that $\widetilde{T}_{sig}[pos_i]$ is null
      ii. $T_F[(\widetilde{tag}_{w_i}, id||(n_{w_i} + 1))] \leftarrow \widetilde{pos}_i$; (iii) $\widetilde{T}'_{sig}[pos_i] \leftarrow 1$
   (e) $\widetilde{\sigma}_i \xleftarrow{\$} \mathbb{G}$
2. $\widetilde{pos} \leftarrow \{\widetilde{pos}_1, \widetilde{pos}_2, \ldots, \widetilde{pos}_{n_{id}}\}$; $\widetilde{\sigma} \leftarrow \{\widetilde{\sigma}_1, \widetilde{\sigma}_2, \ldots, \widetilde{\sigma}_{n_{id}}\}$
3. Return $\widetilde{\tau}^{\Psi}_u = (\widetilde{pos}, \widetilde{\sigma})$

Since, in each entry, the signature generated in T_{sig} is of the form $g^{\alpha m r}$ and corresponding entry in $\widetilde{T}_{sig}$ is of the form $g^{\alpha r'}$, where r is pseudo-random (as R is so) and r' is randomly taken, we can say that power of g in both are indistinguishable. Hence, T_{sig} and $\widetilde{T}_{sig}$ are indistinguishable.

Besides, the indistinguishability of $\widetilde{\tau}^{\Psi}_u$, $\widetilde{\tau}^{\Psi}_s$ with respect to τ^{Ψ}_s, τ^{Ψ}_u respectively follows from the pseudo-randomness of F. □

5.5.2 Soundness

We see that the server can cheat the cloud in four ways only, by returning– (1) incorrect number of identifiers in R_{w_1}, (2) some altered identifier in R_{w_1}, (3) some result $R_{w'}$ of other keyword set w_1 instead of R_{w_1} or (4) some subset of $\hat{R}_{\hat{w}}$. However, for each case, the cheating will be detected as follows.

1. Since, the client stores the frequency of each keyword as in the state of the database, it can identify incorrect frequency.
2. If any identifier is altered, m^j_i in Step 18 of Algo. 5 does not match and consequently, signature verification will be failed.
3. Signature is bounded with keywords by s_w. During proof generation at the client-side, it is regenerated. So signature verification will be failed if result set is changed.
4. Finally, if some subset of $\hat{R}_{\hat{w}}$ is returned, there will be some identifier $id \in \hat{R}_{\hat{w}}$ that is skipped in the returned set. So, the server has to find at least one $w \in \hat{w}$, such that (id, w) pair does not exist. However, since this is not true, the cloud server can not give non-membership proof for any such pair.

6 Comparison with existing schemes

Here we discuss a few previous schemes and compare them with our proposed one. Since we have shown that it is trivial to get a conjunctive scheme with client storage, we are considering the schemes that have no extra client storage for verifiability. We have summarized the comparison in Table 2.

We see that most of the works for verifiability are based on accumulators. The static scheme [4] used two types of accumulators: One for each keyword and another for the total keyword file pair. When the size of the member set increases, generating non-membership proof takes enormous time. Thus it becomes impractical for a large database. Moreover, it does not support dynamic data. Moreover, to verify, the client needs to compute $|R_{w_1}| \times |\hat{R}_{\hat{w}}|$ number of power of g and needs *two round* of communications.

The static scheme [21] used bilinear map for verifiability. Due to the existence of an auditor, the client does not need to compute anything for verification. However, during search token generation requires $O(|\mathcal{W}|)$ amount of storage as well as communication which is large when the keyword set os large.

Table 2: Comparison with existing conjunctive search SE schemes

Scheme name	Is Dyn?	Forward Secrecy	Verifiable	client Comm Cost to verify	client Comp cost to verify	client Comm Cost to update	client Comp cost to update
[4]	×	–	✓	$O(\lvert\mathcal{W}\rvert.\lvert\mathcal{DB}\rvert) + 2\mathtt{R}$	$O(\lvert R_{w_1}\rvert.\lvert\hat{R}_{\hat{w}}\rvert)\mathtt{Ex}$	–	–
[21]	×	–	✓	$O(\lvert\mathcal{W}\rvert) + 1\mathtt{R}$	$O(\lvert R_{\hat{w}}\rvert)(\mathtt{Ex} + \mathtt{Hs})$	–	–
[3]	×	–	✓	$O(\lvert R_{\hat{w}}\rvert) + 2\mathtt{R}$	$O(\lvert R_{\hat{w}}\rvert)(\mathtt{Ex} + \mathtt{Bm})$	–	–
[6]	×	–	✓	$O(\lvert\hat{w}\rvert \log\lvert\mathcal{W}\rvert) + 1\mathtt{R}$	$O(\lvert\hat{w}\rvert \log\lvert\mathcal{W}\rvert)(\mathtt{Ex} + \mathtt{Bm} + \mathtt{Hs})$	–	–
[26]	✓	✓	×	–	–	$O(\lvert f\rvert) + 1\mathtt{R}$	$O(\lvert f\rvert)\mathtt{Hs}$
[5]	✓	×	✓	$O(2\lvert\hat{w}\rvert(\log N + 1)) + 2\mathtt{R}$	$O(3\lvert\hat{w}\rvert \log N)\mathtt{M} + O(3\lvert\hat{w}\rvert)\mathtt{Ex}$	$O(\lvert f\rvert) + 1\mathtt{R}$	$O(\lvert f\rvert)\mathtt{Ex}$
[9]	✓	×	✓	$O(\lvert\hat{w}\rvert \log N) + 1\mathtt{R}$	$O(\lvert\hat{w}\rvert \log N)(\mathtt{Ex} + \mathtt{Bm})$	$O(\lvert f\rvert \log N) + 2\mathtt{R}$	$O(\lvert f\rvert \log N)\mathtt{Ex}$
Our Scheme	✓	✓	✓	$O(\lvert R_{w_1}\rvert) + 1\mathtt{R}$	$O(\lvert\hat{w}\rvert.\lvert R_{w_1}\rvert)\mathtt{Hs}$	$O(\lvert f\rvert) + 1\mathtt{R}$	$O(\lvert f\rvert)\mathtt{Hs} + O(\lvert f\rvert)\mathtt{Ex}$

$\mathtt{M}$– multiplication in G_T, $\mathtt{Ex}$– exponentiation in G, $\mathtt{R}$– rounds of communication, $\mathtt{Hs}$– number of hashes. $\mathtt{Bm}$– bilinear map. * in the complexities we considered most expensive operations only.

Mio et al. [3] is a static verifiable scheme that uses an interactive challenge-response method for verification. However, it requires $O(|\mathcal{W}|.|\mathcal{DB}|)$ cloud storage which is very large. Moreover, in verification, it requires 2 rounds of communication. It does not discuss the case when a subset of the result is returned.

The static scheme [6] used the cuckoo hashing. It similarly keeps an accumulator for each keyword and uses polynomial interpolation to prove the set intersection.

The dynamic scheme Li et al. [5] used an accumulator for each keyword. The accumulators are stored in the cloud in a Merkle tree that ensures the integrity of them. Updates of accumulators are not discussed. One big difference of [5] with us is that [5] computes membership proofs on the go when it requires. So, if the number of searches is high, this scheme will become slow. As in most of the verifiable dynamic schemes, it also has two rounds of communication during an update. Though [26] is forward private, it is not verifiable. It is also difficult to extend it to be verifiable.

[9] is a good dynamic scheme with verifiable support. It keeps an accumulator for each keyword and makes an accumulator tree for them. To verify whether the returned intersection set is correct it uses polynomial interpolation with FFT. This makes the computational cost higher ($O(N \log^2 N)$) for the server and makes the scheme unsuitable when the number of searches is high. Moreover, the scheme is not forward private too.

In our scheme, the verification can be done via any auditor, so if the client wants, it can outsource it. So, in the table bilinear map computation is ignored. Moreover, we see that most of the schemes use no extra client storage for verifiability, but they have different cloud storage requirements.

The works, including [9], [6] etc., that use intersection method, have higher complexity as they have to find all results first. The proof includes related information which increases communication cost too.

7 Conclusion

In this paper, we first designed an efficient authentication tree DIA tree. The tree can be scaled to large databases when searches are too frequent than updates. Then, we have proposed a conjunctive DSE scheme that is verifiable too. The scheme uses a forward private single keyword DSE scheme as the base. Moreover, our scheme does not use any extra client-storage for verifiability. We have used our designed DIA tree for verifiability. Later, we have shown that the scheme is practical comparing with existing schemes.

We can see that most of the verifiable conjunctive search schemes, including ours, use public-key encryption for verifiability though some single keyword verifiable schemes use symmetric one. Solving the same problem with a symmetric-key encryption technique is still a good open problem in this direction. Besides, though there is no practical attacks over non-backward private DSE schemes, still backward privacy should be present in future verifiable conjunctive search DSE schemes which is more challenging.

References

[1] 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.

[2] 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.

[3] Yinbin Miao, Jianfeng Ma, Fushan Wei, Zhiquan Liu, Xu An Wang, and Cunbo Lu. VCSE: verifiable conjunctive keywords search over encrypted data without secure-channel. *Peer-to-Peer Netw. Appl.*, 10(4):995–1007, 2017.

[4] Jianfeng Wang, Xiaofeng Chen, Shifeng Sun, Joseph K. Liu, Man Ho Au, and Zhi-Hui Zhan. Towards efficient verifiable conjunctive keyword search for large encrypted database. In Javier López, Jianying Zhou, and Miguel Soriano, editors, *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II*, volume 11099 of *Lecture Notes in Computer Science*, pages 83–100. Springer, 2018.

[5] Yuxi Li, Fucai Zhou, Yuhai Qin, Muqing Lin, and Zifeng Xu. Integrity-verifiable conjunctive keyword searchable encryption in cloud storage. *Int. J. Inf. Sec.*, 17(5):549–568, 2018.

[6] Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, and Refik Molva. Publicly verifiable conjunctive keyword search in outsourced databases. In *2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015*, pages 619–627, 2015.

[7] Wenhai Sun, Xuefeng Liu, Wenjing Lou, Y. Thomas Hou, and Hui Li. Catch you if you lie to me: Efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data. In *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015*, pages 2110–2118, 2015.

[8] Xiaoyu Zhu, Qin Liu, and Guojun Wang. A novel verifiable and dynamic fuzzy keyword search scheme over encrypted data in cloud computing. In *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*, pages 845–851, 2016.

[9] Shunrong Jiang, Xiaoyan Zhu, Linke Guo, and Jianqing Liu. Publicly verifiable boolean query over outsourced encrypted data. *IEEE Trans. Cloud Computing*, 7(3):799–813, 2019.

[10] Man Ho Au, Patrick P. Tsang, Willy Susilo, and Yi Mu. Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In *Topics in Cryptology - CT-RSA 2009, The Cryptographers' Track at the RSA Conference 2009, San Francisco, CA, USA, April 20-24, 2009. Proceedings*, pages 295–308, 2009.

[11] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Cryptographic accumulators for authenticated hash tables. Cryptology ePrint Archive, Report 2009/625, 2009.

[12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 965–976, 2012.

[13] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, pages 563–592, 2016.

[14] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1465–1482, 2017.

[15] Shifeng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 763–780, 2018.

[16] Qi Chai and Guang Gong. Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In *Proceedings of IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada, June 10-15, 2012*, pages 917–922, 2012.

[17] Rong Cheng, Jingbo Yan, Chaowen Guan, Fangguo Zhang, and Kui Ren. Verifiable searchable symmetric encryption from indistinguishability obfuscation. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 621–626, 2015.

[18] Wakaha Ogata and Kaoru Kurosawa. Efficient no-dictionary verifiable searchable symmetric encryption. In *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*, pages 498–516, 2017.

[19] Yinbin Miao, Jianfeng Ma, Ximeng Liu, Junwei Zhang, and Zhiquan Liu. VKSE-MO: verifiable keyword search over encrypted data in multi-owner settings. *SCIENCE CHINA Information Sciences*, 60(12):122105:1–122105:15, 2017.

[20] Azam Soleimanian and Shahram Khazaei. Publicly verifiable searchable symmetric encryption based on efficient cryptographic components. *Des. Codes Cryptography*, 87(1):123–147, 2019.

[21] Yinbin Miao, Jianfeng Ma, Ximeng Liu, Qi Jiang, Junwei Zhang, Limin Shen, and Zhiquan Liu. VCKSM: verifiable conjunctive keyword search over mobile e-health cloud in shared multi-owner settings. *Pervasive and Mobile Computing*, 40:205–219, 2017.

[22] Cheng Xu, Ce Zhang, and Jianliang Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 141–158. ACM, 2019.

[23] Kazuki Yoneyama and Shogo Kimura. Verifiable and forward secure dynamic searchable symmetric encryption with storage efficiency. In *Information and Communications Security - 19th International Conference, ICICS 2017, Beijing, China, December 6-8, 2017, Proceedings*, pages 489–501, 2017.

[24] Laltu Sardar and Sushmita Ruj. Fspvdsse: A forward secure publicly verifiable dynamic sse scheme. In *Provable Security - 13th International Conference, ProvSec 2019, Cairns, QLD, Australia, October 1-4, 2019, Proceedings*, pages 355–371, 2019.

[25] Meixia Miao, Jianfeng Wang, Sheng Wen, and Jianfeng Ma. Publicly verifiable database scheme with efficient keyword search. *Inf. Sci.*, 475:18–28, 2019.

[26] Zhiqiang Wu and Kenli Li. Vbtree: forward secure conjunctive queries over encrypted data for cloud computing. *VLDB J.*, 28(1):25–46, 2019.

[27] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *Advances in Cryptology - ASIACRYPT 2003, 9th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, November 30 - December 4, 2003, Proceedings*, pages 188–207, 2003.

[28] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004.

[29] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. *IACR Cryptology ePrint Archive*, 2016:62, 2016.